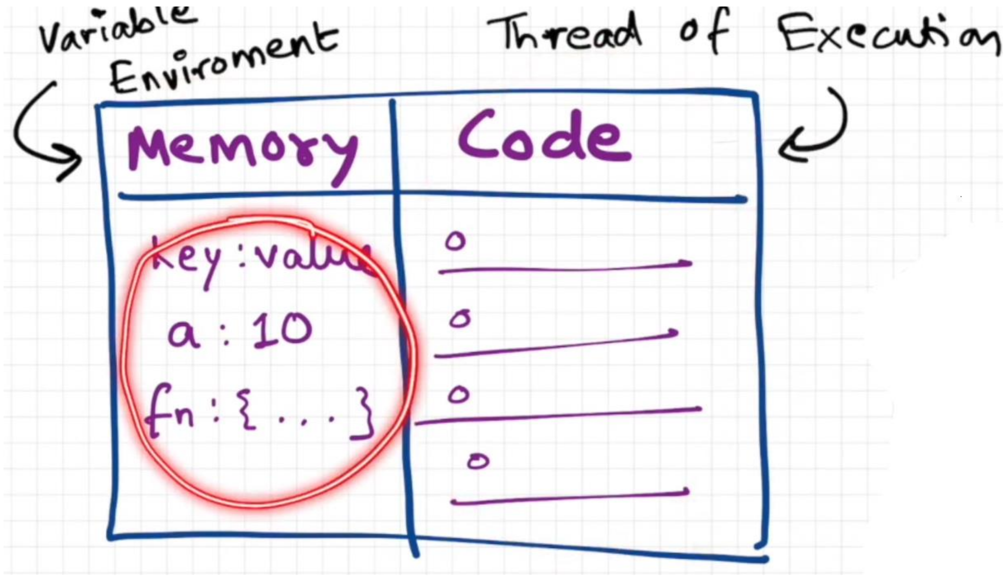
1. **How JS Works and Execution Context.**
   Everything in JavaScript happens inside the Execution Context.
   Execution context has 2 components, memory component and code component.
   In memory component all the variables and functions are assigned a memory.
   In code components, our code runs, line by line.



JavaScript is a synchronous single **threaded language, i.e. JS can only execute one command at a time, that too in a specific order, i.e. top to bottom. It can only execute the next line once the previous line is finished executing.**

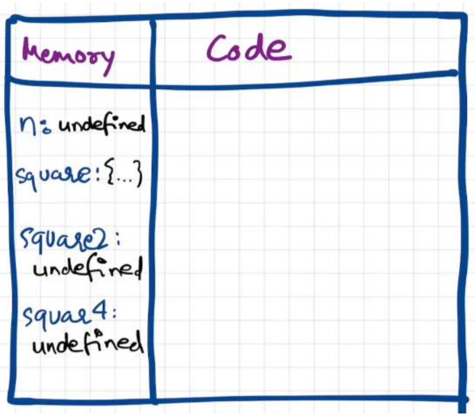
Whenever we run a JS Program, a Global execution context is created.
Execution context is created in Two phases –

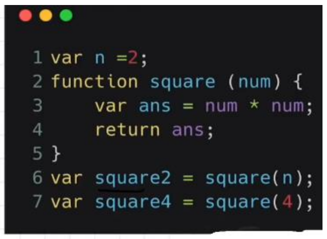
- Memory Creation Phase

  In this phase JS allocates memory to all functions and variables.

  For Variables it allocates memory with the value undefined as variable value.

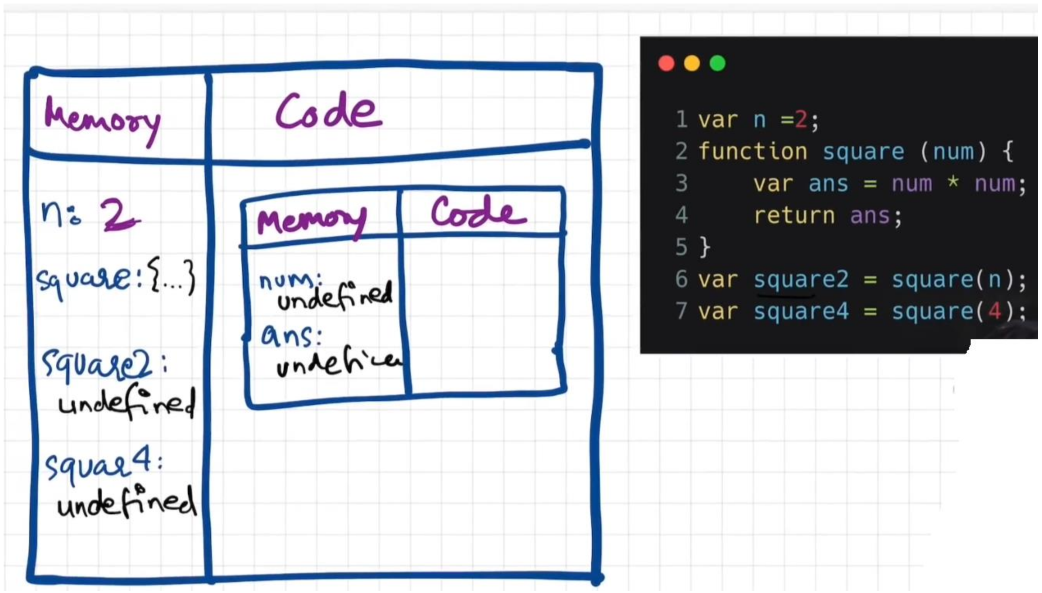
  For functions it stores the whole function.



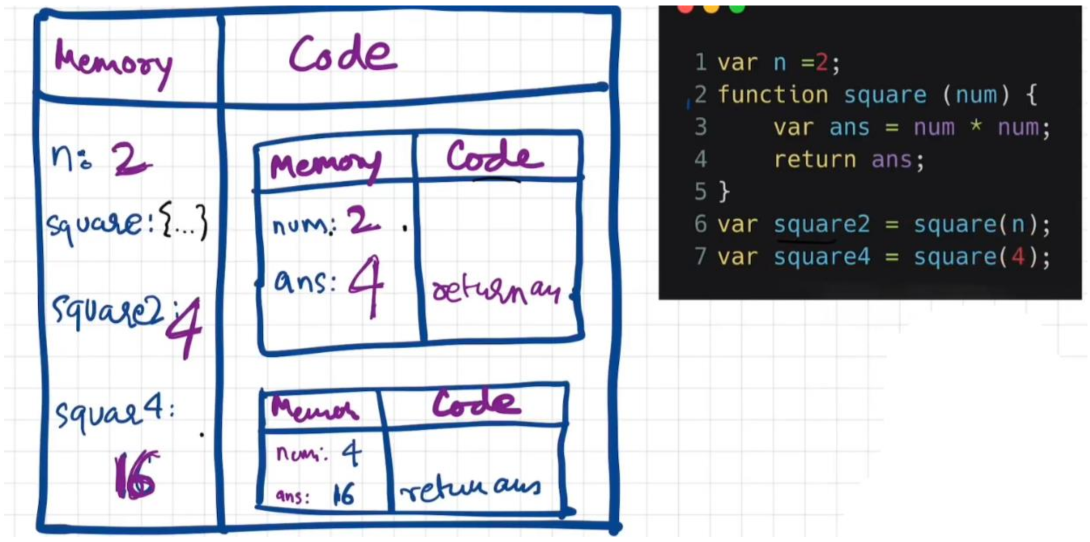


```
var n =2;
function square (num) {
    var ans = num * num;
    return ans;
}
var square2 = square(n);
var square4 = square(4);
```

- Code Execution Phase
  JS once again runs through the whole program and assigns the actual value to the variables in the memory, instead of undefined.
  Whenever JS finds a function execution in the 2nd phase, it creates a new execution context for that function.



