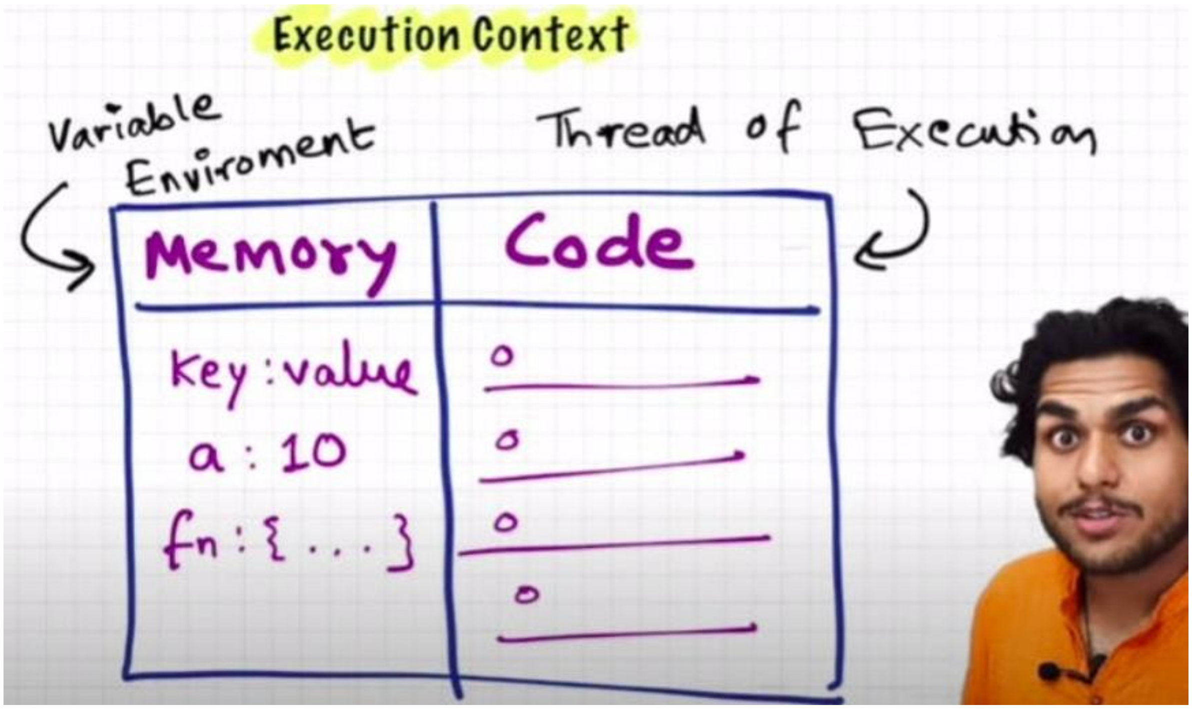
Chapter 1: Execution Context

Everything in JS happens inside the execution context. Imagine a sealed-off container inside which JS runs.



- 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. It will go to the next line once the current line has been finished executing.
  - Synchronous: - In a specific synchronous order.
  - Single-threaded: - One command at a time.

Chapter 2: How JS is executed inside Call Stack

* When a JS program is running, a **global execution context** is created.

* The execution context is created in two phases.

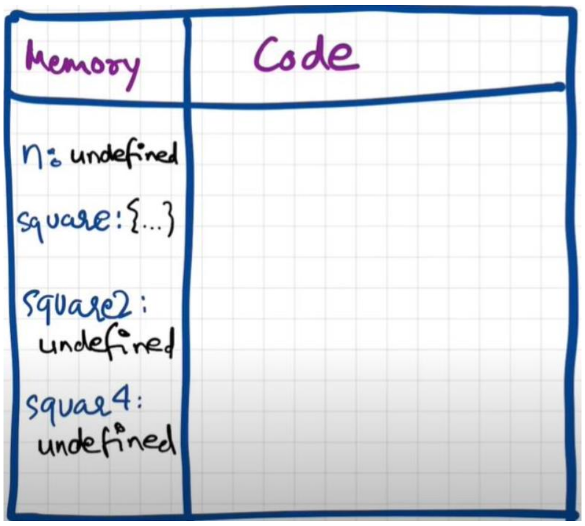
  * Memory creation phase - JS will allocate memory to variables and functions.
  * Code execution phase

* 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. In short, in **memory creation phase** JS skims through the entire code and allocates memory for variables and functions.

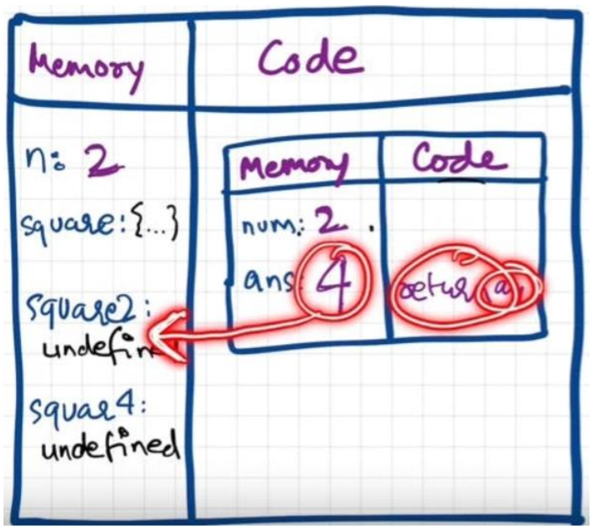
So, O/P will look something like



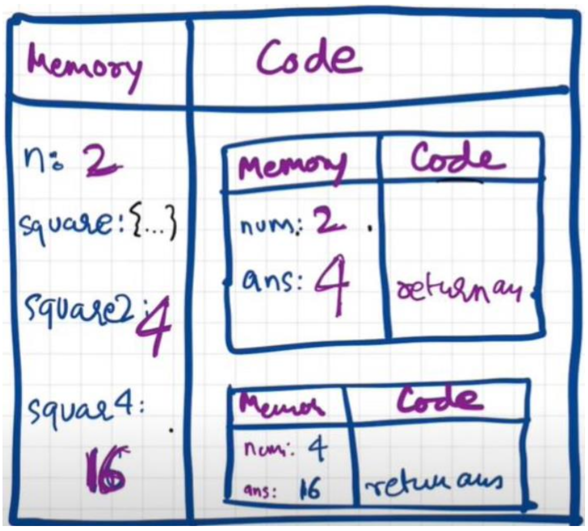
Now, in **2nd phase** i.e. code execution phase, JS engine starts going through the entire code line by line. As it encounters 'var n = 2', it assigns 2 to 'n'. Until now, the value of 'n' was undefined.

In the next line, JS encounters a function definition. For function, there is nothing to execute. As these lines were already dealt with in memory creation phase.

In the next line **var square2 = square(n)**, JS creates a new execution context for the function **square**. This execution context is known as **Function Execution Context (FEC)**. The way JS works is It creates a FEC when it comes across a function invocation / function call. Again, in FEC, in memory creation phase, JS engine allocate memory to num and ans, the two variables. And undefined is placed in them. Now, in code execution phase of this FEC, 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:



Call Stack -

* JavaScript manages code execution context creation and deletion with the help of **Call Stack**.

* **Call Stack** is a normal stack which follows LIFO (Last in First Out) principle.

* When JS executes a program, it creates a Global Execution Context (GEC) and pushes it to the bottom of the call stack.

* Within code execution phase of GEC When a function is invoked a new/Function Execution Context is created which is again pushed inside the stack. And the process goes on.

* When Function execution context is done with its two phases, it gets removed from the stack.

* Finally, when the program execution is finished, The GEC gets removed from the 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.

# Chapter 3: Hoisting in JavaScript (variables & functions)

* Let's observe the below code and its explanation:

```
getName(); // Namaste JavaScript
console.log(x); // undefined
console.log(getName); // f{...}
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 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 code execution phase, it executes whatever is asked. Here, as code 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 Reference Error: 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 phases, 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); // Uncaught Reference: x is not defined.
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.

```
console.log(x)  // x is not defined anywhere in the code. We get an error here as Not defined.
getName();     // Uncaught TypeError: getName is not a function. It’s a variable and undefined will be
stored in memory creation phase. we get this error because we are calling undefined().
console.log(getName);
var getName = function () {
   console.log("Namaste JavaScript");
}

// Note - The function expression and Arrow functions cannot be hoisted.
```

Execute below lines of code on console with debugging ON and check the Call stack.

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

# Chapter 4: Functions and Variable Environments

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

function a() {
 var x = 10; // local scope 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 and pushed into the Call Stack.

> Call Stack: GEC

* In phase 1 of GEC, variable x is initialised with undefined and a and b are initialised with their function definitions. In phase 2 of GEC, x is initialised with 1 and in subsequent lines a and b functions are invoked. As soon as JS encounters function invocations inside GEC, it creates a local or function execution Context for each of the function invocations or function calls. At present function a's execution context is created and pushed into the call stack.

> Call Stack: [GEC, a()]

* In phase 1 of a's local EC, a totally different variable x is initialised with undefined and in phase 2 it is assigned with 10 and printed in the console. After printing, no more commands to run, so function a's local EC is removed from both GEC and from Call stack.

> Call Stack: GEC

* In the next line, When JS encounters b function invocation, b's execution context is created. Same steps for b's Execution Context.

> Call Stack: [GEC, b()]

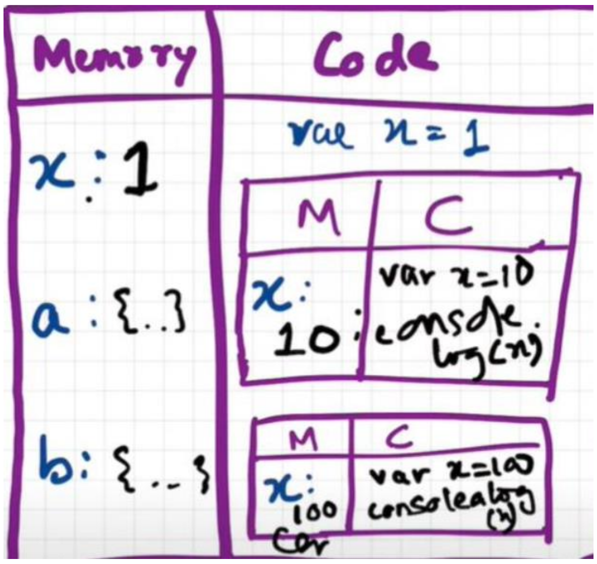
* when b's code execution phase is finished, there is no longer code exist for b to get executed and the local execution context for b is removed from the call stack. At this moment both functions execution contexts are removed from the stack.

> Call Stack: GEC

* In the next line JS encounters **console log (x)** which will print the value of x from the GEC into the console.JS cannot encounter further code after the current line execution, Thus GEC is removed from the call stack and JS program ends.

> Call Stack:

* reference:



# Chapter 5: Shortest JS Program, window & this keyword

* The shortest JS program is an empty file. Even in this case, JS engine does a lot of things. It creates the GEC which has memory phase and the code execution phase.

* JS engine creates something known as '**window**'. It is a global object for browser, 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 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.

example:

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

# Chapter 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 its own accord.

Chapter 7: The Scope Chain, Scope & Lexical Environment

* **Scope** in JavaScript is directly related to **Lexical Environment**. Scope refers to the current context of the code, which determines the accessibility of variables.

* 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 the variable b outside the function scope.
}
var b = 10;
a();
```

// In this case JS first tried to find b inside the local scope or function scope/a function scope and did not find b, Then It tried to find b in its parent scope / global scope and found it.

CASE 2

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

// In this case JS first tried to find b inside the local scope or function scope/c function scope and did not find b, then it tried to find b in its parent scope which is 'a function scope' and found it.

CASE 3

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

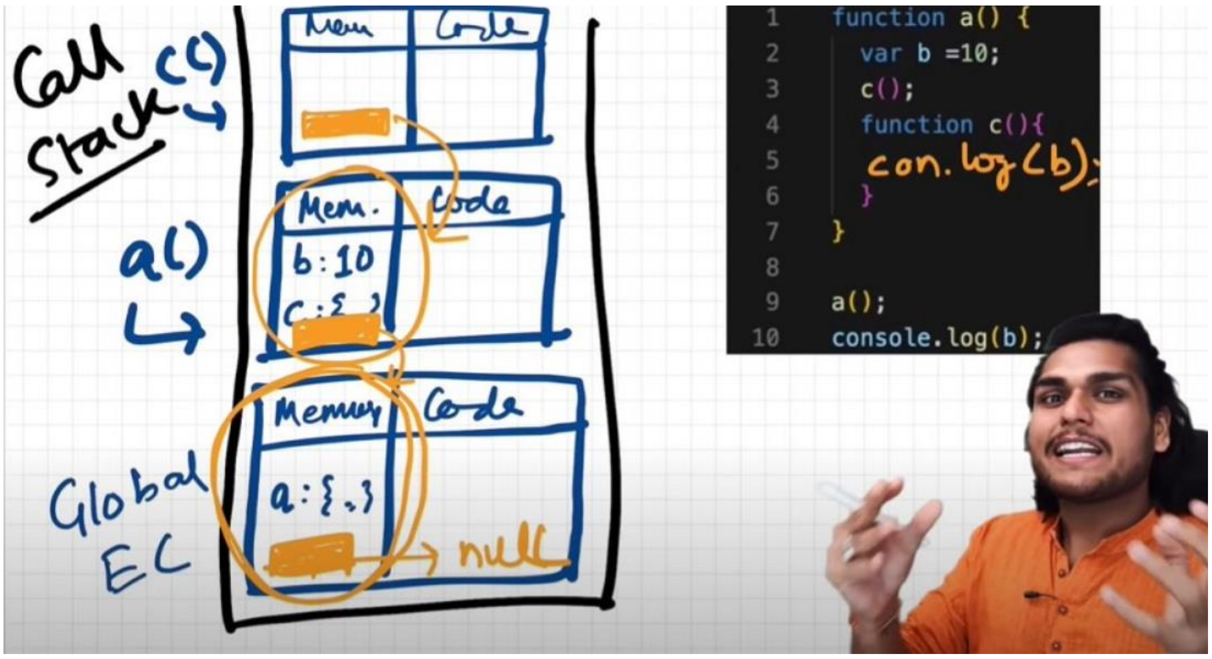
// In this case JS first tried to find b inside the local scope or function scope/c function scope and found it.

