# Internal Working of JS | JS Execution

***Everything in JavaScript happens inside an Execution Context***.

**Execution Context**

**Code | Thread of Execution**

**Memory | Variable Environment**

Here whole code gets executed one line at a time.

Here all the variable and function are stored as a key value pair.

***JavaScript is a synchronous single-threaded language.***

This Execution Context is created into **two phases**.

1. **Memory Creation phase**

JS engine will skim through the whole JS code and **store variable and function as a key: value pair in Memory**.

i.e., Memory is allocated to all variable & functions even before code start executing.

*Before code execution Phase/ while memory creation phase*  
{

Variable: undefined, // **undefined** *allocated memory to the variable but value is not yet set so* **not defined** *not allocated memory to the variable.*

Function: {…} (stores the whole function code)

}

**Arrow Function**- var x = () => {…}

**Function Expression**- var y = function {…}

Are stored as undefined. i.e., {x: undefined, y: undefined}

1. **Code Execution phase**

*While Code Execution* line by line the values of variables get change and replaced with actual value.

Every time a new function is found a new **Local Execution Context** is created inside Global Execution context.

Execution context order is maintained by **Call Stack**.

As soon as return is observed the Local execution context get destroyed and popped out from call stack.

After whole code gets executed Global execution context also get destroyed.

***Call stack is used to maintains the order of execution of Execution Contexts.***

Call Stack | Execution Context stack | program stack | control stack | runtime stack | machine stack

# Hoisting

Through hoisting we can access the variables and functions even before initializing it.

If we try to access a variable which is not defined into the memory (execution context) then it will throw an error that reference of variable is not defined (means variable is not defined into memory/ variable is nowhere initialized into program).

When we try to call an arrow function before initializing of the function it will throw an error.

As it behaves like a variable and value is set to undefined in global execution context thus calling such function result error.

*As soon as we run JavaScript code in* ***browser****.* JS engine creates a

1. **Global Execution Context**
2. **Global Object (Window Object)**
3. **This opeartor**.

Even for empty JS file.

**At Global level this will point to window object**. (***this === window # true***).

And every variable gets attach to window object (global space).

var a = 10;

console.log(window.a); // 10

console.log(this.a); // 10

console.log(a); // as it automatically assumes it referring to global space (window.a). // 10

***JavaScript is loosely typed language.* | Variable mutation**

It does not attach any specific type of data type to the variables.

var a; // undefined

a = 10; // number

a = ‘hi’; // string

a = false; // Boolean

a = undefined; // bad practice

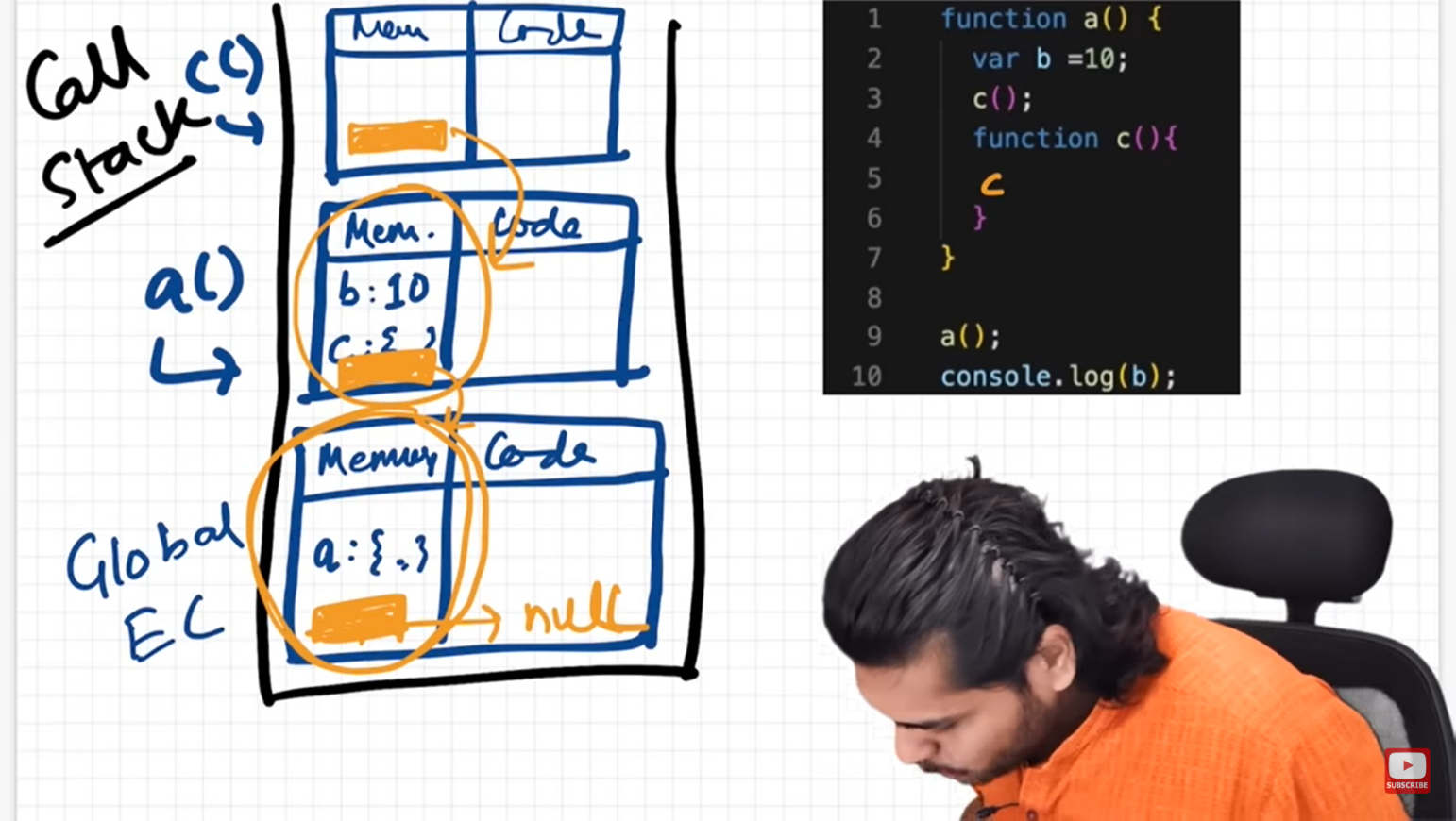
# Scope | Lexical Environment | Scope Chain