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

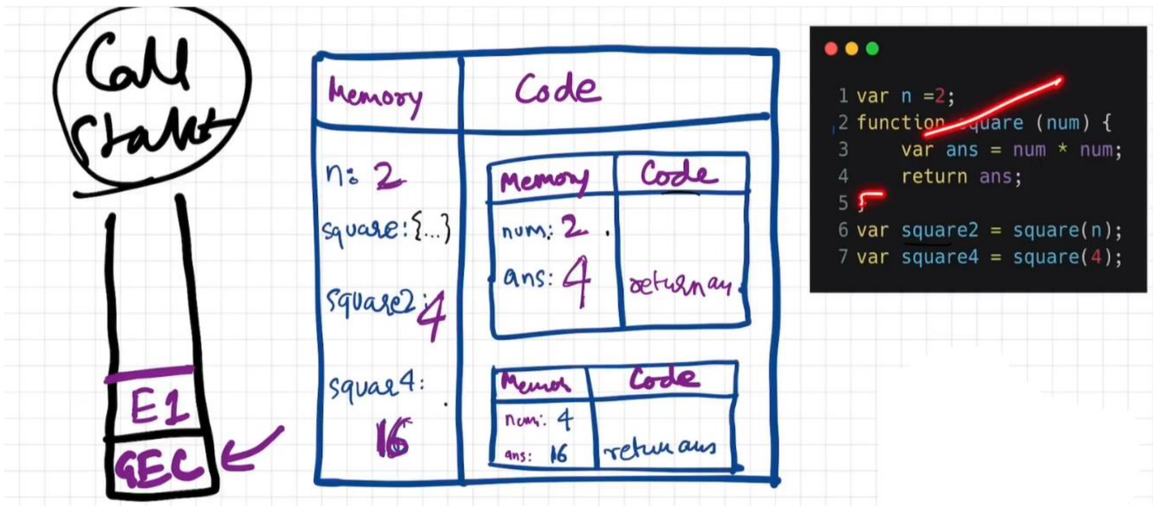


```
var n =2;
function square (num) {
    var ans = num * num;
    return ans;
}
var square2 = square(n);
var square4 = square(4);
```

Once the code execution context is finished executing, the whole Global execution context gets deleted from the call stack.

The Work of call stack is to contain Global and non-global execution context, and to append once the code starts executing and pop them out once the code is finished executing.

Call Stack is aka. Execution Context Stack, Program Stack, Control Stack, Runtime Stack, etc.



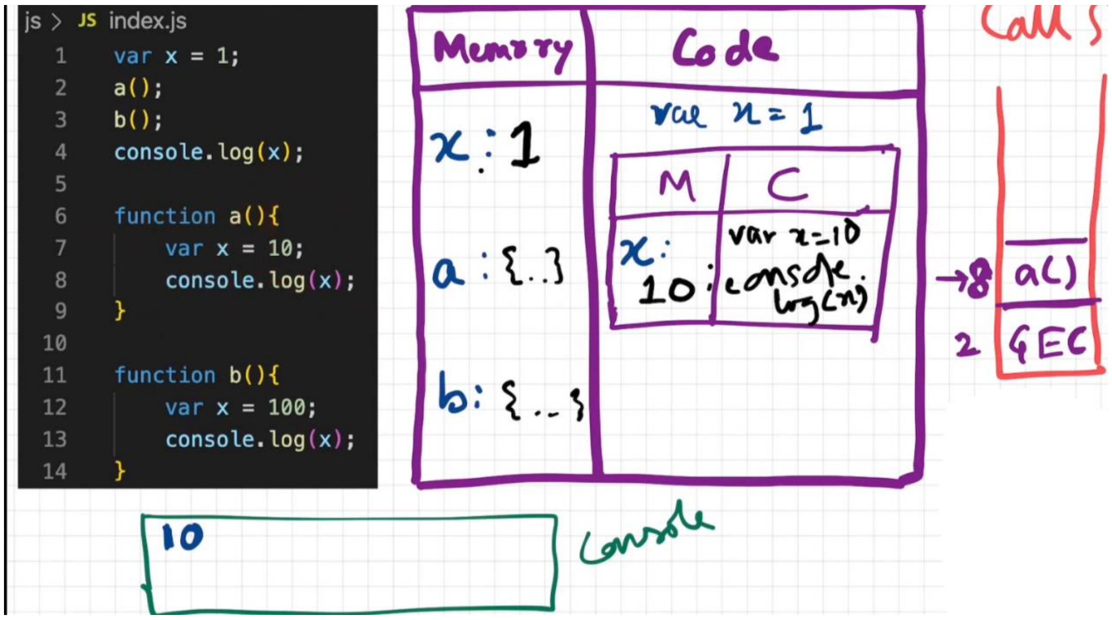
```
var n =2;
function square (num) {
    var ans = num * num;
    return ans;
}
var square2 = square(n);
var square4 = square(4);
```

Arrow functions don’t store the whole function just like normal function in the memory Creation Phase, because they are stored in a variable, so arrow functions stores undefined, just like It does In the case of normal variable.

Whenever a new Execution context Is created, it has a reference to the variables and functions that are written inside it and also has reference to the variables and functions of the parent Context.

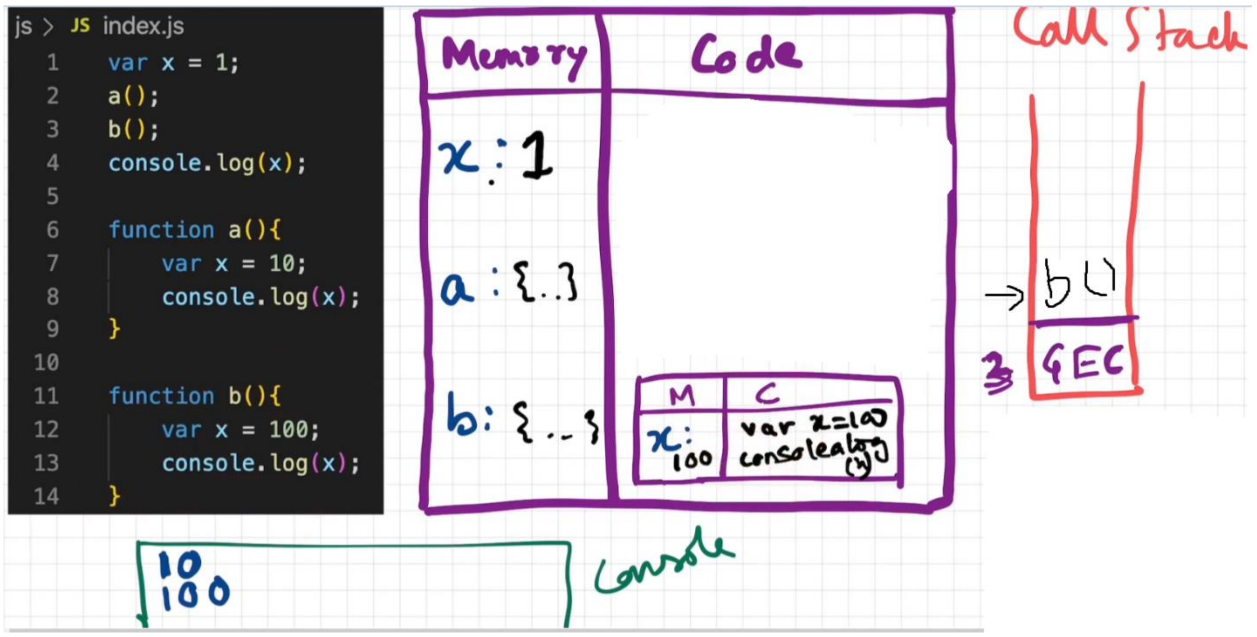
2. **Hoisting In JavaScript.**
   Hoisting is a process by which we can access var type variables and normal functions, even before they are being declared in the program.
   This happens because of the execution context. During the creation of execution context in Phase 1, in memory space normal function stores the whole code inside that function and the var type variables are assigned to be as undefined.
   “var” and “functions” are hoisted in the Global Space.

3. **How functions Work in JavaScript and Variable Environment.**
   The way function works in JS is, whenever a new function is invoked in the JS, a new Execution context is created, and inside that execution context we again go through 2 phases and the variables and functions inside that function are assigned memory and then the code In it is being run. After the function is done executing that specific function execution context is deleted. That’s the reason why the code inside the function does not affect the code outside that function.