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:
   CallStack = [GEC, a(), c()]
   Now let's also assign the memory sections of each execution context in CallStack.
   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]]



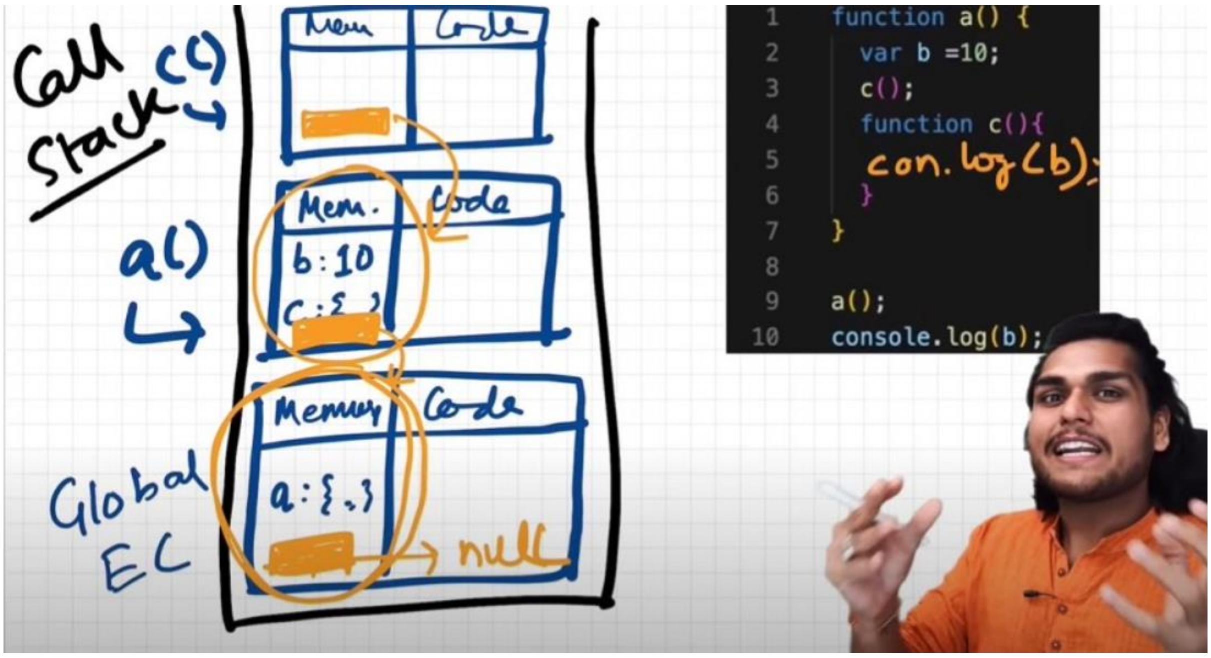
```
function a() {
  var b =10;
  c();
  function c(){
  }
}

a();
console.log(b);
```

* Scope = Environment

* **Lexical Environment** = Local memory + Lexical Environment of its parent or Surrounding Environment

* **Lexical Scope**: The ability of a function to access variables from the parent Scope.

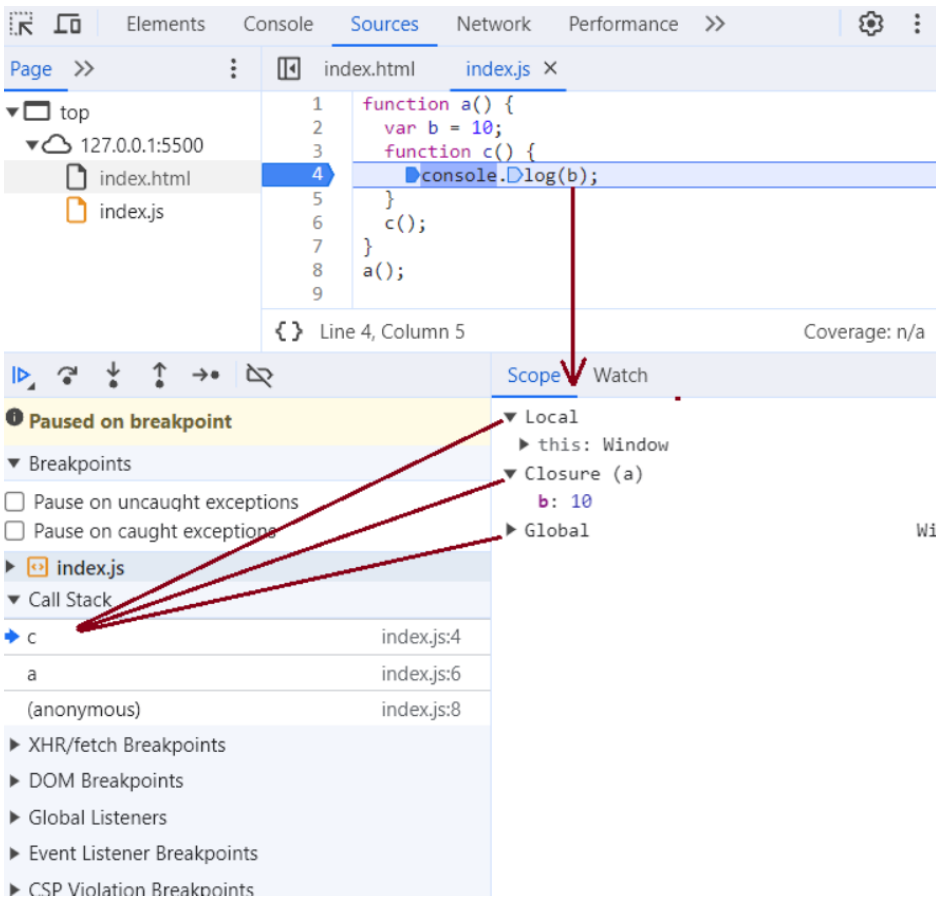
* Whenever an Execution Context is created, a Lexical environment (LE) is also created and is referenced in the local Execution Context (in memory space).



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

a();
console.log(b);
```

orange mark is nothing but the reference pointing to lexical Environment of its parent.

* **Scope Chain** is the chain of lexical environments. That means If JS cannot find a variable in a current scope then it will check the same variable in its parent scope or Lexical scope. If the variable is not present either in the parent scope, it will check the same variable in parent's parent scope and this search will continue till global scope. The way of finding the variables across the lexical scopes form a Scope Chain.



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

```
Line 4, Column 5                                   Coverage: n/a

Paused on breakpoint                    Scope  Watch
Breakpoints                             ▼ Local
☐ Pause on uncaught exceptions            ▶ this: Window
☐ Pause on caught exceptions            ▼ Closure (a)
▶ index.js                                  b: 10
▼ Call Stack                            ▶ Global                      Wi
➡ c                      index.js:4
  a                      index.js:6
  (anonymous)            index.js:8
▶ XHR/fetch Breakpoints
▶ DOM Breakpoints
▶ Global Listeners
▶ Event Listener Breakpoints
▶ CSP Violation Breakpoints
```

In the above figure, inside scope section **local -> closure -> global** forms a **Scope Chain** and in this chain, JS is trying to find out the value of variable b and it finds it in the closure (function a's scope)

```
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 physical location in source code.

```
Global {
  Outer {
    Inner
  }
}
// Inner is surrounded by lexical scope of Outer
```

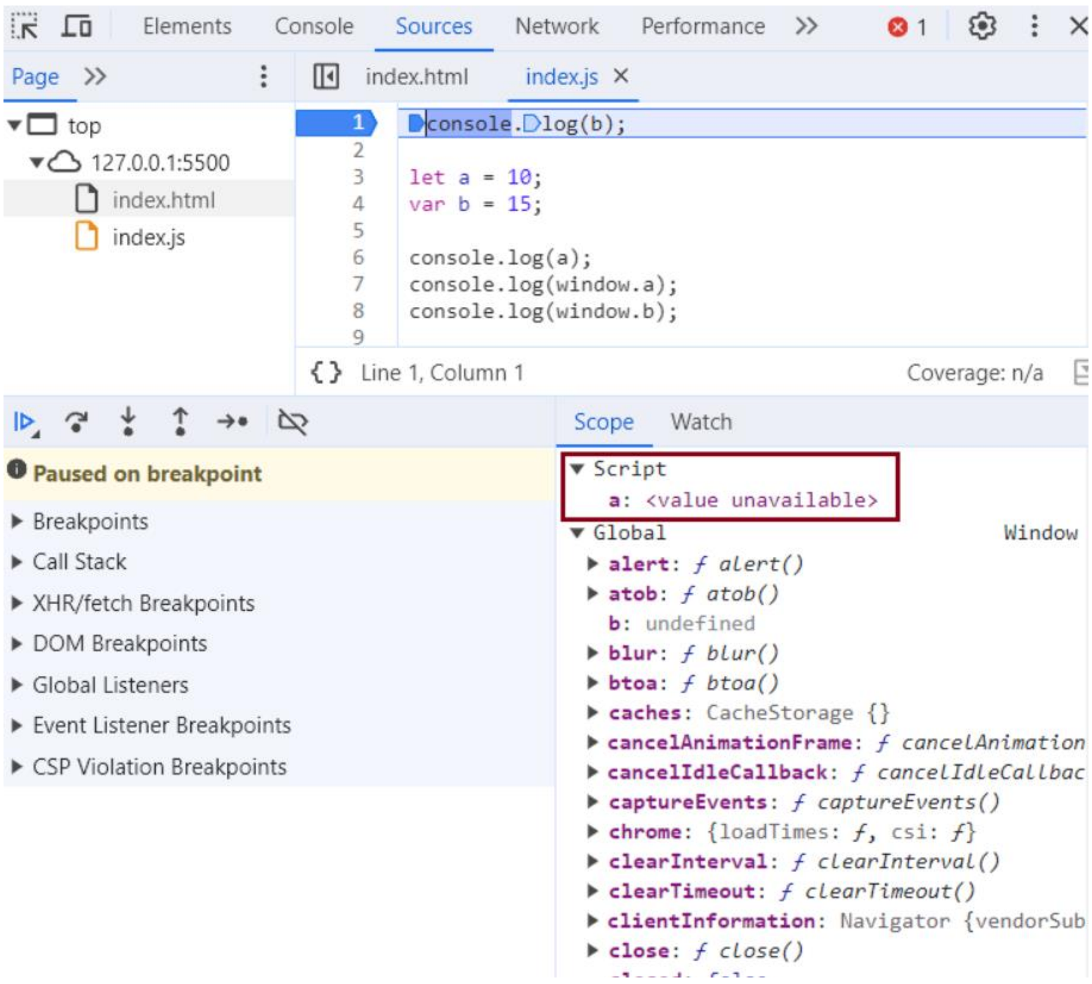
* **TLDR**; An inner function can access variables which are in outer functions even if inner function is nested deep. In any other case, a function can't access variables not in its scope.

Chapter 8: let & Const in JS, Temporal Dead Zone

* **let and const** declarations are hoisted. But it's different from **var** declaration.

```
console.log(a); // ReferenceError: Cannot access 'a' before initialization
console.log(b); // prints undefined as expected because of 'var' hoisting

let a = 10;
var b = 15;

console.log(a); // 10
console.log(window.a); // undefined
console.log(window.b); // 15
```

It looks like let isn't hoisted, **but it is**, let's understand
* In the above example, var b is hoisted in global scope with a value *undefined* whereas let a is hoisted in a separate memory object or 'Script' scope with a value <value unavailable>. Within this scope, value of 'a' won't be available until it is initialised. So, the time between **let variable 'a' is hoisted** and **'a' is initialised** is called Temporal Dead Zone. If we try to get access to let variable 'a' within this time frame we get a reference error. Same rule applies to const variable as well.



```
console.log(b);

let a = 10;
var b = 15;

console.log(a);
console.log(window.a);
console.log(window.b);
```

```
▼ Script
    a: <value unavailable>
▼ Global                                   Window
  ▶ alert: ƒ alert()
  ▶ atob: ƒ atob()
    b: undefined
  ▶ blur: ƒ blur()
  ▶ btoa: ƒ btoa()
  ▶ caches: CacheStorage {}
  ▶ cancelAnimationFrame: ƒ cancelAnimation
  ▶ cancelIdleCallback: ƒ cancelIdleCallbac
  ▶ captureEvents: ƒ captureEvents()
  ▶ chrome: {loadTimes: ƒ, csi: ƒ}
  ▶ clearInterval: ƒ clearInterval()
  ▶ clearTimeout: ƒ clearTimeout()
  ▶ clientInformation: Navigator {vendorSub
  ▶ close: ƒ close()
```

* **Temporal Dead Zone**: Time since when the let variable was hoisted until it is initialized.
  - 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)

* **Reference Error** are thrown when variables are in temporal dead zone.

* **Syntax Error** doesn't even let us run single line of code.

  *
```
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 stricter than **let**.

```
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 accessed.

  * 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.

Note

If a variable is not present in the window object and if we try to get access to the variable through window.VarName Then we get undefined as output.

Are let and const variables are hoisted?