Where a variable or function can be access in a code.

Wherever execution context is created a lexical environment is also created with it.

Lexical Environment is the local memory along with the lexical environment of its parent.

In memory space of child execution context contains a reference to the lexical environment of its parent.



Lexical Env of c = local memory + reference to the parent lexical env (lexical env of a)

Lexical Env of a is referring to global execution context

Lexical Env of global execution context is referring to the null

# Temporal Dead Zone | let & const

Along with var***, let and const declarations are hoisted***

But they are in temporal dead zone.

***Temporal dead zone is the time between let and const is hosted and till it is initialized some value.***

As soon as we declare let & const variables in a file it gets hosted to the top of the execution (as undefined) and placed in script space (Temporal dead zone) and not in global space as soon as value is assigned to the let & const variable it then placed inside the separate space not in global space. If we try to access let variable before assigning a value to it. It throws a reference error.

Block Scope

Block is defined by {…} anything inside it considers as block scope.

Block it is used to group/wrap multiple statements inside it just like if {…}.

Each block has its own lexical scope.

All scope rule for function and arrow function are same.

Closure

A function binds/ bundle together with its lexical environment.

Function along with its lexical scope forms a closure.

When a function is returns from another function, they still maintain their lexical scope.

Whenever function is returned from another function it is not just a function is returned it is a closure (function along with its lexical scope).

A screenshot of a computer

Description automatically generated

**Uses of Closure**

* Module design Pattern
* Currying
* Functions like once
* Memoize
* Maintaining state in async world
* setTimeouts
* Iterators
* Data Privacy

**Disadvantages of Closure**

* Over consumption of memory
* Variables inside the closure are not garbage collected till the program expires

setTimeout(fun, time) => take a callback function and store in memory and attached a timer to it

A screenshot of a computer

Description automatically generated

The difference between function expression & Statements comes from hoisting and this operator.

Anonymous Functions

A function without a name is anonymous function

First-Class Function

A programming language is said to have First-class functions if functions in that language are **treated like other variables**. So, the functions can be assigned to any other variable or passed as an argument or can be returned by another function.  
JavaScript treat function as a **first-class-citizens**. This means that functions are simply a value and are just another type of object.

Higher-Order Function

Higher-order functions are functions that work with other functions, meaning that they take one or more functions as arguments and can also return a function. Think of higher-order functions as functions that **exist to help other functions**.

call(), apply() & bind()

A screenshot of a computer

Description automatically generated

Currying using bind

A screenshot of a computer

Description automatically generated

Currying using closure

A screenshot of a computer

Description automatically generated

Callback

Functions are first-class citizen.

We can take a function and pass it to another function this other function is called as callback function.

Used for async. (setTimeout(), setInterval())

setTimeout(fun, 1000); setTimeout take a callback function within it

Need to Remove Event Listeners

Event Listeners are heavy. It takes memory.