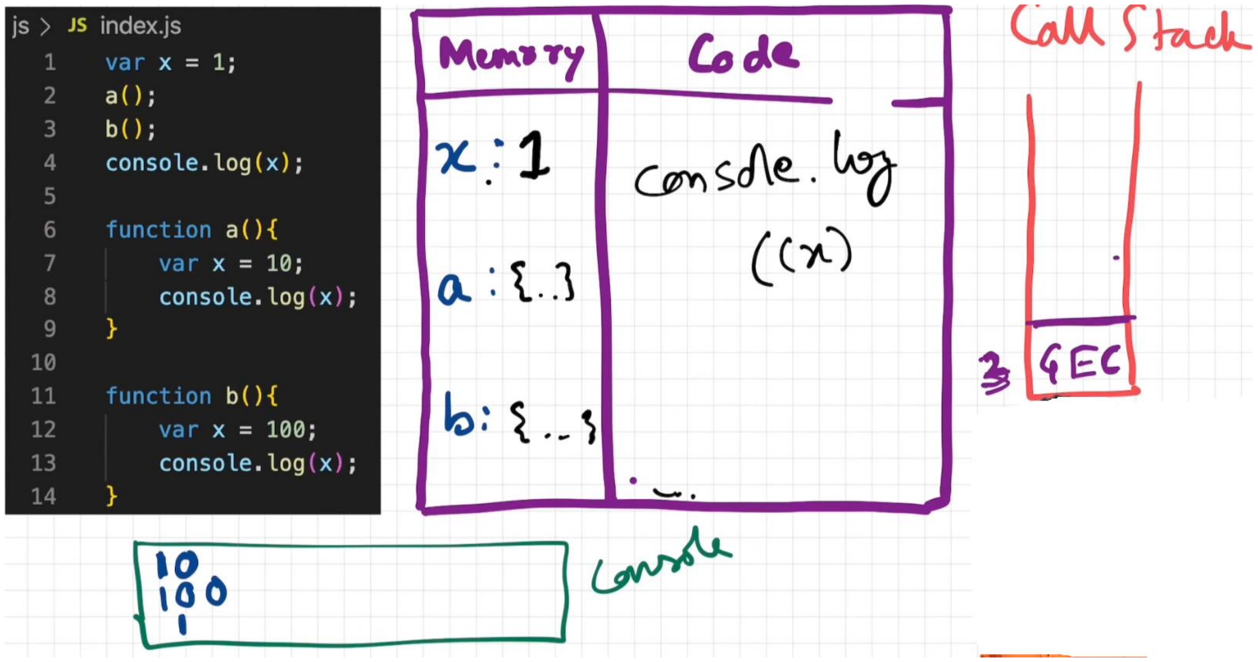
js > index.js

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

function a(){
    var x = 10;
    console.log(x);
}

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

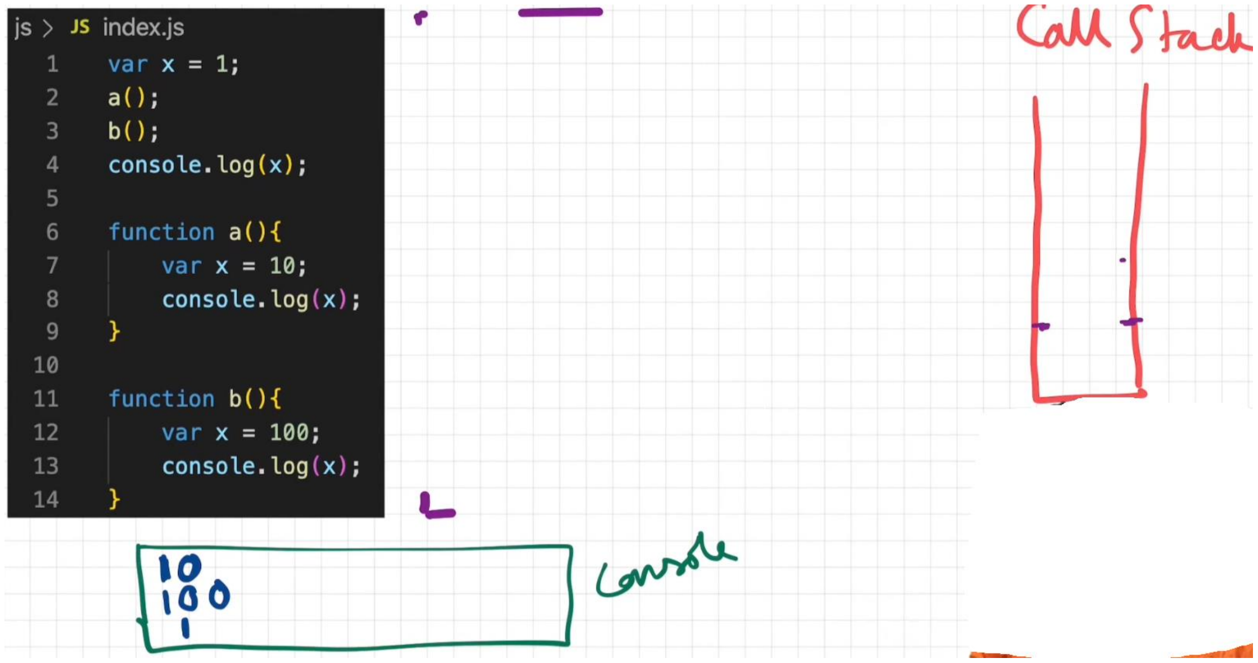


js > index.js

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

function a(){
    var x = 10;
    console.log(x);
}

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



```
js > JS index.js
var x = 1;
a();
b();
console.log(x);

function a(){
    var x = 10;
    console.log(x);
}

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



```
js > JS index.js
var x = 1;
a();
b();
console.log(x);

function a(){
    var x = 10;
    console.log(x);
}

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

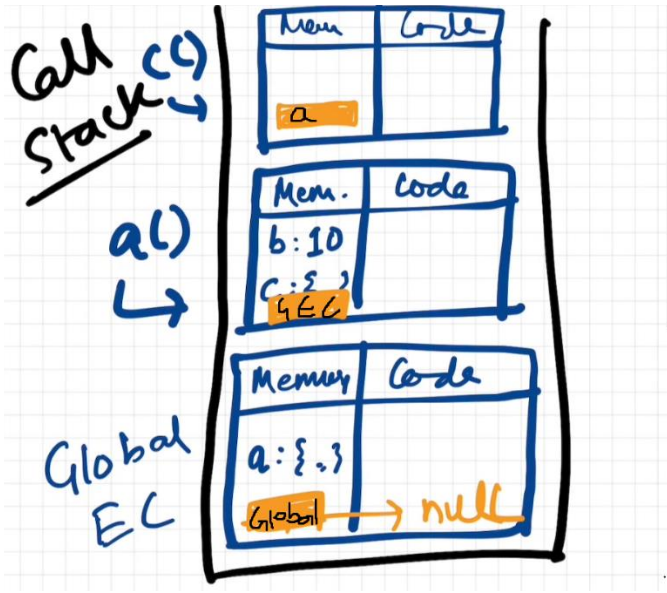
4. **this and window in JavaScript.**
   this and window functionality is given to us by JavaScript Engine. this and window are a huge object with bunch of properties. All the variables and functions that we declared are stored window object only. At Global level this also points to window object. We can also use this.var_name or window.var_name to access a variable.

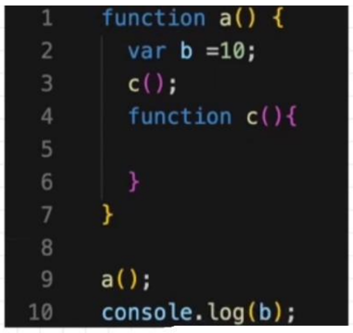
5. **Undefined vs not defined.**
Undefined = Undefined is like a placeholder for the Phase1 of Execution Context, i.e. memory Creation Phase. Also if we do not assign any value to a variable then also it will come out to be undefined, though it will be present in the memory, but corresponding to that, it's value will be undefined. We should never assign any variable value to be undefined. Use "null" instead.
Not defined = Not defined means that variable does not exists in the memory, i.e. we forgot to declare that variable in our code.

6. **JavaScript is a loosely types Language.**
It means that In JavaScript we don't have to tell the variable what data type are we going to store inside it. It identifies that by itself.