Like var declarations, let and const declarations are also hoisted. It looks like it is not hoisted but internally it is hoisted. Only difference is var declarations are hoisted in global scope but let and const declarations are hoisted in script scope which is a temporary JavaScript object scope.

In var declaration we can access the variable before the variable is initialised but in **let** and **const** declaration we can't access the variable before it is initialised because from the time when let or const variables are declared till they are initialised is called Temporal Dead zone and We can’t access let and const variables with in the Temporal Dead Zone.

# Chapter 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 {...}.

Why do we need to group all these statements together?

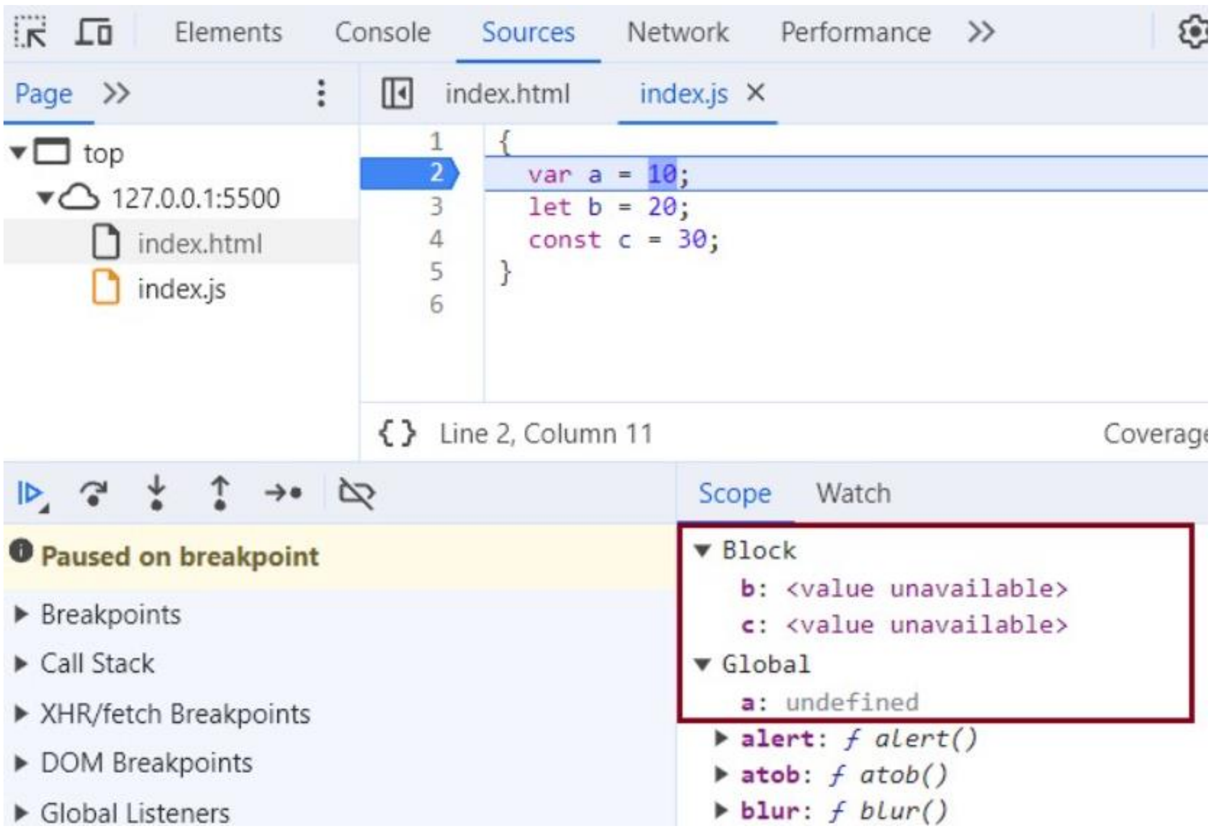
* We need to group all these statements together so that we can use multiple statements in a place where JS expects one statement.
for example - JS expects one statement after if(condition).

case 1

```
if(true) console.log("JS expecting single statement after if condition")
```

case 2

```
if(true) {
   var x = 20
   console.log("JS expecting multiple statements as a single compound statement after if
condition")
}
```



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

```
Block
  b: <value unavailable>
  c: <value unavailable>
Global
  a: undefined
▶ alert: ƒ alert()
▶ atob: ƒ atob()
▶ blur: ƒ blur()
```

```
{
  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

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

* Reason?
    * Inside **block** scope, let and const are hoisted and set their value to <value not available> before they are initialised. Once they are initialised we can access them within the block.

    * While, a is stored inside a GLOBAL scope.

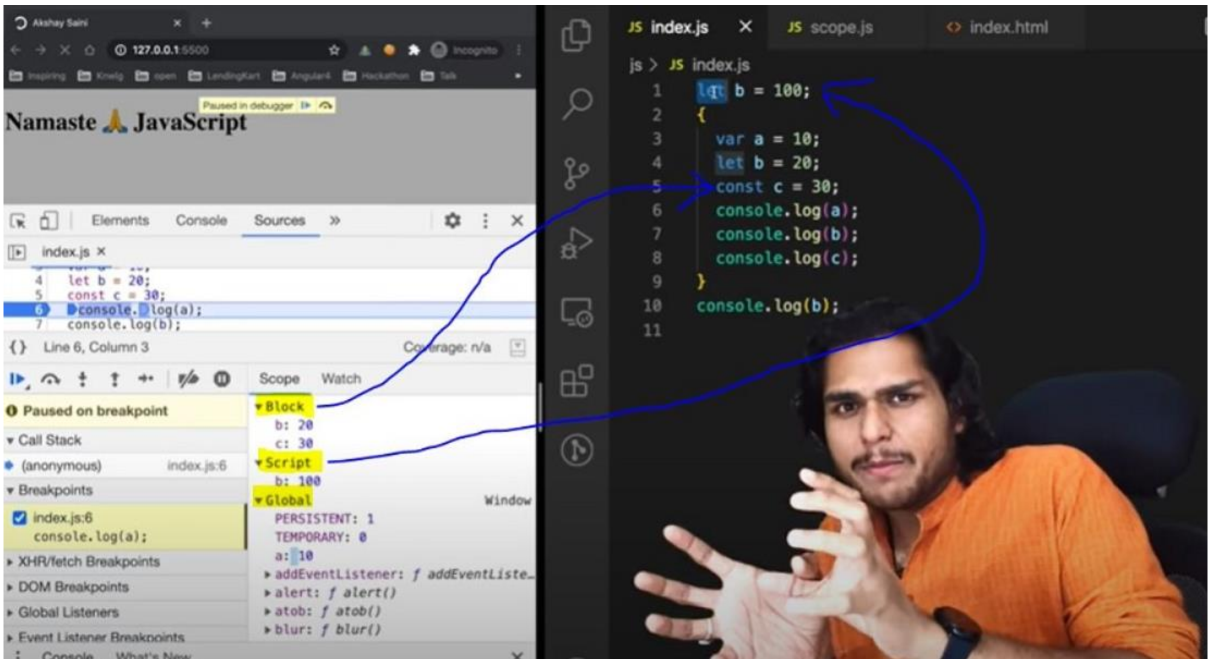
* Thus, we say, **let** and **const** are BLOCK SCOPED. They are stored in a separate memory space which is reserved for this block. Also, they can't be accessed outside this block. But var a can be accessed anywhere as it is in global scope. Thus, we can't access them outside the Block.

What is **Shadowing**?

*
```
  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 value 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 its 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.
```



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

```
Paused on breakpoint
Scope
▾Block
   b: 20
   c: 30
▾Script
   b: 100
▾Global                                   Window
   PERSISTENT: 1
   TEMPORARY: 0
   a: 10
 ▸addEventListener: ƒ addEventListe…
 ▸alert: ƒ alert()
 ▸atob: ƒ atob()
 ▸blur: ƒ blur()
```

* Same rule applies to **functions**

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

What is **Illegal Shadowing**?

```
let a = 20; // a value lies in script scope
{
   var a = 30; // a value lies in global scope
}
console.log(a) // which a is accessed by JS. This is ambiguity for JS to understand. We get an error that a is already declared. You cannot shadow a like this. This is called illegal Shadowing.

// 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.

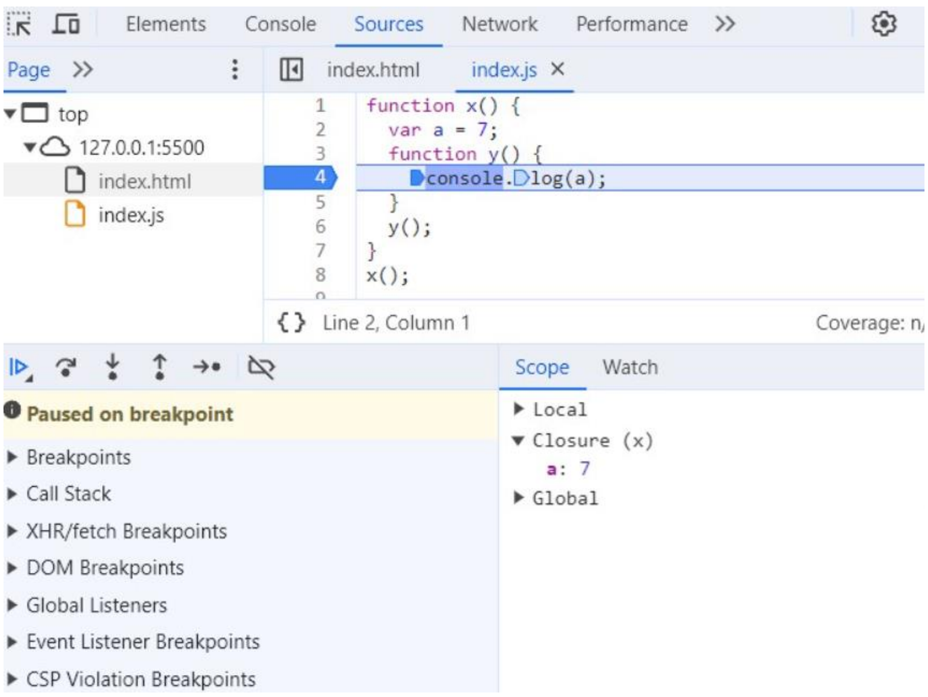
* Since var is function scoped, it is not a problem with the code below.

```
let a = 20;
function x() {
   var a = 20; // function scoped. Valid Shadowing
}

var a = 20;
{
   let a = 30; // block scoped. Valid Shadowing
   console.log(a) // block scope a is accessed
}
// when JS debugger checks below line, at that point block scope a variable is gone. It is garbage collected.
console.log(a) // global scope.  a is accessible
```

Note - Each block has their own lexical scope and they follow scope chain pattern.

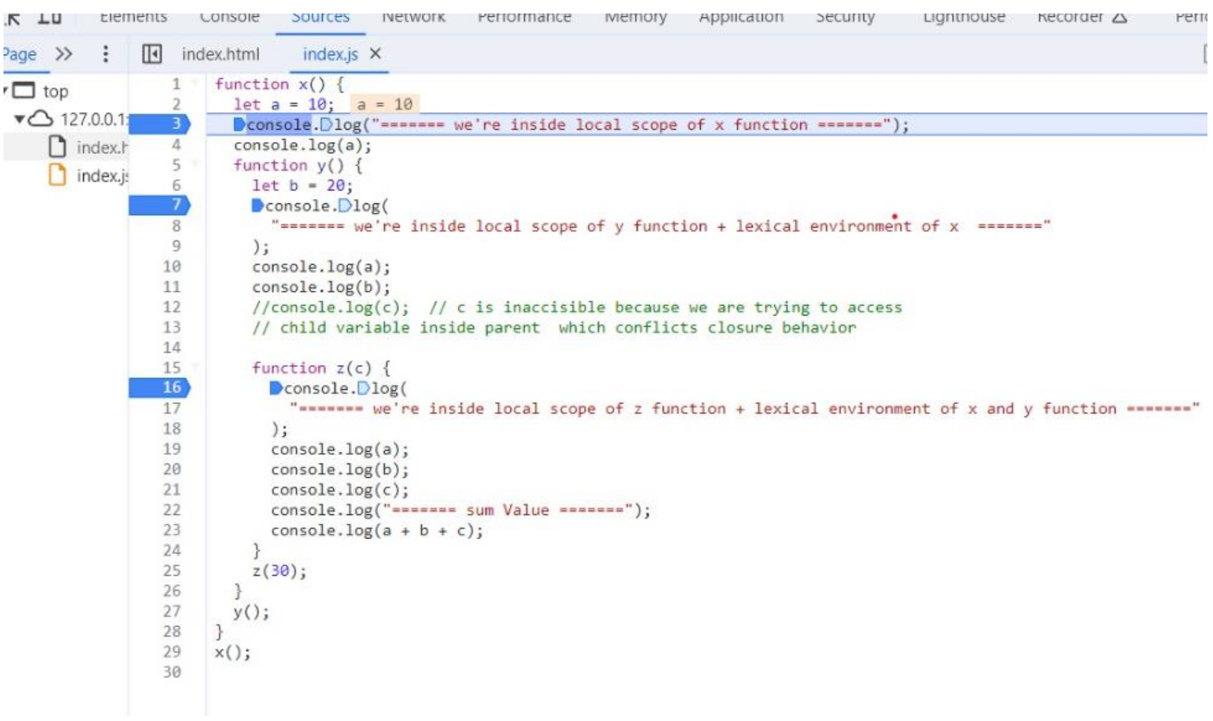
# Chapter 10: Closures in JS

* Function bundled along with its lexical scope is **closure**.