7. **Scope Chain and Lexical Environment.**
Scope means where we can access a specific variable or a function in our code.
When we say what is the scope of the variable "b". It means where in our code can we access the variable "b".
Another way of saying it, is "b" inside the scope of function x(), or class x(), or object x, etc.
Scope is dependent on the lexical environment.

Whenever an Execution Context is Created, a lexical Environment is also created. Lexical Environment is the Local Memory + reference to the Lexical Environment of it's parent. Lexical as a term means hierarchy or in a sequence.



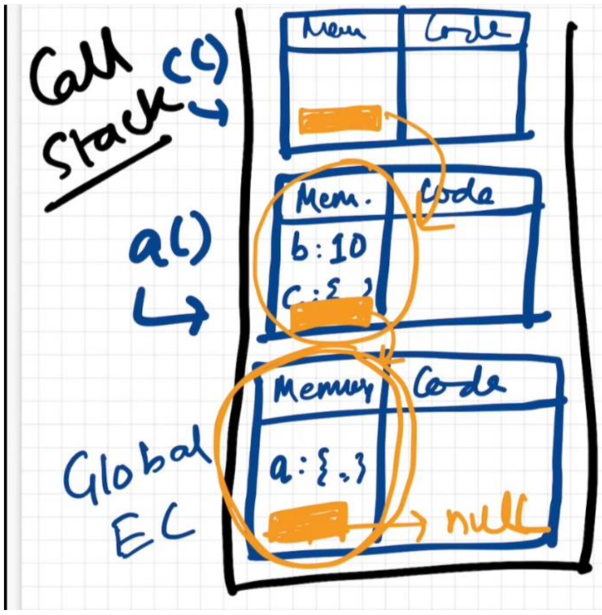


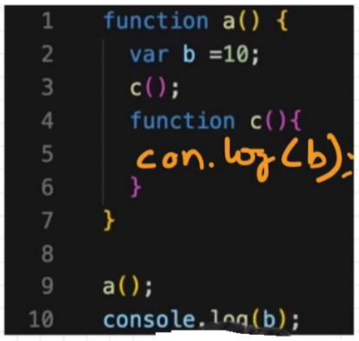
```
function a() {
  var b =10;
  c();
  function c(){

  }
}

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

- c() function Is lexically sitting inside a() function.
- a() is lexically sitting inside Global Execution Context.
- Whole code is pointing to outer lexical environment which basically points to null.

Whenever we try search, prints or access a variable it first search It inside the local memory. If it fails to do so, JS goes to the lexical environment of it's parent and tries to search it over there, and it keeps on doing it unless it finds the variable or ends up to the Outer Global Env. Which points to null, and we get an error that variable is not defined.

This way of searching variables lexically from parent to parent reference, this process is know as Scope Chain.





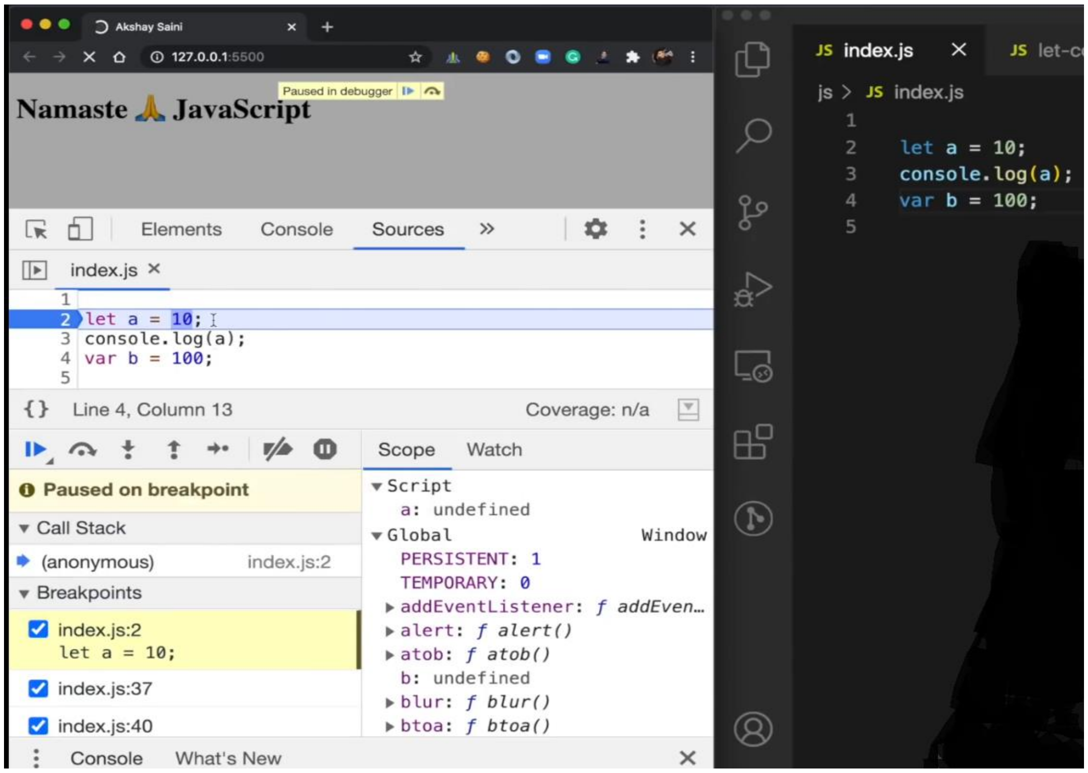
```
function a() {
  var b =10;
  c();
  function c(){
  con.log(b);
  }
}

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

8. **let and const and temporal dead zone.**
"let" and "const" declarations are hoisted in JS, but they are hoisted in a very different way as compared to "var" and "functions".
Const and let are hoisted in the Script Scope and not Global Scope. They are not hoisted on the Global object i.e. window object.

Temporal Zone is the time, since variables are hoisted, till it is initialized some value in the memory. When the variables are in temporal dead zone we can't access them.

Let and const variables are not present in the window object, so we can't do window.var_name;

We can't do re-declaration of let and const. Also we can't change the value of const once it is being initialized.



```
let a = 10;
console.log(a);
var b = 100;
```

```
▼Script
  a: undefined
▼Global                Window
  PERSISTENT: 1
  TEMPORARY: 0
 ▶addEventListener: f addEven…
 ▶alert: f alert()
 ▶atob: f atob()
  b: undefined
 ▶blur: f blur()
 ▶btoa: f btoa()
```