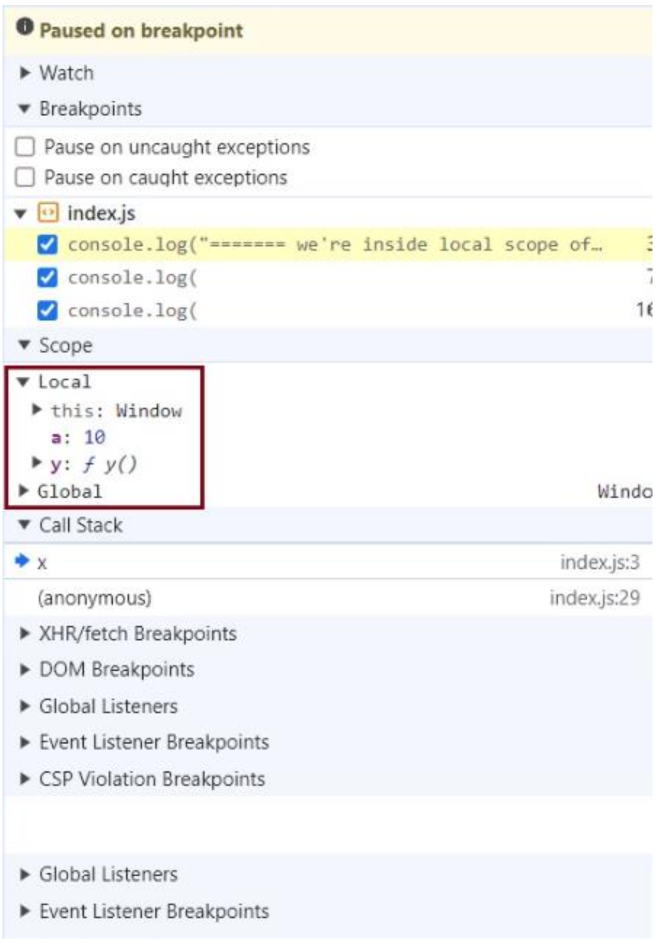


```
function x() {
  var a = 7;
  function y() {
    console.log(a);
  }
  y();
}
x();
```

In the above Image, function y along with its lexical environment i.e. function x's environment forms a closure.

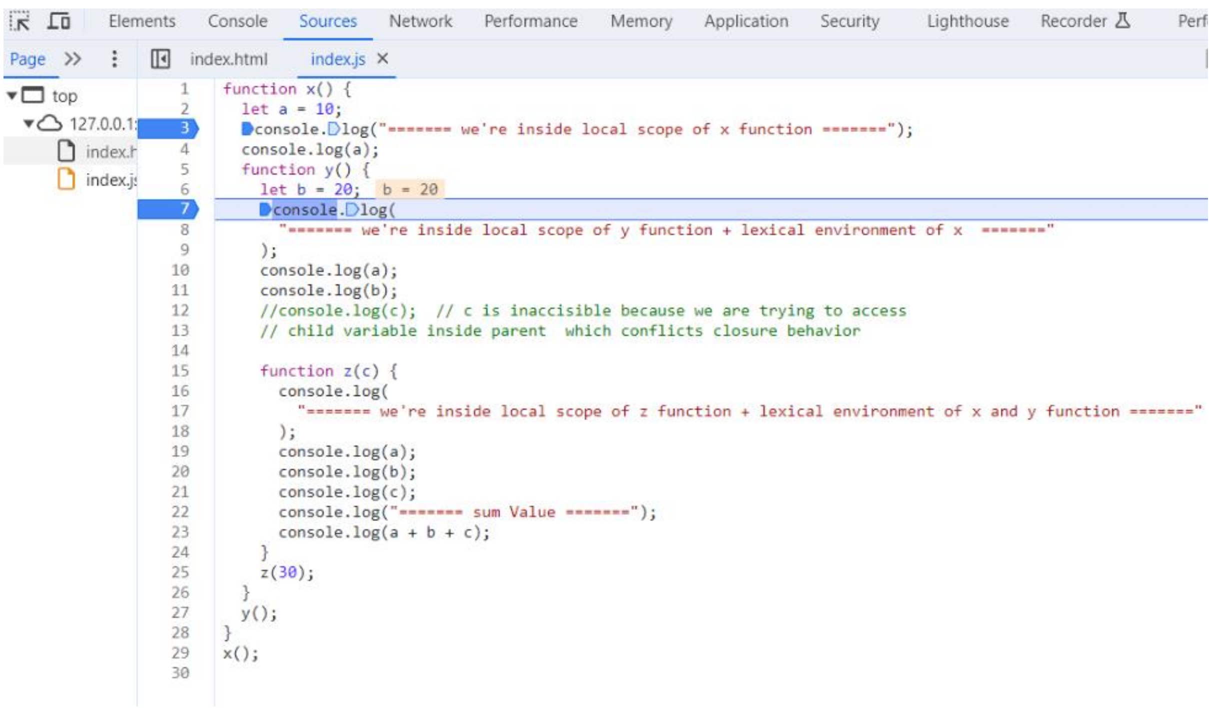


```
function x() {
  let a = 10;  a = 10
  console.log("======= we're inside local scope of x function =======");
  console.log(a);
  function y() {
    let b = 20;
    console.log(
      "======= we're inside local scope of y function + lexical environment of x  ======="
    );
    console.log(a);
    console.log(b);
    //console.log(c);  // c is inaccisible because we are trying to access
    // child variable inside parent  which conflicts closure behavior

    function z(c) {
      console.log(
        "======= we're inside local scope of z function + lexical environment of x and y function ======="
      );
      console.log(a);
      console.log(b);
      console.log(c);
      console.log("======= sum Value =======");
      console.log(a + b + c);
    }
    z(30);
  }
  y();
}
x();
```

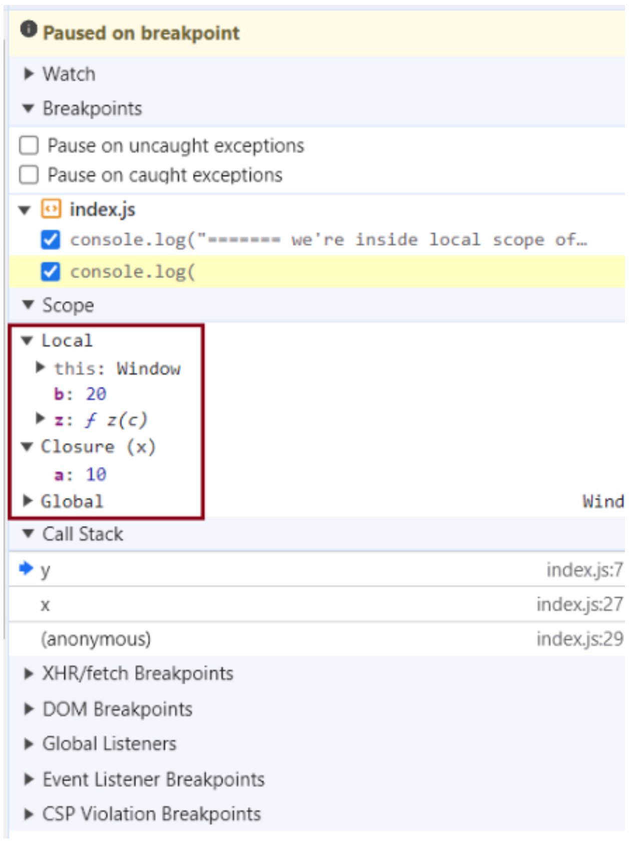


```
Paused on breakpoint
▶ Watch
▼ Breakpoints
☐ Pause on uncaught exceptions
☐ Pause on caught exceptions
▼ index.js
☑ console.log("======= we're inside local scope of…   3
☑ console.log(                                          7
☑ console.log(                                          16
▼ Scope
▼ Local
  ▶ this: Window
    a: 10
  ▶ y: ƒ y()
▶ Global                                                Windo
▼ Call Stack
➜ x                                              index.js:3
  (anonymous)                                    index.js:29
▶ XHR/fetch Breakpoints
▶ DOM Breakpoints
▶ Global Listeners
▶ Event Listener Breakpoints
▶ CSP Violation Breakpoints

▶ Global Listeners
▶ Event Listener Breakpoints
```

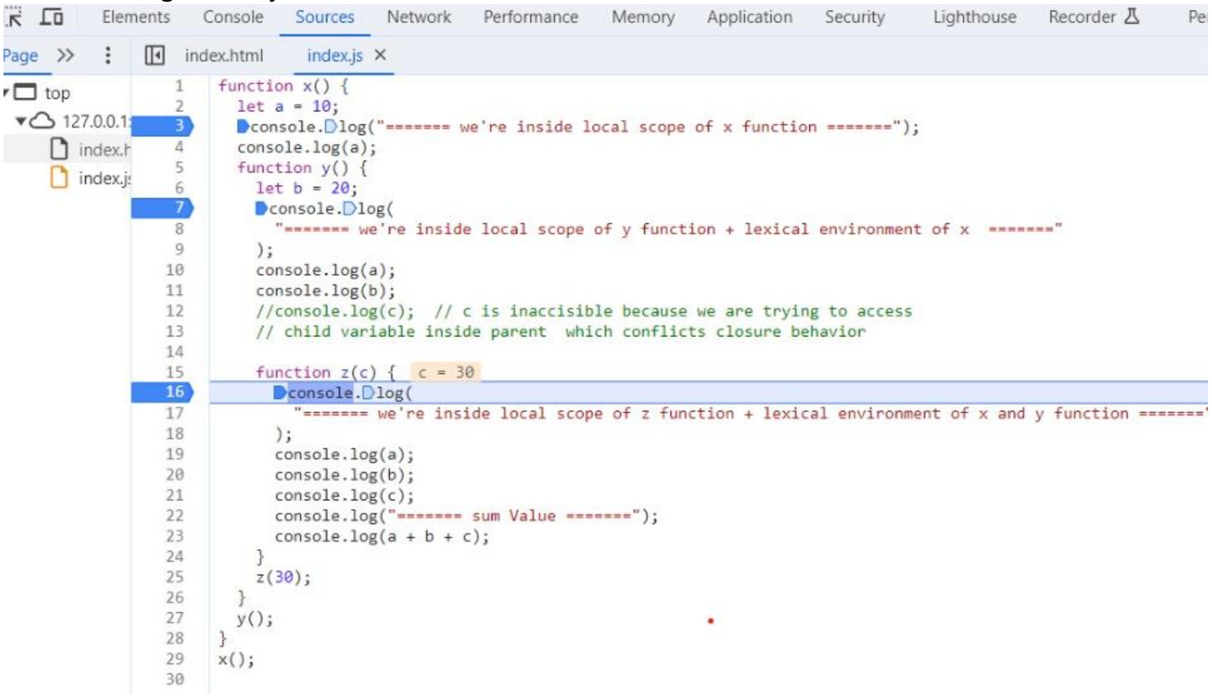
when we are inside line number 3 we are inside local scope of function x . we have access to x's local variables and global variables.



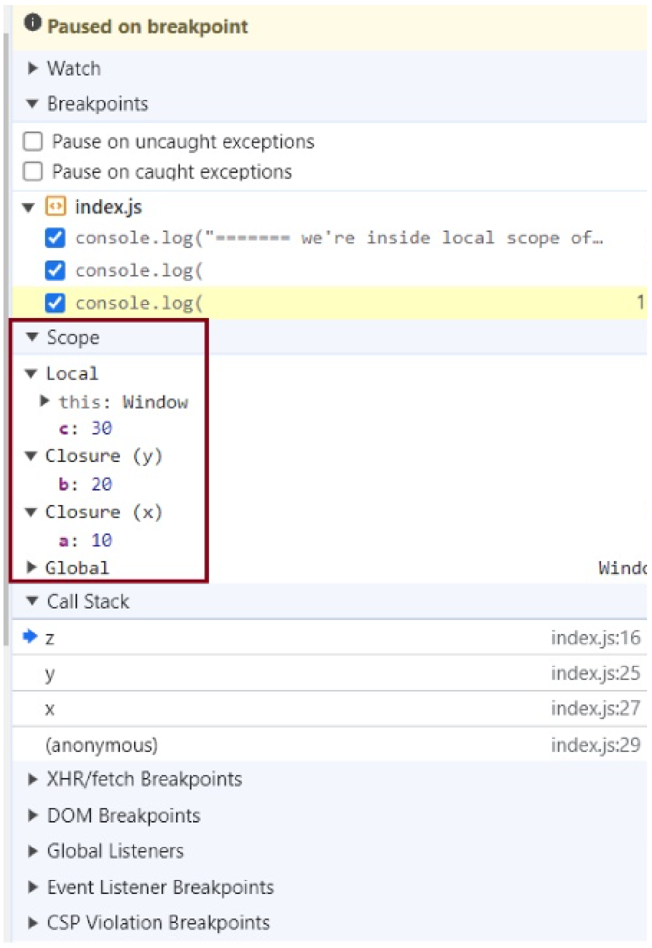
```
1  function x() {
2    let a = 10;
3    console.log("======= we're inside local scope of x function =======");
4    console.log(a);
5    function y() {
6      let b = 20;  b = 20
7      console.log(
8        "======= we're inside local scope of y function + lexical environment of x  ======="
9      );
10     console.log(a);
11     console.log(b);
12     //console.log(c);  // c is inaccisible because we are trying to access
13     // child variable inside parent  which conflicts closure behavior
14
15     function z(c) {
16       console.log(
17         "======= we're inside local scope of z function + lexical environment of x and y function ======="
18       );
19       console.log(a);
20       console.log(b);
21       console.log(c);
22       console.log("======= sum Value =======");
23       console.log(a + b + c);
24     }
25     z(30);
26   }
27   y();
28 }
29 x();
30
```


```
Paused on breakpoint
▶ Watch
▼ Breakpoints
☐ Pause on uncaught exceptions
☐ Pause on caught exceptions
▼ index.js
  ☑ console.log("======= we're inside local scope of…
  ☑ console.log(
▼ Scope
▼ Local
  ▶ this: Window
    b: 20
  ▶ z: ƒ z(c)
▼ Closure (x)
    a: 10
▶ Global                                         Wind
▼ Call Stack
➡ y                                          index.js:7
  x                                          index.js:27
  (anonymous)                                index.js:29
▶ XHR/fetch Breakpoints
▶ DOM Breakpoints
▶ Global Listeners
▶ Event Listener Breakpoints
▶ CSP Violation Breakpoints
```

when we are inside line number 7 we are inside local scope of function y. We have access to y's local variables and It forms a closure with x function meaning we can also access the variables of x function and global object.



```
1   function x() {
2     let a = 10;
3     console.log("======= we're inside local scope of x function =======");
4     console.log(a);
5     function y() {
6       let b = 20;
7       console.log(
8         "======= we're inside local scope of y function + lexical environment of x  ======="
9       );
10      console.log(a);
11      console.log(b);
12      //console.log(c);  // c is inaccisible because we are trying to access
13      // child variable inside parent  which conflicts closure behavior
14
15      function z(c) {  c = 30
16        console.log(
17          "======= we're inside local scope of z function + lexical environment of x and y function =======
18        );
19        console.log(a);
20        console.log(b);
21        console.log(c);
22        console.log("======= sum Value =======");
23        console.log(a + b + c);
24      }
25      z(30);
26    }
27    y();
28  }
29  x();
30
```


```
Paused on breakpoint
▶ Watch
▼ Breakpoints
☐ Pause on uncaught exceptions
☐ Pause on caught exceptions
▼ index.js
  ☑ console.log("======= we're inside local scope of…
  ☑ console.log(
  ☑ console.log(                                  1
▼ Scope
▼ Local
  ▶ this: Window
    c: 30
▼ Closure (y)
    b: 20
▼ Closure (x)
    a: 10
▶ Global                                      Windc
▼ Call Stack
➡ z                                   index.js:16
  y                                   index.js:25
  x                                   index.js:27
  (anonymous)                         index.js:29
▶ XHR/fetch Breakpoints
▶ DOM Breakpoints
▶ Global Listeners
▶ Event Listener Breakpoints
▶ CSP Violation Breakpoints
```

when we are inside line number 16 we are inside local scope of function z. we have access to z's local variables and It forms a closure with x function and y function, meaning we can also access the variables of x function and y function and global object.

* 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 and so on up to the global scope.

```
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 which is function y() {console.log(a); }

z() // It will print 7 on the console , but where 'a = 7' is coming from .At this point It's gone from
the local scope ,It's not even there in global scope. It will come from the closure. (closure is also a
temporary memory space with some values in it which later garbage collected when unused)
```

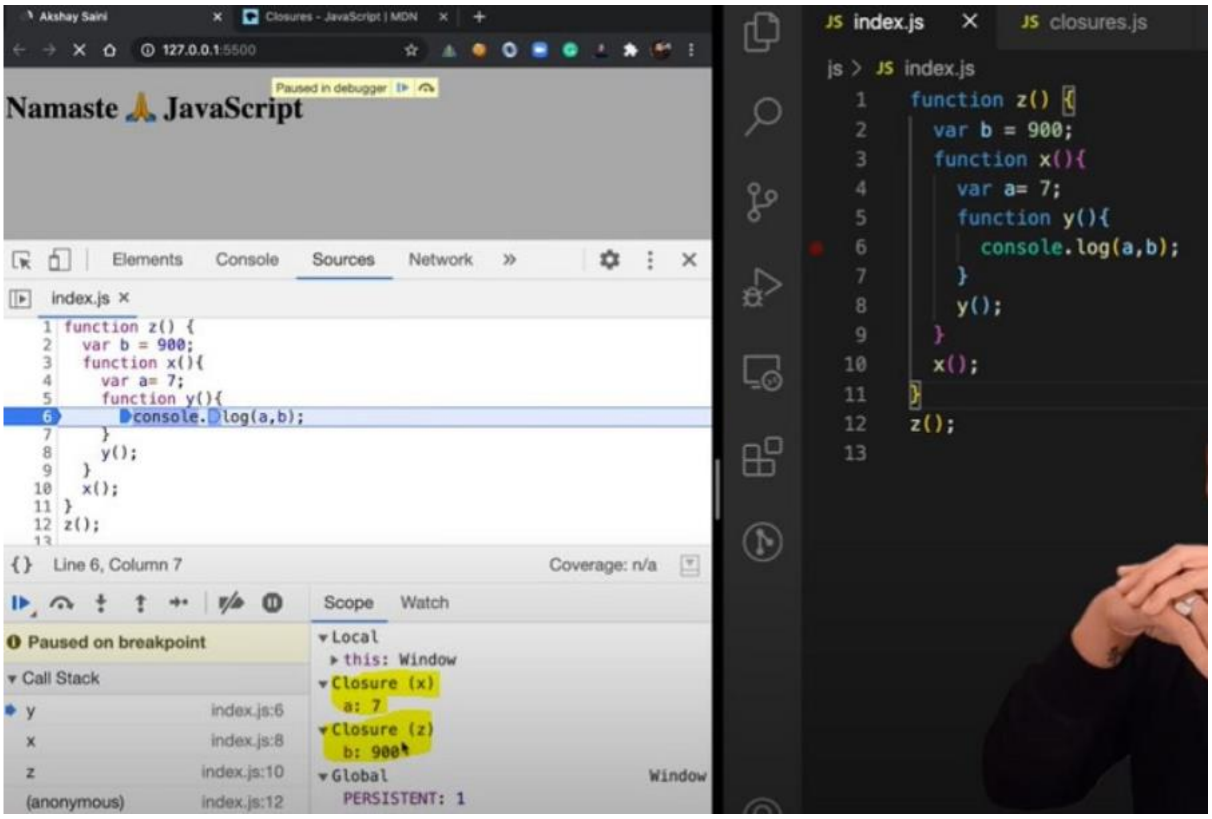
* In above code, outer function x returns the inner function y and once after that it gets removed from the call stack. Behind the scene not only the y function is returned but also the entire closure (fun y + its lexical scope) is returned and put inside z. So when z is used somewhere else in program, it still remembers var a inside x().

```
function x() {
   var a = 7;
   function y() {
      console.log(a);
   }
   var a = 100;
   return y;
}
var z = x(); // returns y with its lexical scope. In its lexical scope the value of a is not returned the reference of a is returned.
// At first a was 7 then it modified to 100 in its execution context.
z() // 100
```

* Another Example

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

* 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.


Namaste JavaScript

Paused in debugger

```
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

```
Paused on breakpoint
Call Stack
  y              index.js:6
  x              index.js:8
  z              index.js:10
  (anonymous)    index.js:12
```

```
Scope
▾Local
  ▸this: Window
▾Closure (x)
   a: 7
▾Closure (z)
   b: 900
▾Global                     Window
   PERSISTENT: 1
```

* 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

# Chapter 11: setTimeout + Closures Interview Question

> **Time, tide and JavaScript wait for none.**

*
```
  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 and it remembers reference to i. So, wherever the function goes it carries this ref along with it.
  * setTimeout takes this Callback function & attaches a timer of 3000ms and stores it. Now the JS goes to next line without waiting and prints the string.
  * After 3000ms runs out, JS takes the function, put it into call stack and runs it.

*** Q: Print 1 after 1 sec, 2 after 2 sec till 5: Tricky interview question?**

```
  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
```

why it is behaving like this?

* This happens because of closures. When setTimeout stores the function somewhere and attaches a timer to it, the function remembers its reference to i, **not the value of i**. So, all 5 copies of function point to the same reference of i.

* setTimeout is a JS WEB API which stores these 5 functions in a separate memory space and pushes them into the Message queue one by one once the timer is expired. In the meantime, JS executed the next line and prints string 'Namaste JavaScript'.

* By the time call stack is empty the value of i is changed to 6. At this moment Browser's Event loop mechanism kicks in which pushes these message queue functions one by one to the call stack and inside the CallStack these functions get executed one after another and we see the outputs like these.

* 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). Every time setTimeout is run, the inside/Callback function forms closure with new variable i.

Implementation using var?

```
function x() {
  for(var i = 1; i<=5; i++){
  function close(i) {
    // I've put the setT function inside a new function close () to form a closure with respect to close (). why? Because every time setT() is called it creates a local copy of i at that moment. This is because of closure's added advantage.
    setTimeout(function() {
    console.log(i);
    }, i*1000);

  }
  close(i); // Every time we call close(i) it creates new copy of i.
  }
  console.log("Namaste JavaScript");
}
x();
```

# Chapter 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.

Example –

```
function outer() {
   var a = 10; // local scope to function
   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 second `()` to call inner
function
```

### 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; // local scope to function
   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 str.

### Q5: In below code, will inner function 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 explanation?

```
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 found a inside outer function then it would have gone more outer to find a and thus would have printed 100. So, it tries 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 outside of the counter function scope. This way we achieve data hiding. We are hiding count data to the outside world.
```

```
//(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 won't impact the output of counter1
```

****

```
// 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();  // creating counter1 object to get an access to the constructor fun.
counter1.IncrementCounter();
counter1.IncrementCounter();
counter1.DecrementCounter();
// returns 1 2 1
```

### Q9: Disadvantage of closure?

Ans: Overconsumption of memory when using closure as closures consume some memory space to accumulate data inside of it. These data are not garbage collected until the program expires. So, when creating many closures, more memory is accumulated and this can create memory leaks if not handled properly.

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 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 closure over var x. So, memory of x cannot be freed. Like this if more closures formed, it becomes an issue. To tackle this, JS engines like v8 and Chrome have smart garbage collection mechanisms. Say we have var x = 0, z = 10 in above code. When console log happens, x is printed as 0 but z is removed automatically because z is unused.

// JavaScript engine calls garbage collector (GC) to free unused memory when they are no longer be used. GC internally uses mark and sweep algorithm.

// IIFE - Search Online – Explore more.

# Chapter 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. This is a normal way of creating a JS function.

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

### Q: What is Function Expression?

Assigning a function to a variable is Function Expression. Function acts like a value here.

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

### 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 is created like a variable (b:undefined) and until code reaches the function()  part, it is still undefined. So, it cannot be called.
```

### 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 e.g. 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-Higher Order Function). These abilities are altogether known as First class function. This programming concept is available in some other languages too.

```
var b = function(param1) {
  console.log(param1); // prints " f() {} "
}
b(function(){});// passing anonymous function as value as argument
```

```
// Other way of doing the same thing:
var b = function(param1) {
  console.log(param1);
}
function xyz(){
}
b(xyz); // same thing as previous 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.
```

# Chapter 14: Callback Functions in JS ft. Event Listeners

### Callback Functions

* Functions are first class citizens i.e. 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 call backs, we can do async things in JS.

```
 setTimeout(function () {
   console.log("timer");
  }, 30000);

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

Outputs

```
x
y
timer
```

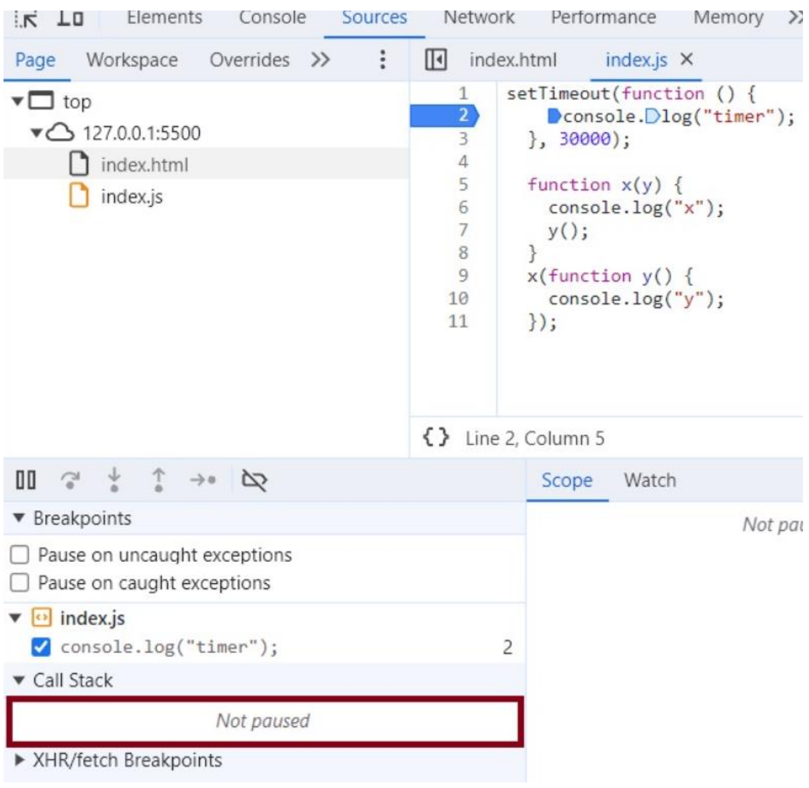
Execution Order

* setTimeout takes the Callback function in WEB API ENV and attaches a Timer to it.

* In CallStack function x is executed first since it is called first.

* Then y is executed since its being called inside x function.