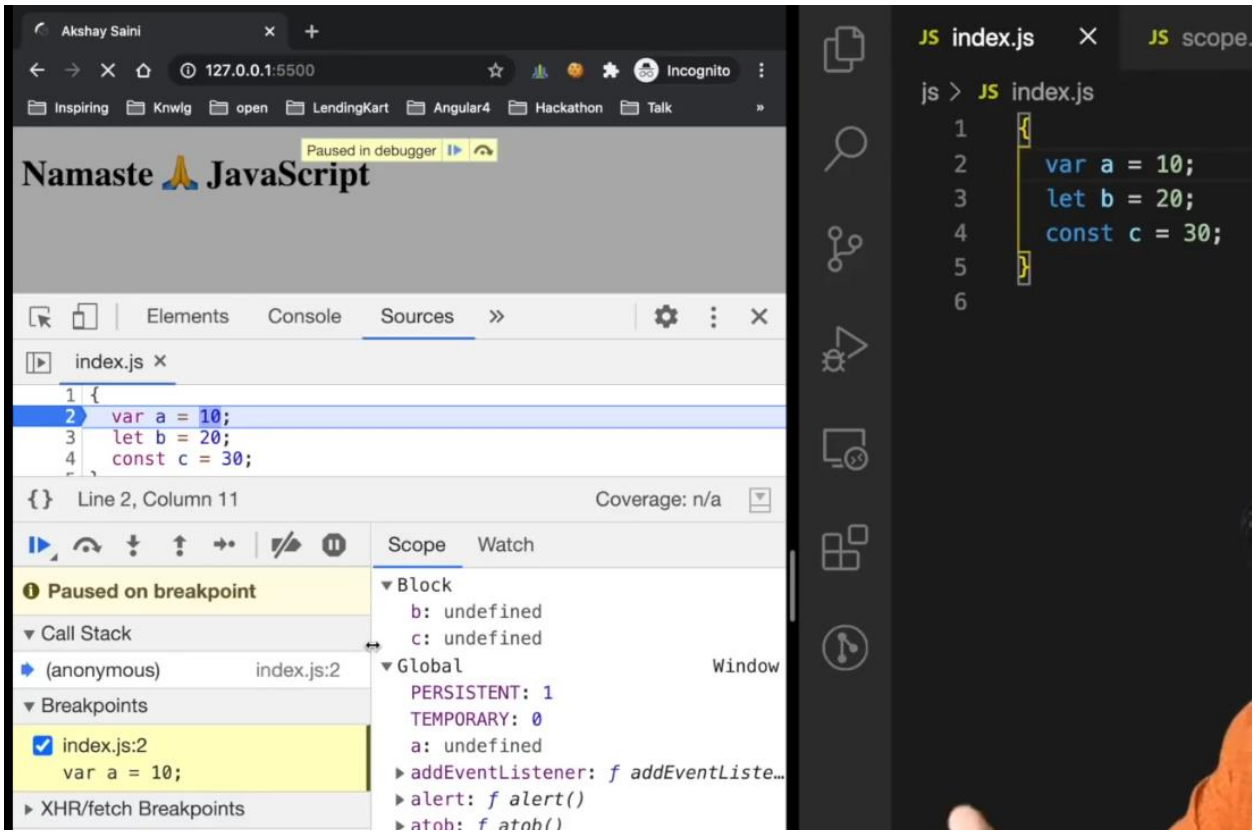
9. **What is Block, Block Scope and Shadowing.**
Block - Block is used to combine multiple JS statements into a group/block. We need to group these statements together so that we can use multiple statements in a place where JS expects only one statement.

```
if(true) return true;
```

This is a perfectly valid JS statement, but now let's imagine we need to pass multiple lines of statement inside the if condition, in that case, we need to use a scope. That's the use case of Block to group multiple statements together where JS expects only on statement.

```
if(true){
    var a = 10;
    console.log(a);
}
```

Block Scope – What all variables and functions we can access inside that block.



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

```
Paused on breakpoint
Call Stack
(anonymous)    index.js:2
Breakpoints
index.js:2
var a = 10;

Scope
▼Block
  b: undefined
  c: undefined
▼Global                          Window
  PERSISTENT: 1
  TEMPORARY: 0
  a: undefined
 ▸addEventListener: f addEventListe…
 ▸alert: f alert()
 ▸atob: f atob()
```

If we declare “var”, “let”, and “const” inside a block, let and const are placed inside block scope in the memory whereas var is placed inside Global Scope. That’s where the statement **"var” is function or global scope and “let” and “const” are block scoped**, comes from. We can’t access let and const outside the scope whereas we can access var outside the scope. Once code inside the block is fully executed, it get’s removed from the scope. That’s why we cannot access variables inside the block.

Shadowing – If we try to declare a same named variable outside the scope which is of type “var”, then the variable inside the scope shadows the value of variable outside the scope, i..e. it changes it’s value for the rest of the code.

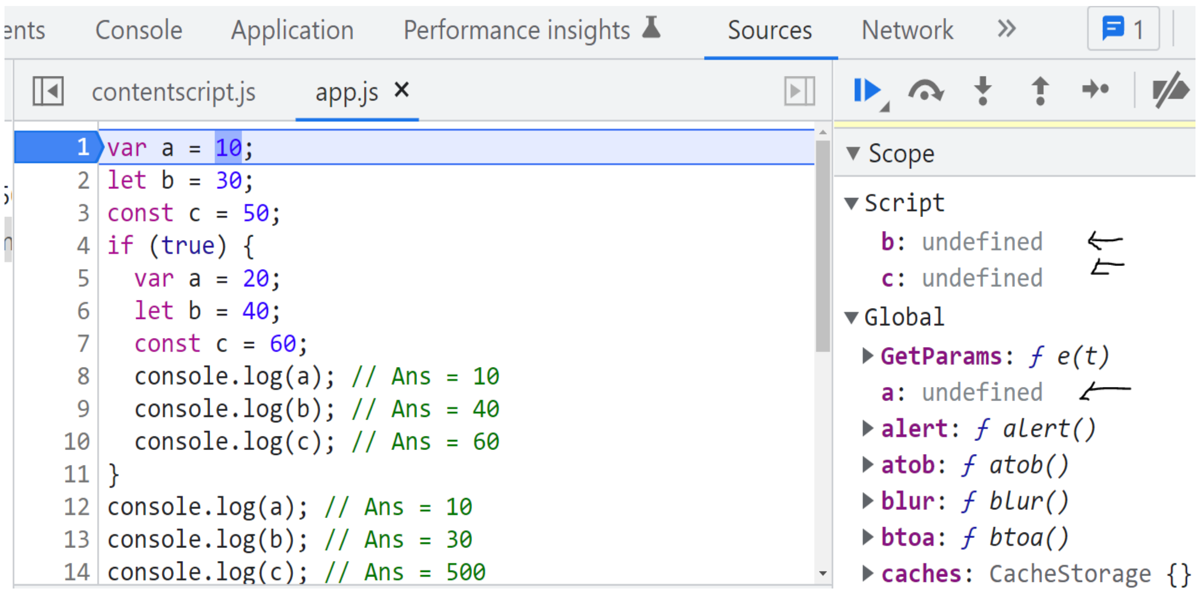
Shadowing works same for both Block and Function.

Arrow functions and Normal Functions works same in case of Scoping.

```
var a = 10;
let b = 30;
const c = 50;
if(true){
    var a = 20;
    let b = 40;
    const c = 60;
    console.log(a); // Ans = 10
    console.log(b); // Ans = 40
    console.log(c); // Ans = 60
}
console.log(a); // Ans = 10
console.log(b); // Ans = 30
console.log(c); // Ans = 50
```

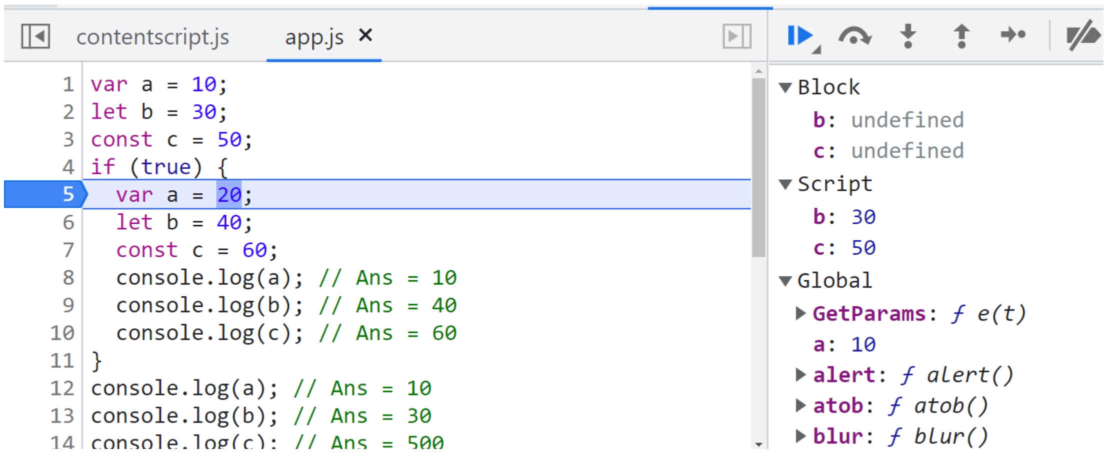
Since both the var a, are referencing to the same memory location in the Scope i.e. Global, it's value get's updated.

In case of let and const, the let and const declared on the Global Scope is referencing to the Script Scope in the Memory, whereas let declared inside the block is referencing to the Block Scope in Memory, so When we print let and const inside the scope, it first check for it's value in the local environment, if it can't find it there, it goes to it's parent lexical scoping.



```
var a = 10;
let b = 30;
const c = 50;
if (true) {
  var a = 20;
  let b = 40;
  const c = 60;
  console.log(a); // Ans = 10
  console.log(b); // Ans = 40
  console.log(c); // Ans = 60
}
console.log(a); // Ans = 10
console.log(b); // Ans = 30
console.log(c); // Ans = 500
```

```
▼ Scope
▼ Script
    b: undefined
    c: undefined
▼ Global
  ▶ GetParams: ƒ e(t)
    a: undefined
  ▶ alert: ƒ alert()
  ▶ atob: ƒ atob()
  ▶ blur: ƒ blur()
  ▶ btoa: ƒ btoa()
  ▶ caches: CacheStorage {}
```



contentscript.js app.js ×

```
var a = 10;
let b = 30;
const c = 50;
if (true) {
  var a = 20;
  let b = 40;
  const c = 60;
  console.log(a); // Ans = 10
  console.log(b); // Ans = 40
  console.log(c); // Ans = 60
}
console.log(a); // Ans = 10
console.log(b); // Ans = 30
console.log(c); // Ans = 500
```

```
▼ Block
    b: undefined
    c: undefined
▼ Script
    b: 30
    c: 50
▼ Global
  ▶ GetParams: ƒ e(t)
    a: 10
  ▶ alert: ƒ alert()
  ▶ atob: ƒ atob()
  ▶ blur: ƒ blur()
```

Illegal Shadowing – If we try to shadow variable of one data type into another data type inside a block or function, then JS throws an error and it's aka. Illegal Shadowing.

Examples of Legal Shadowing.
The reason why these are legal shadowing is because variables should remain inside it's boundaries. The variable should not interfere/collide with other variable of the same inside the memory/scope.

```
var a = 20;
{
  let a = 30;
}
```

```
let a = 20;
function (){
  var a = 30;
}
```

```
const a = 20;
{
  const a = 100;
  {
    const a = 200;
  }
}
```

In the above case 2 block scopes will be created in the Memory.

10. **What are Closures ?**

    A closure is a function bind together with it's lexical/outer environment, and has access to it's parent lexical scope. When functions are returned form a closure, it remembers it lexical scoping.

    The function along with the reference to the variables are returned.

    Uses/Advantages of Closures –

    Currying

    Memoize

    Function like once

    setTimeout

    Data Hiding or Data Encapsulation

    Disadvantages of Closures –

    Takes up more memory space.

    Memory Leak because of closure function that is still holding value/reference to the outer scope variables.

11. **What is Garbage Collector ?**

    Whenever there is some unused variables or functions it takes it out of memory, freeze out of the memory.

12. **More about functions.**

    - Function statement aka Function declaration

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

    - Function Expression

    ```
    var b = function () {
      console.log("Hello again mate!!!")
    }
    ```

    - Anonymous functions = Functions that don't have a name. Anonymous functions are used in a place where functions are used as values.

    ```
    var b = function () {
      console.log("Hello again mate!!!")
    }
    ```

    - Named function Expression

    ```
    var b = function x () {
      console.log("Hello again mate!!!")
    }
    ```

Still we need to invoke the function as "b()" and not as "x()".

- Difference b/w parameters and Arguments.

```
var b = function (parameters) {
  console.log("Hello again mate!!!")
}
b(arguments)
```

- First Class Functions
  JavaScript allow us to pass functions as parameter to another function and even return functions from other functions. Aka. First Class Citizens.

```
var b = function (param1) {
  console.log(param1);
};

b(function () {});
```

```
var b = function () {
  return function () {};
};

b();
```