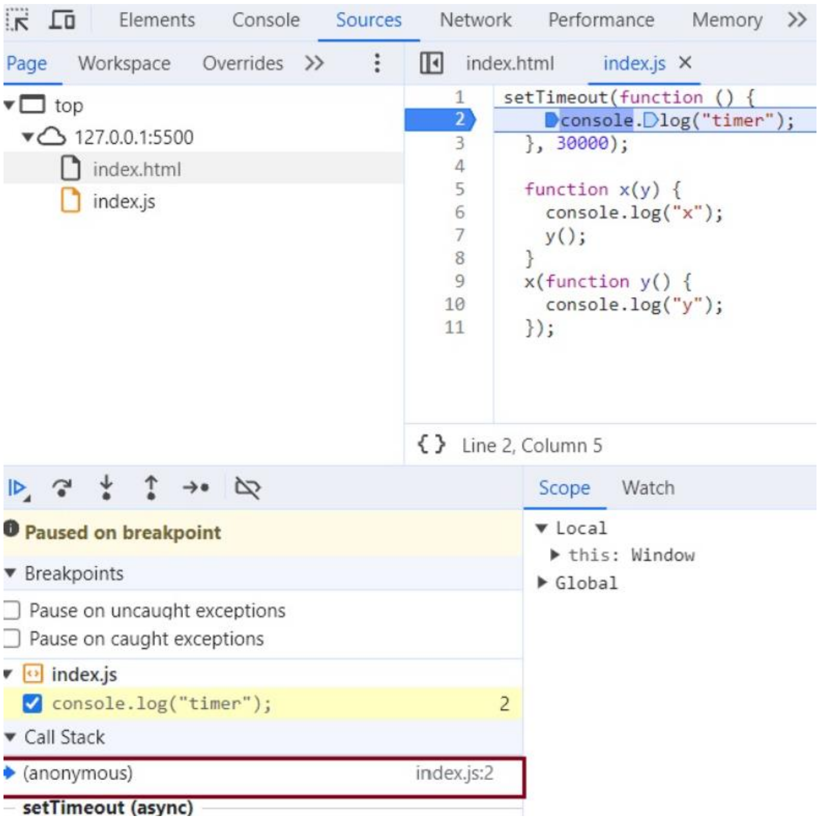
Call stack before 30 sec – Before Set Timeout is invoked.



```
setTimeout(function () {
  console.log("timer");
}, 30000);

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

* In the call stack, first x and y are present. After code execution, they go away and stack is empty. Then after 30 seconds anonymous suddenly appear up in the stack i.e. setTimeout

CallStack after 30 sec



```
setTimeout(function () {
  console.log("timer");
}, 30000);

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

* In summary all these 3 functions are executed through call stack. If any operation blocks the call stack, it’s called blocking the main thread. For example if x() takes 30 sec to run, then JS has to wait for it to finish because it has only 1 call stack,1 main thread. So never ever block the main thread.

* Always use **async** for functions that take time e.g. setTimeout internally uses async.

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.

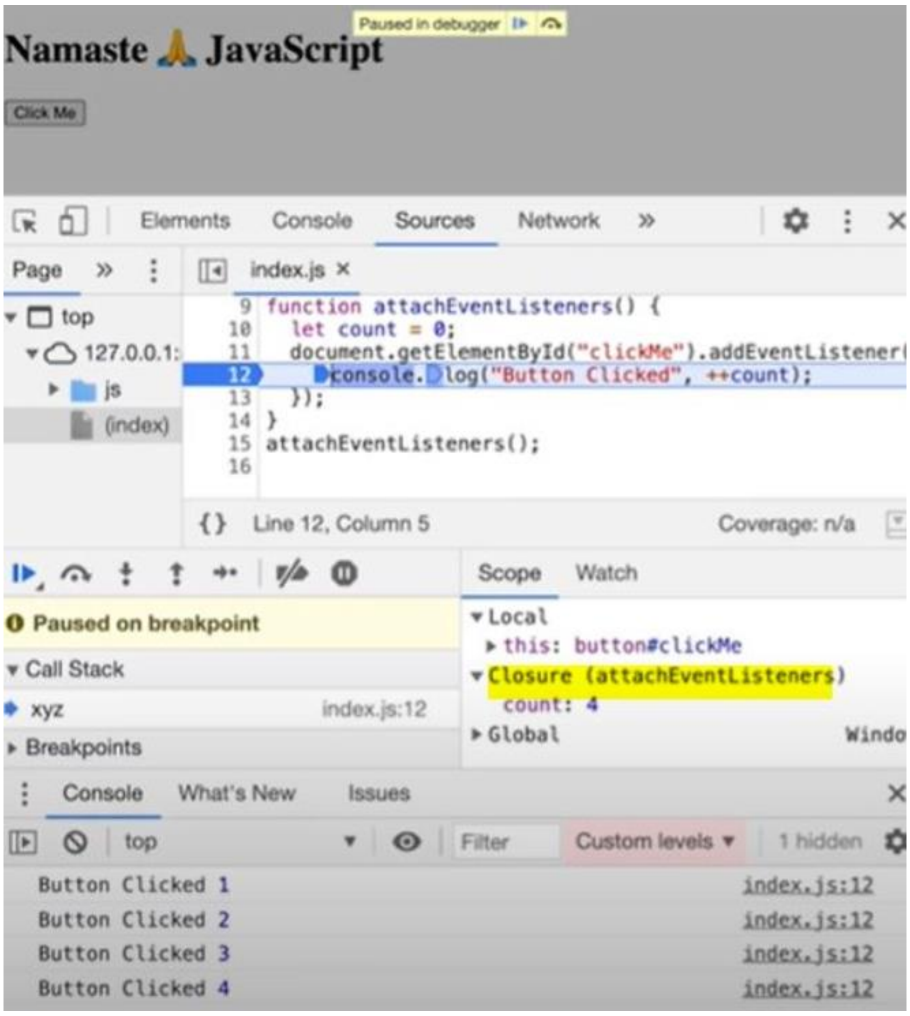
```
// index.html
  <button id="clickMe">Click Me!</button>

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

* Let’s implement a 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 with Event Listener 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.
  });
}
attachEventList();
```



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

Line 12, Column 5   Coverage: n/a

Paused on breakpoint

Call Stack
xyz   index.js:12
Breakpoints

```
▾Local
  ▸this: button#clickMe
▾Closure (attachEventListeners)
    count: 4
▸Global                          Windo
```

```
Button Clicked 1                 index.js:12
Button Clicked 2                 index.js:12
Button Clicked 3                 index.js:12
Button Clicked 4                 index.js:12
```

### Garbage Collection and RemoveEventListener

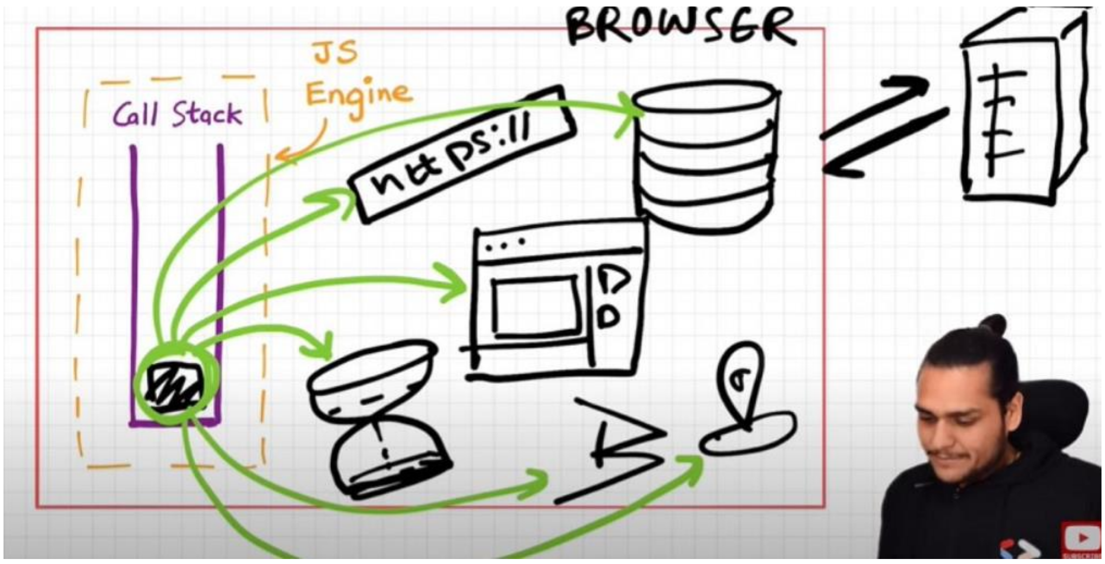
### Why do we remove Event Listeners?

* Event listeners are heavy as they form closures. So even when call stack is empty, Event Listener won't free up memory allocated to count as it doesn't know when it may need count again or when, user may click the button again. So, we remove event listeners when we don't need them. ALL these event listeners onClick, onHover, onScroll consumes a lot of memory because of closures. Once we remove them with the help of RemoveEventListener they will be garbage collected.

To do - Dev tool explanation - show Event Listener closure practically - Refer this session Video again.

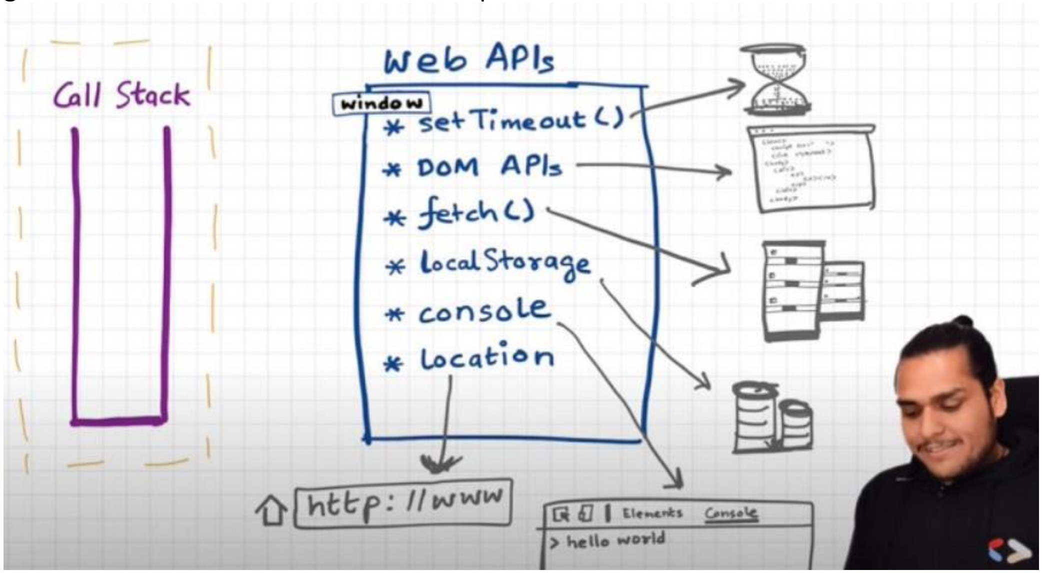
# Chapter 15: Asynchronous JavaScript & EVENT LOOP from scratch

* Note: Call stack will execute any execution context which enters it. Time, tide and JS waits for none. TLDR; 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 a way out to connect the CallStack with all these superpowers. This is done using Web APIs.



### WebAPIs

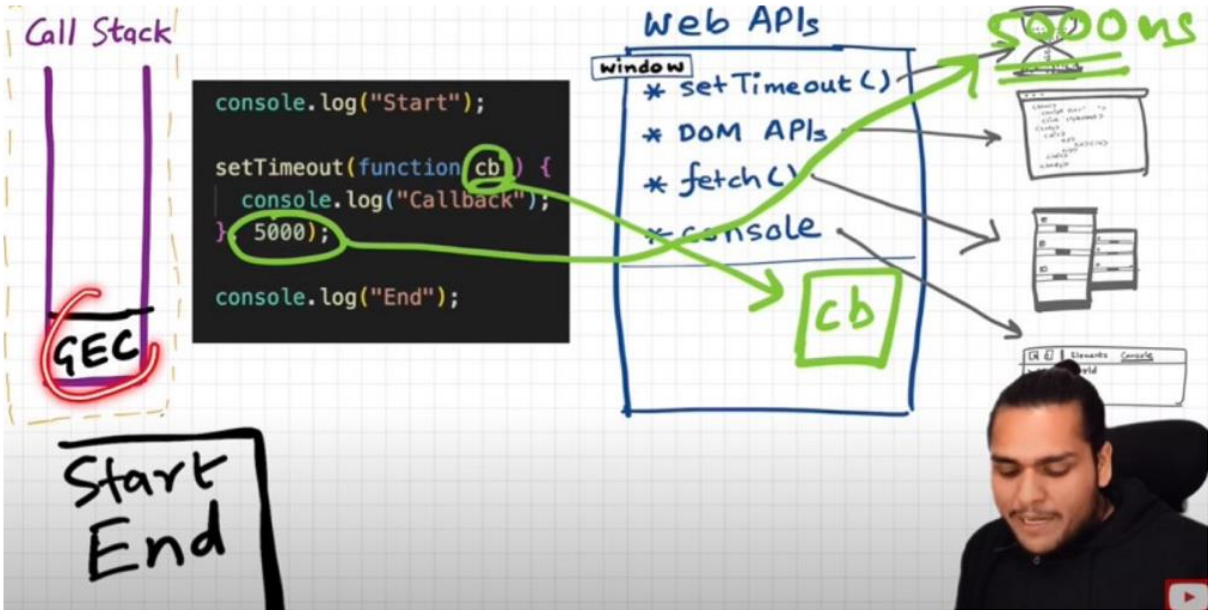
None of the below are part of JavaScript! These are extra superpowers that browser has. Browser gives access to JS CallStack to use these powers.



* setTimeout(), DOM APIs, fetch(), local storage, console (yes, even console.log is not JS!!), location and so many more.
  * setTimeout() : Timer function.
  * DOM APIs : e.g.Document.xxxx ; Used to access HTML DOM tree. (Document Object Manipulation)
  * fetch() : Used to make connection with external servers e.g. Netflix servers etc.