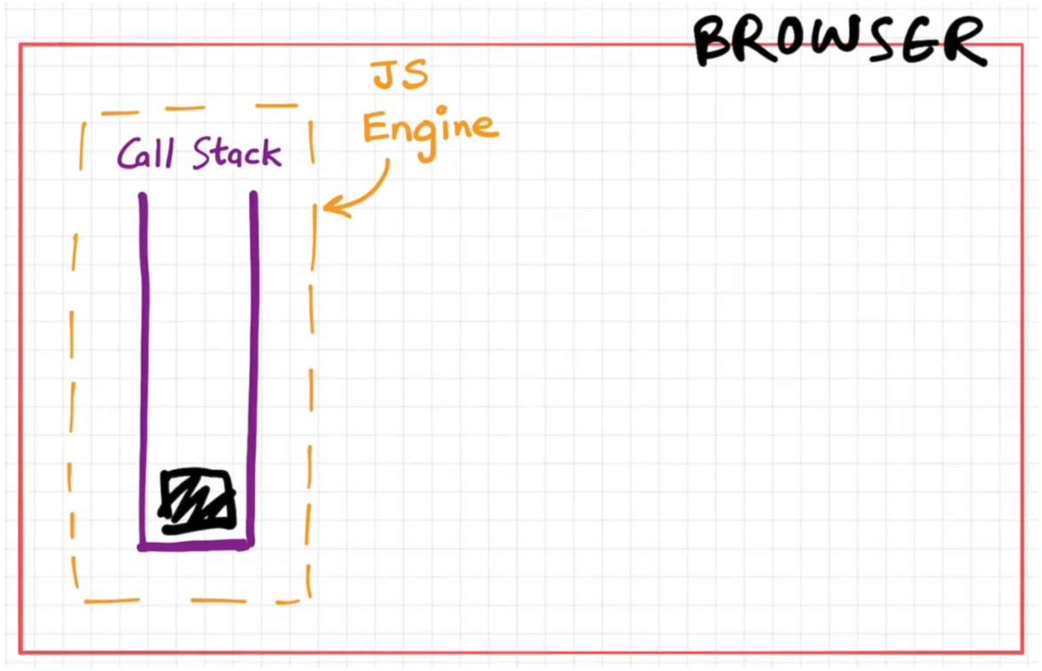
13. Callback Functions In JS.
    The function that we pass into another function is called a Callback function.

```
function x(param1) {

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

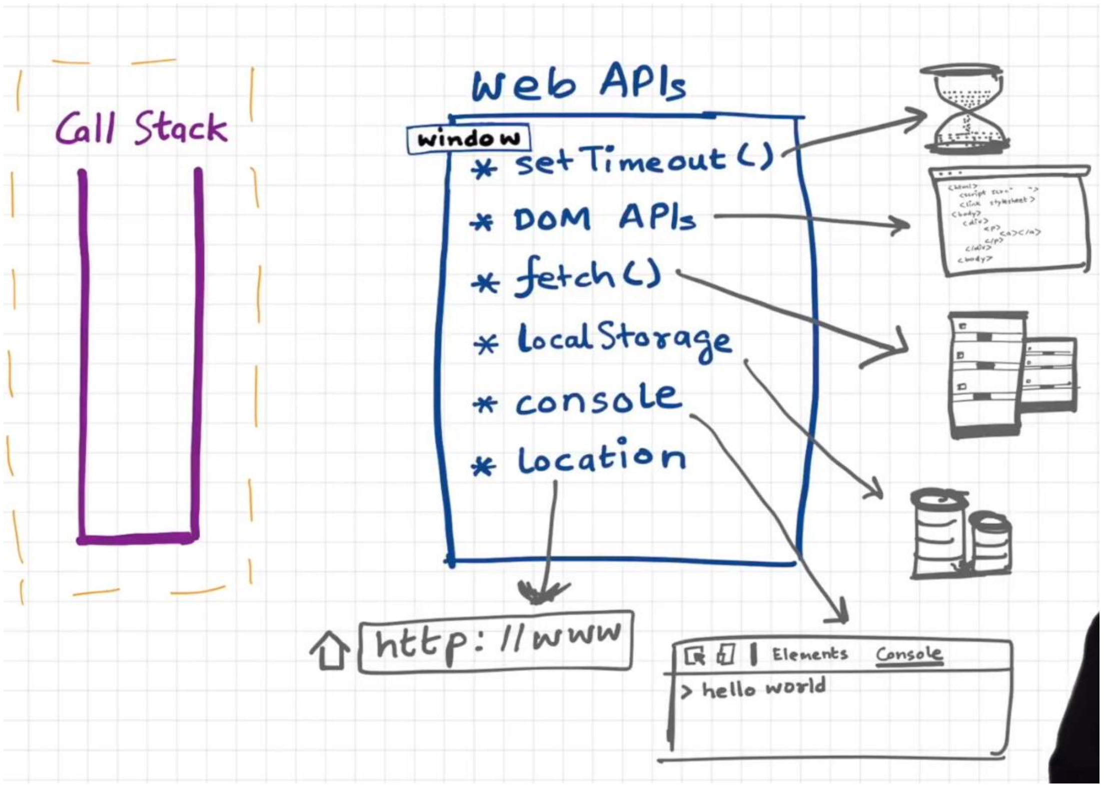
function y () is called the callback Function.
The asynchronous functionality of JS is performed using Callback Functions only.
If an operation is Taking too much time to execute and because of that other operations get delayed, this delay/block is know as Blocking the Main Thread.

14. Event Loops and Asynchronous JavaScript.

The call stack is inside the JS Engine. And JS engine is run inside browser.



The browser provides us with a Bunch of Web-API's



We can access these Web API's in our JS. Browser gives access to JS Engine to access all these properties. These Browser API's are present inside the global window object.

When a callback function of Web API is run, JS takes that callback function and registers the callback inside the browser Web API. As soon as the whole code of JS is executed, the callback function that is registered by the Web API needs to be sent back to call Stack to Execute. Before sending callback function to call Stack, first callback function is send to Callback Queue, it cannot directly go to the call stack, it goes to the call stack through callback Queue.

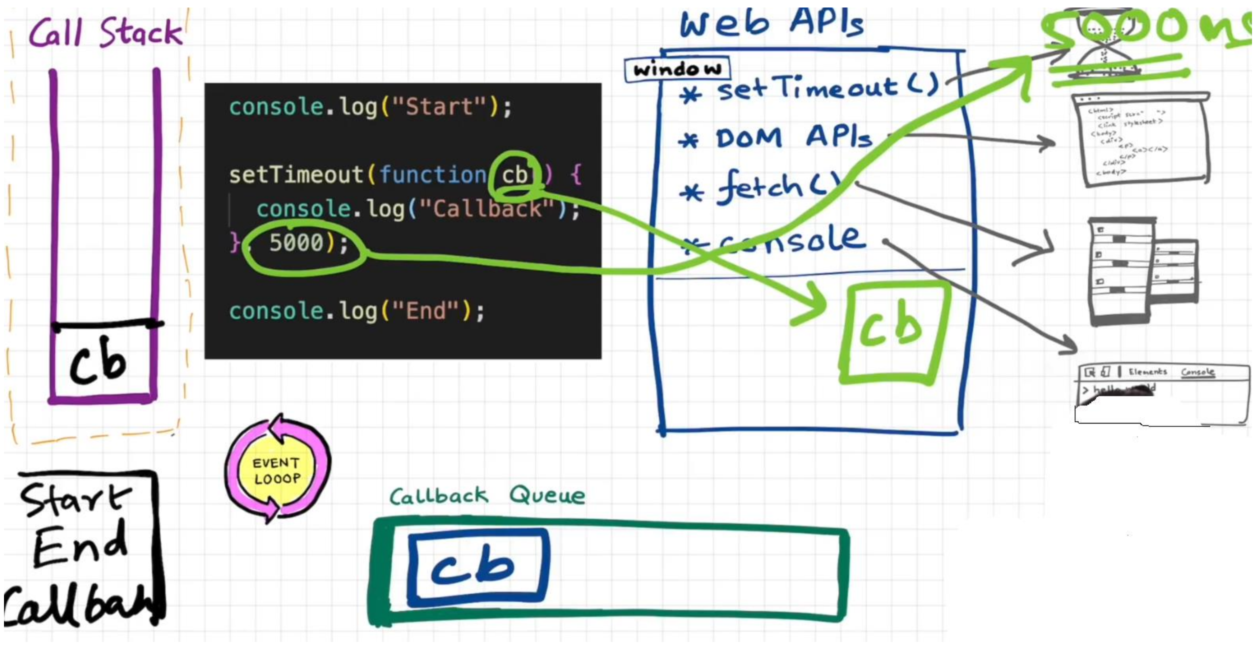
The Job of the event Loop is to check if the call Stack is empty an if we have something in the Callback Queue and put the callback functions that are present in the callback Queue to the call Stack.

The execution context of that callback function is created in the Call Stack and it executes the code inside that function line by line.

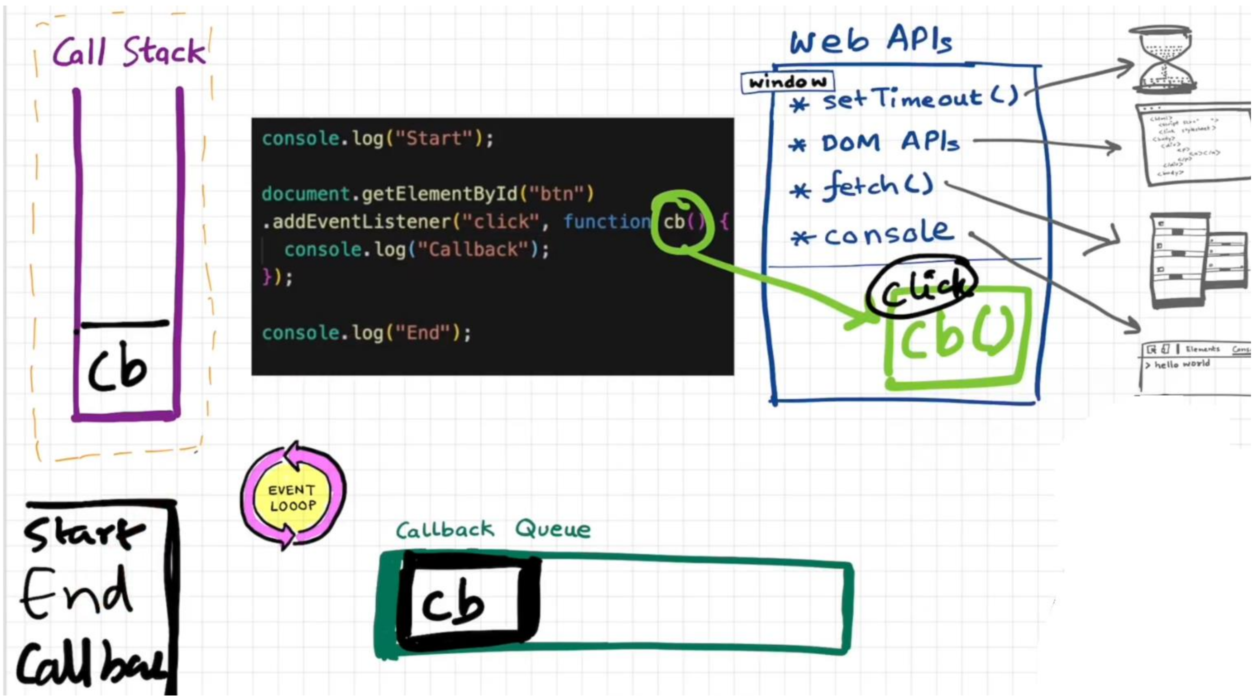
We also have a Microtask Queue, which is like a Callback Queue only, the difference is it has Higher Priority as Compared to Callback Queue. The functions in the Microtask Queue are executed first and then the functions in the Callback Queue. The type of functions that are passed in the Microtask Queue are Promises and Mutation Observer.

No matter how many number of Callback function we have in MicroTask Queue, first all the functions of the Microtask Queue are Executed then only Callback Queue gets the chance to execute it's callback functions.

If for some reason every time the callback function from the Microtask Queue produces a new Callback Function that belongs to the Microstask Queue, in that Case callback functions of the Callback Queue never gets the chance to execute.

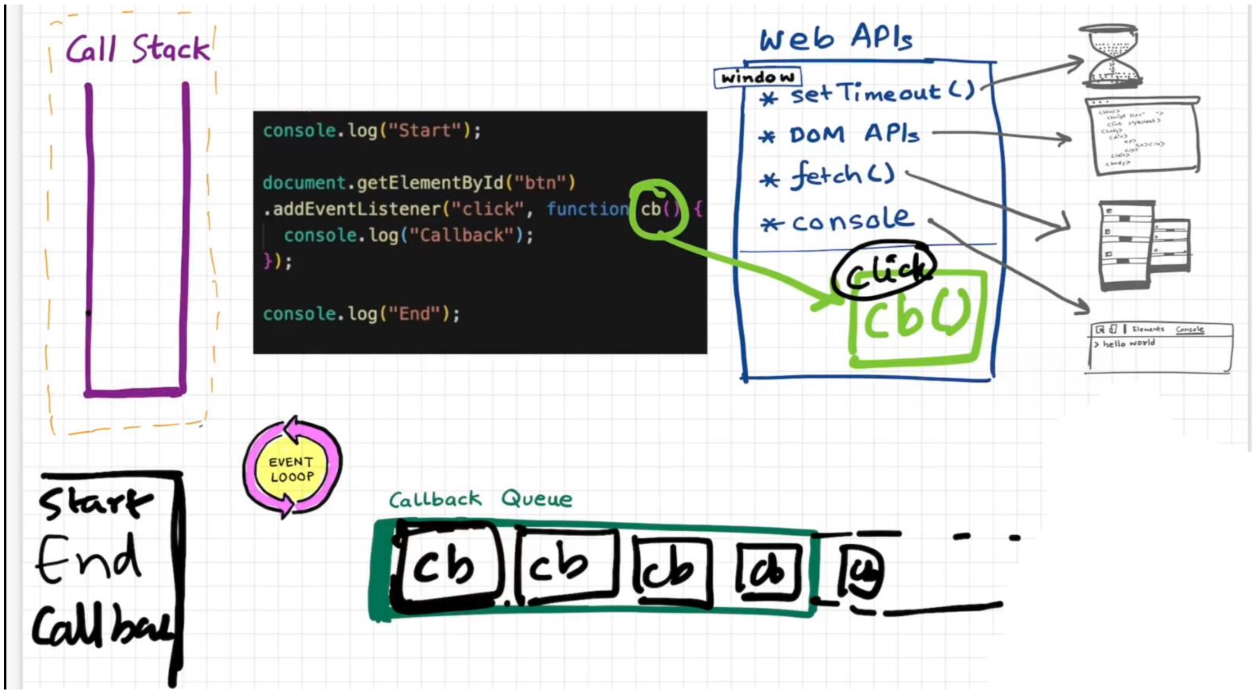