* We get all these inside call stack through global object i.e. 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 explicitly write window but it is implied.

* Let's underhand the below code image and its explanation



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

setTimeout(function cb() {
  console.log("Callback");
}, 5000);

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

```
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 logs values in console.
  * setTimeout(function cb() {
    console.log("timer");
  }, 5000); // This calls the setTimeout web API which gives access to timer feature. It stores the callback cb() and starts timer.

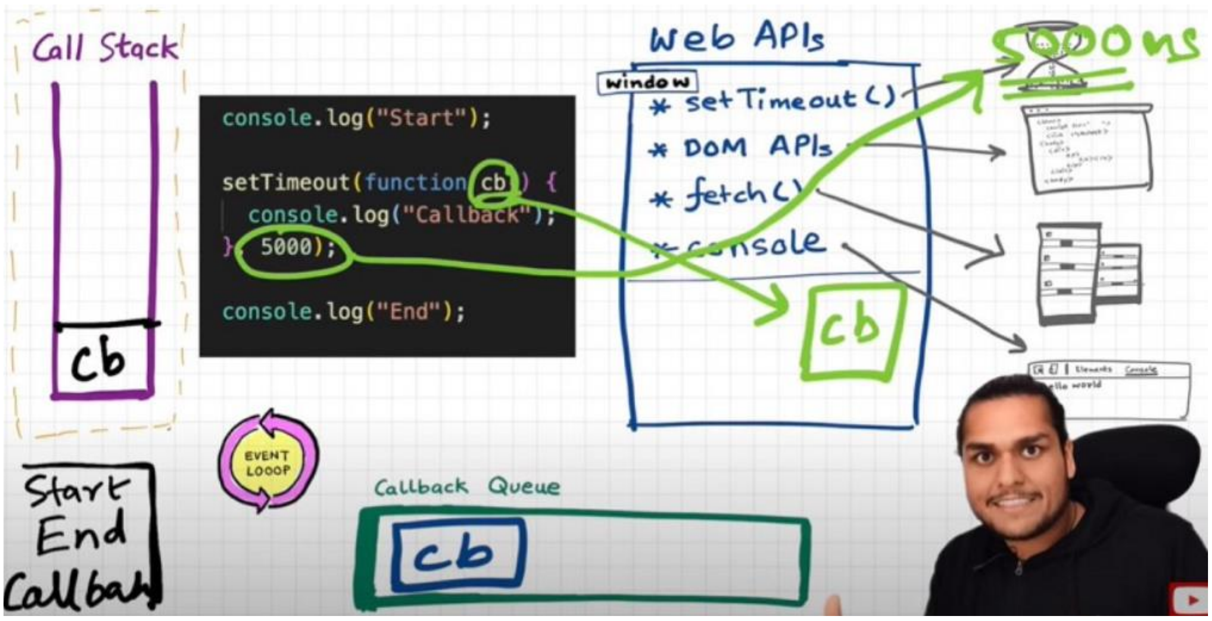
  * console.log("End"); // calls console API and logs "End" in console window. After this GEC pops from call stack.

  * 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 second of time, 'timer' is logged in the console?

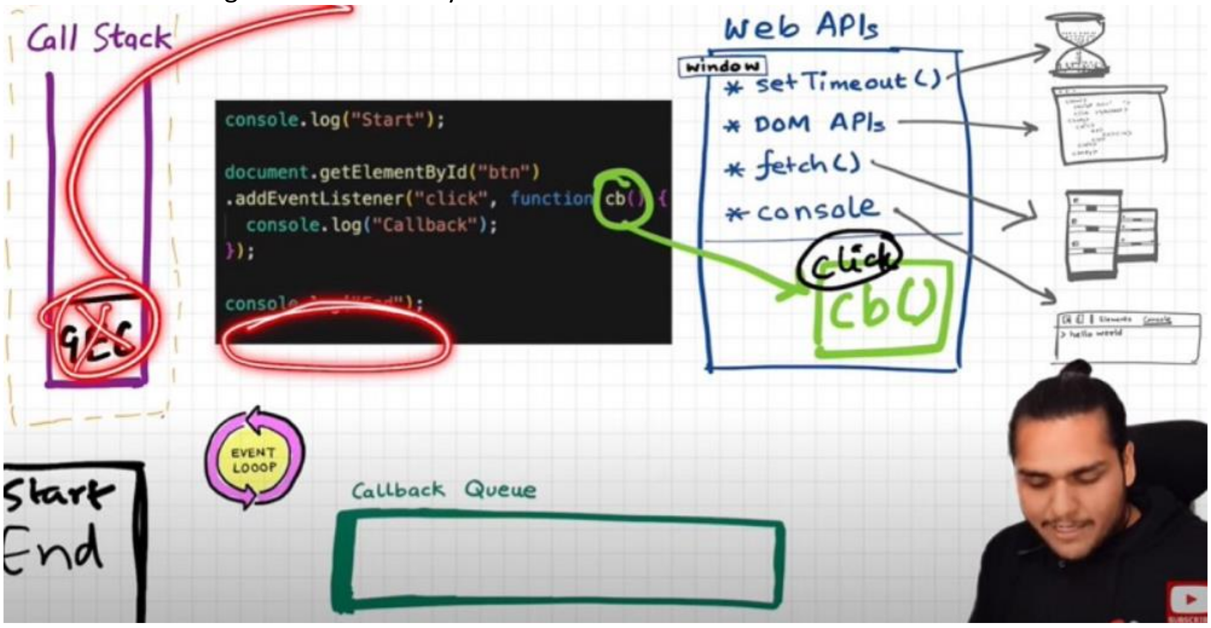
* cb() cannot simply directly go to CallStack to be executed. It goes through the Callback queue when timer expires.
* Event loop keep checking the Callback queue, and see if it has any element to puts it into call stack. It is like a gate keeper.
* Once cb() is in Callback queue, event loop pushes it to CallStack to run. Then the call stack connects with Console API to log 'timer' into the console window.



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

setTimeout(function cb() {
  console.log("Callback");
}, 5000);

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

Q: Another example to understand Event loop & Callback Queue.

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



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

document.getElementById("btn")
.addEventListener("click", function cb() {
  console.log("Callback");
});

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

Explanation?

```
 console.log("Start");
 document. getElementById("btn").addEventListener("click", function cb() {
   // cb() registered inside webapi environment and event(click) gets attached to it. i.e. REGISTERING
CALLBACK AND ATTACHING EVENT TO IT.
   console.log("Callback");
 });
 console.log("End"); // calls console API and logs in console window.

 // After this GEC get removed from call stack.

 // In above code, even after console prints "Start" and "End" and pops GEC out, the event Listener
stays in web API env (with hope that user may click it someday) until explicitly removed, or the
browser is closed.
```

* Event loop has just one job i.e. it keeps monitoring Callback queue and call stack. It acts like a gate keeper. If it finds some code blocks or statements in the Callback queue, it pushes them one by one into the call stack and deletes them from the Callback queue. This only happens when the call stack is empty.

Q: Need of Callback/message queue?

Ans: Suppose user clicks button x6 times. So 6 cb() are put inside Callback queue. At this moment Event loops checks whether the call stack is empty or not. If call stack is empty It pushes these call backs one by one from the Callback queue into the call stack till every Callback are executed inside the CallStack.

### Behaviour of fetch (Microtask Queue?)

Let's observe the code below and try to understand

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

setTimeout(function cbT() {
  console.log("CB Timeout");
}, 5000);

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

// millions of lines of code

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

Code Explanation

* In first Line, JS engine communicates with console Web API and logs 'Start' in the console window.

* In next line setTimeout takes the call back function cbT and stores it in Web API ENV and attaches a Timer of 5 seconds to it.

* Moving forward JS encounters fetch method given by browser which makes an API call, but before Making the API call, fetch takes the Callback function cbF and stores it in WEB API env. After that it makes a request to the server.

* At this moment both CbF and cbT functions are stored in WEB API Env.

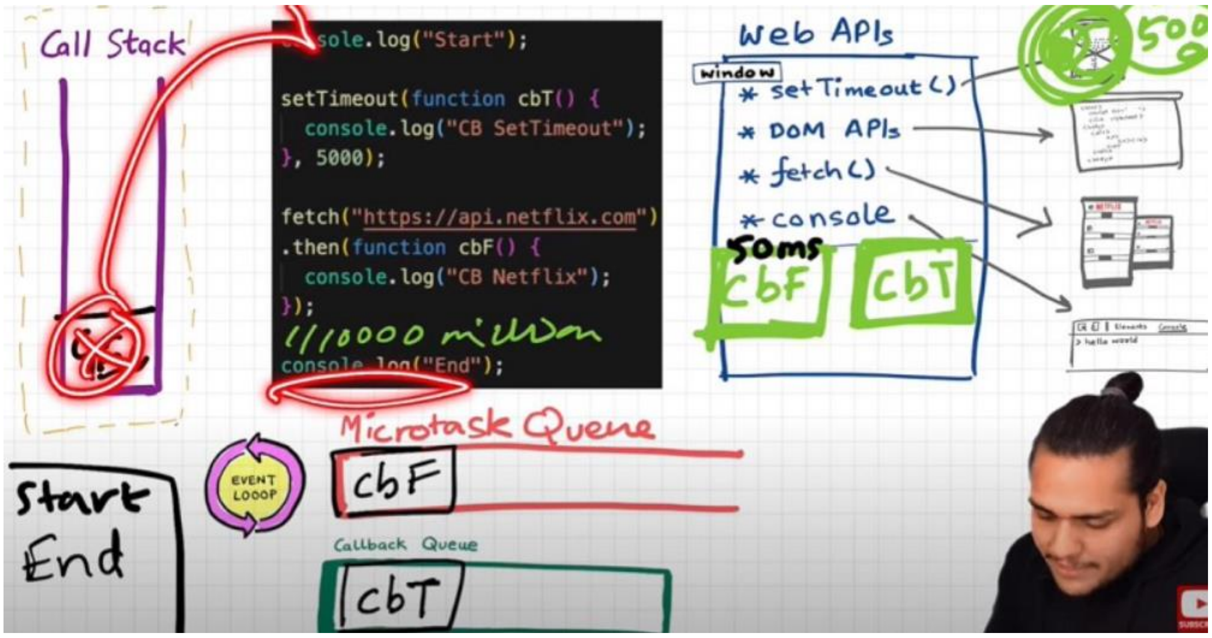
* Meanwhile JS engine keeps executing the next million lines of code in the call stack. When 5 sec Timer attached to cbT expires, cbT is pushed into the Callback queue. Also, when data is returned from the server for the fetch API, cbF is pushed into microtask queue.

* Microtask Queue is exactly same as Callback Queue, but it has higher priority. Functions in Microtask Queue are executed earlier than Callback Queue.

* When JS is done executing the last line of code which is console.log("End"), call task is empty. At this moment function inside microtask queue is pushed into the call stack and the function gets executed inside the CallStack. After the function is done with its execution, CallStack is empty again. This time function inside message queue is pushed into the call stack and that function is executed inside the CallStack.

* These pushing operation of functions from microtasks/message queue to call stack is performed by our Hero **Event Loop**

* See below Image for more understanding



```
sole.log("Start");

setTimeout(function cbT() {
  console.log("CB SetTimeout");
}, 5000);

fetch("https://api.netflix.com")
.then(function cbF() {
  console.log("CB Netflix");
});
//10000 million
console.log("End");
```

#### 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 is, 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. Macro 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 call backs 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 call backs stay in the web API environment forever, that's why it's advised 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.

# Chapter 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.

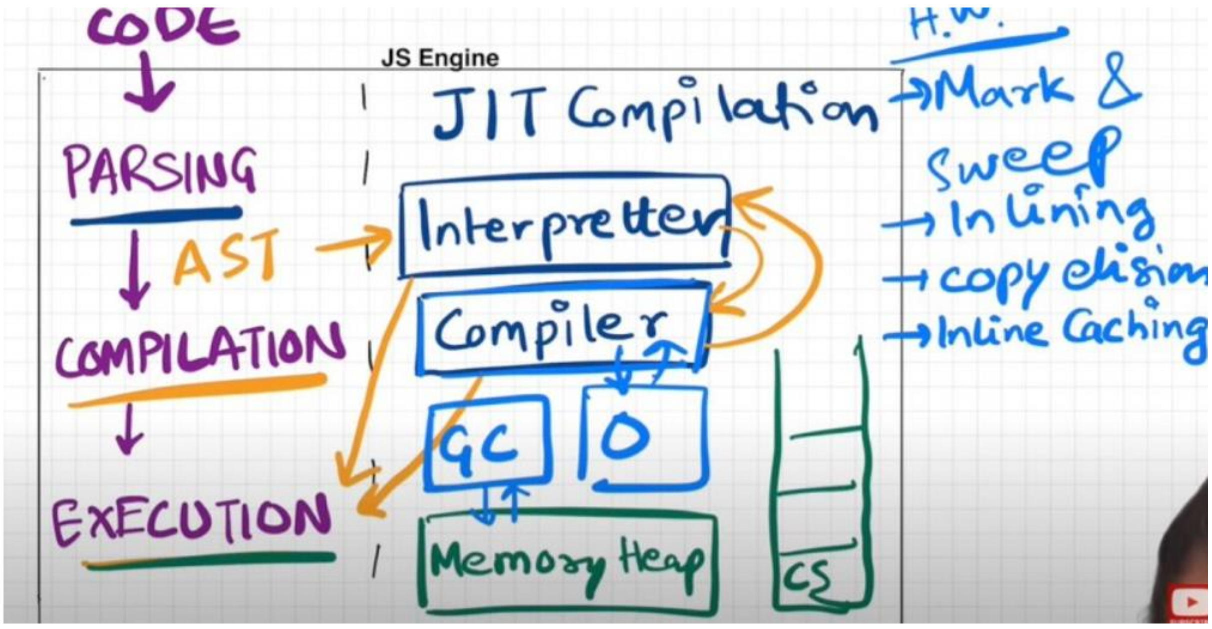
* ECMAScript is a governing body of JS. It has set of rules which are followed by all JS engines like Chakra (Edge), Spidermonkey (Firefox) (first JavaScript engine created by JS creator himself), v8(Chrome)

* JavaScript Engine is not a machine. Its software written in low level languages (e.g. C++) that takes in hi-level code in JS and spits out low level machine code.

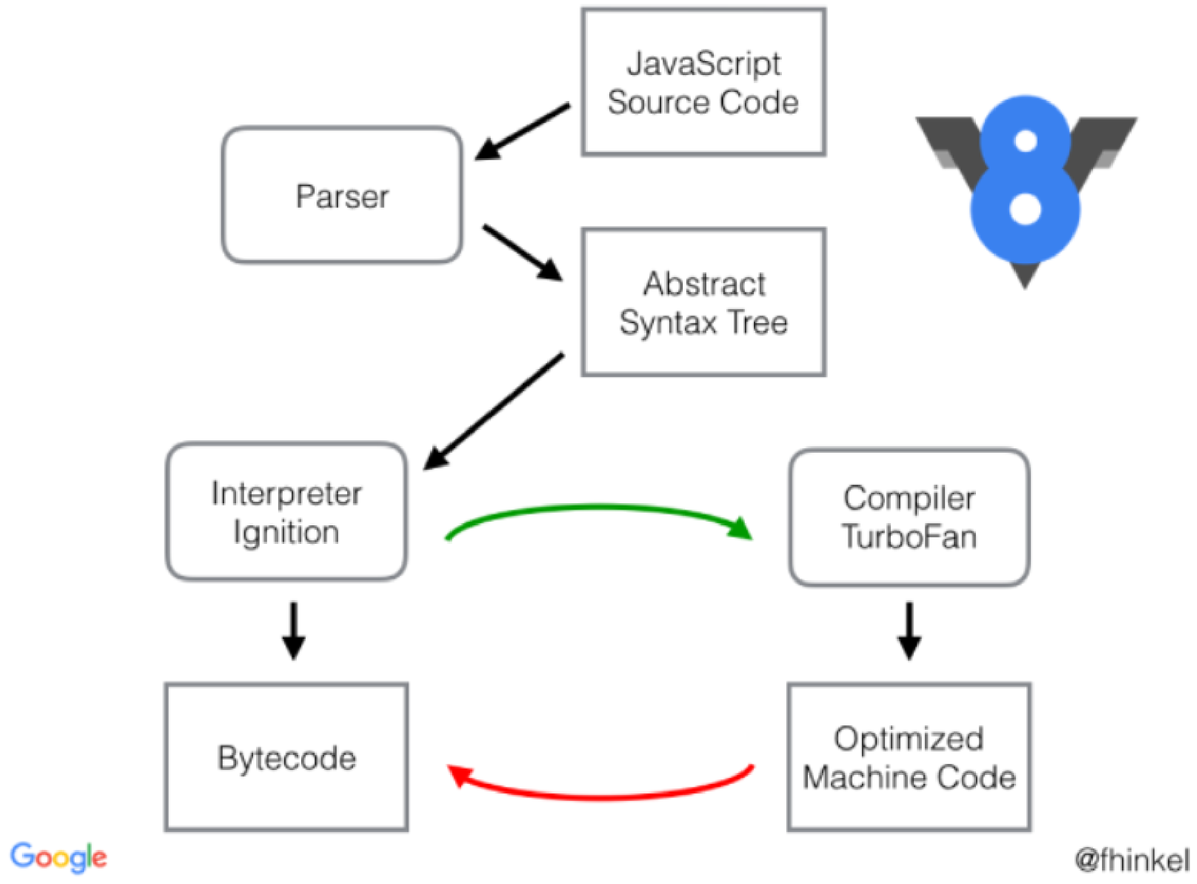
JS Engine Architecture

* 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. Kind of 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 high-level code to byte code and moves the byte codes to execution phase. While interpreting, compiler also works hand in hand to compile and form optimized code during runtime. Compiler talks to interpreter while the code is interpreted line by line. In each line compiler does the compilation **Does JavaScript really Compiles?** The answer is a loud **YES**. More info at: [Link 1](https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/get-started/ch1.md#whats-in-an-interpretation), [Link 2](https://web.stanford.edu/class/cs98si/slides/overview.html), [Link 3](https://blog.greenroots.info/javascript-interpreted-or-compiled-the-debate-is-over-ckb092cv302mtl6s17t14hq1j). JS used to be only interpreter in old times, but now has both to compile and interpreter code and this makes JS a JIT compiled language, it's like best of both worlds.
3. **Execution** - This phase has 2 components i.e. Memory heap (place where all memory is stored) and Call Stack (same call stack from previous chapters). There is also a garbage collector. It uses an algorithm called **Mark and Sweep**.



* 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:



Interpreter -
==============
Interpreter executes code line by line. It does not know what will happen in the next line.

pro: Fast
cons: Less performant

Compiler -
==========
compiler compiles the whole code even before executing. The compiled code is optimised version of original code and then the optimised one gets executed which improves the performance.

pros: Performant, high efficiency
cons: a bit slow.

JS behaves like an interpreter and compiler, depending on JS Engine.

Chapter 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 its explanation

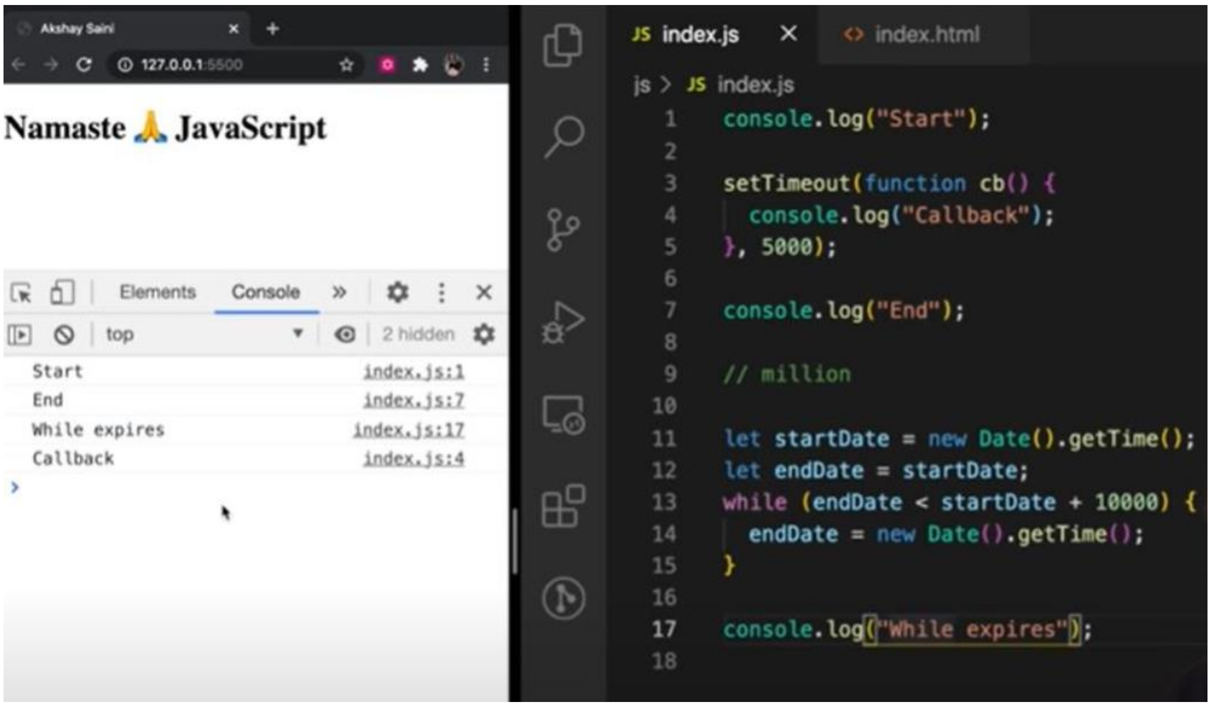
```
console.log("Start");
setTimeout(function cb() {
console.log("Callback");
}, 5000);
console.log("End");
// Millions of lines of code to execute

// 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?
```

Reason?

* First GEC is created and pushed in CallStack.
* Start is printed in console
* When setTimeout is seen, Callback function is registered into web API'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. 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** (When GEC popped after 10sec, Callback () is pushed into call stack and immediately executed (Whatever is pushed to CallStack is executed instantly).
* This is called as the **[Concurrency model](**https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop**)** 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). Don't let the call stack/main thread run indefinitely/ run for additional time.

* In below example, we are blocking the main thread. Observe Question and Output.



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

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

setTimeout(function cb() {
  console.log("Callback");
}, 5000);

console.log("End");

// million

let startDate = new Date().getTime();
let endDate = startDate;
while (endDate < startDate + 10000) {
  endDate = new Date().getTime();
}

console.log("While expires");
```

* 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**?

```
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 web API'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

# Chapter 18: Higher-Order Functions ft. Functional Programming

### Q: What is Higher Order Function?

Ans: Higher-order functions are regular functions that take one or more functions as arguments and/or return functions as a value from it. E.g.:

```
function x() {
   console.log("Hi");
};
function y(x) {
   x();
};
y(x);

// 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 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
// Over here 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 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++) { // this = radiusArr
      output.push(operation(this[i]));
   }
   return output;
}
console.log(radiusArr.calculate(area))
```

# Chapter 19: map, filter & reduce

> map, filter & reducer are Higher Order Functions.

## 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 creates a new array and returns it. It does not mutate the original array

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

// Task 2: Triple the array element

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

function triple(x) {  // Transformation logic
  return x * 3;
}
const tripleArr = arr.map(triple);

console.log(tripleArr); // [15, 3, 9, 6, 18]
```

// Task 3: Convert array elements to binary

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

function binary(x) { // Transformation logic:
  return x.toString(2);
}
const binaryArr = arr.map(binary);

// The above code can be rewritten as:
const binaryArr = arr.map(function binary(x) {
  return x.toString(2);
}

// OR -> Arrow function

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

console.log(binaryArr);
```

So basically, map function is mapping each and every value and transforming 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 a condition set by the filter method argument.

```
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 evaluates to true.

## Reduce function

It is a function which take all the values of array and gives a single output of it. It reduces the array to give a single output or returns a scalar value.

```
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
```

Calculate sum of array elements reduce function way

```
const sumOfElem = arr.reduce(function (accumulator, current) {
  // current represents current iteration array element.
  // accumulator represents a variable which gets updated depending on current context.
  // In this case accumulator gets updated in each iteration and returns an arr elements sum value at
Last.
  // 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
```

Find max inside array: Non-functional programming way

```
const array = [5, 1, 3, 2, 6];
function findMax(arr) {
   let max = 0;
   for(let i = 0; i < arr.length; i++) {
       if (arr[i] > max) {
           max = arr[i]
       }
   }
   return max;
}
console.log(findMax(array)); // 6
```

Find max inside array using reduce

```
const output = arr.reduce((acc, current) => {
  if (current > acc ) {
    acc = current;
  }
  return acc;
}, 0);
console.log(output); // 6
```

Note- acc is just a label which represents the accumulated value till now, we can also label it as max in this case

```
const output = arr.reduce((max, current) => {
  if (current > max) {
    max= current;
  }
  return max;
}, 0);
console.log(output); // 6
```

## Tricky MAP

```
const users = [
  { firstName: "Alok", lastName: "Raj", age: 23 },
  { firstName: "Ashish", lastName: "Kumar", age: 29 },
  { firstName: "Ankit", lastName: "Roy", age: 29 },
  { firstName: "Pranav", lastName: "Mukherjee", age: 50 },
];

// Get array of full name : ["Alok Raj", "Ashish Kumar", ...]

const fullNameArr = users.map((user) => user.firstName + " " + user.lastName);
console.log(fullNameArr); // ["Alok Raj", "Ashish Kumar", ...]

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

// Get the count/report of how many unique people with unique age are there
// Like: {29: 2, 75: 1, 50: 1}
// We should use reduce, why? we want to deduce some information from the array. Basically, we
want to get a single object as output

const report = users.reduce((acc, curr) => {
  if(acc[curr.age]) {
    acc[curr.age] = ++ acc[curr.age] ;
  } else {
    acc[curr.age] = 1; // adding prop value to the object acc if the key is not found
  }

  return acc;  // Updating the acc object every time in every iteration and finally a scalar value / single
modified or desired object is returned.
}, {})
console.log(report) // {29: 2, 75: 1, 50: 1}
```

## Function Chaining

```
// First name of all people whose age is less than 30
const users = [
  { firstName: "Alok", lastName: "Raj", age: 23 },
  { firstName: "Ashish", lastName: "Kumar", age: 29 },
  { firstName: "Ankit", lastName: "Roy", age: 29 },
  { firstName: "Pranav", lastName: "Mukherjee", age: 50 },
];
```

```
// function chaining
const output = users
  .filter((user) => user.age < 30)
  .map((user) => user.firstName);
console.log(output); // ["Alok", "Ashish", "Ankit"]
```

Homework challenge: Implement the same logic using reduce

```
Const output = users.reduce((acc, curr) => {
  if (curr.age < 30) {
    acc.push(curr.firstName);
  }
  return acc;
}, []);

console.log(output); // ["Alok", "Ashish", "Ankit"]
```