```
console.log("Start");

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

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

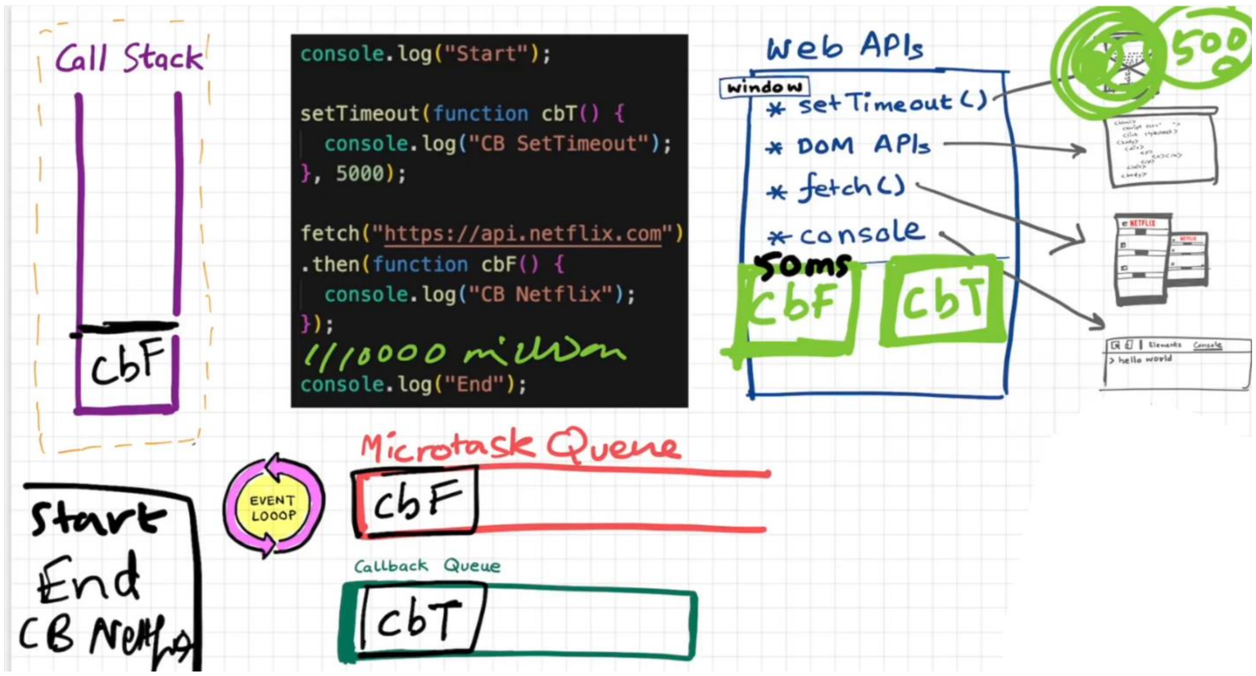


```
console.log("Start");

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

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

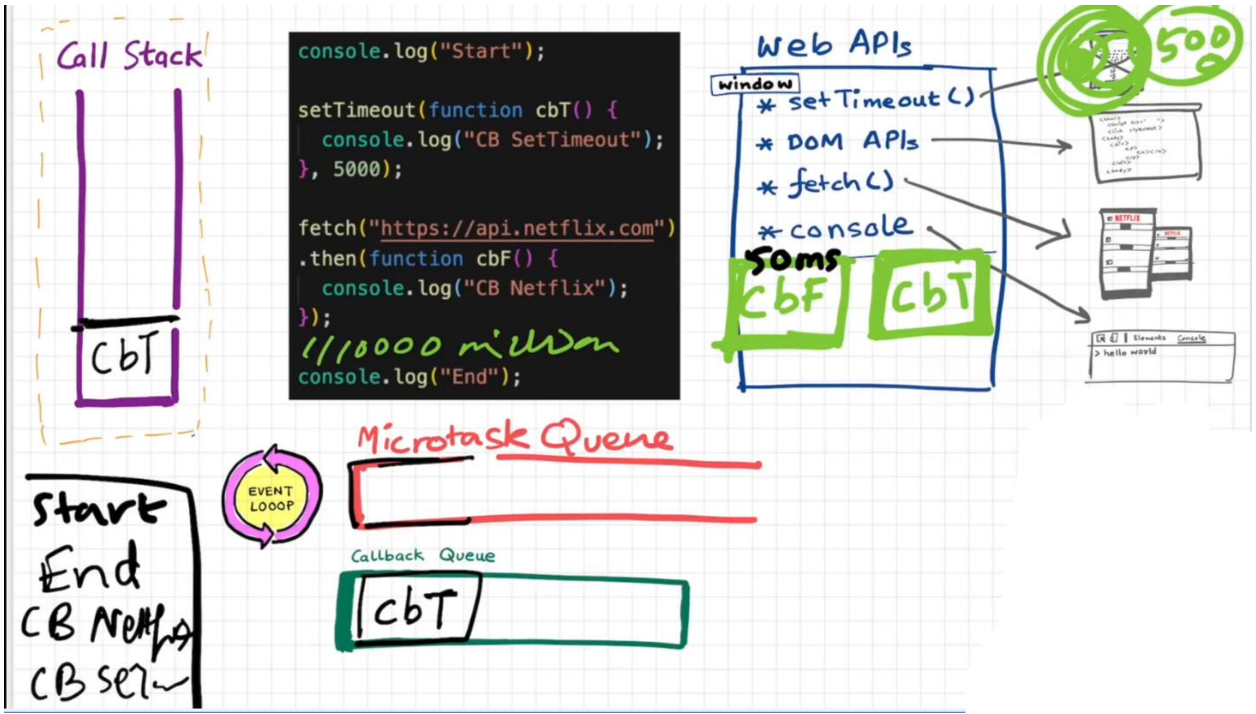
If the user click multiple times on the button we will have multiple callback function in the Callback Queue waiting to be executed. They are executed/sent to Call Stack in Sequence in FIFO Pattern. That's why we need a Callback Queue.



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

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

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



```
console.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");
```



```
console.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");
```