Whenever we attach an event listener it forms a closure. Even if call stack is empty (i.e., no code is executing) but still the program variable is not freeing up the memory. Automatic garbage collection doesn’t work. Hence consumption of lot of memory.

Web APIs is not part of JavaScript. They are part of browser.

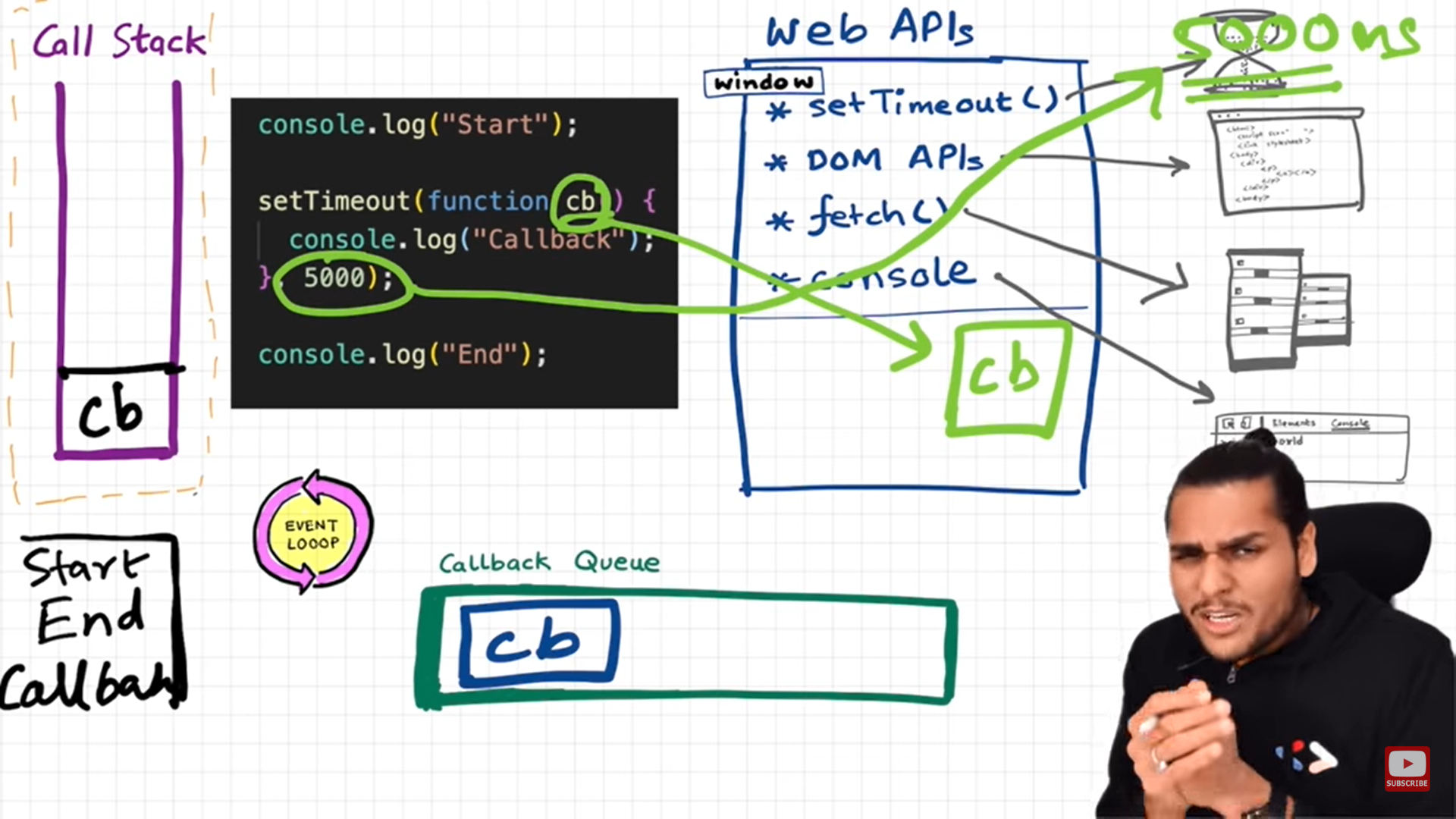
Browser gives access of Web API in call stack (i.e., JavaScript engine) using global object (window) to use all the feature of Web API.

Diagram

Description automatically generated

JS Engine

4. Once inside the call Stack the function will get execute as soon as possible.



3. Event Loop will continuously monitor call stack & callback queue. If call stack it empty, then it will push the callback function into call stack and remove it from queue.

2. Once the timer gets expire for callback function. Then function is push inside the callback Queue

1. Register the callback function and wait for timer to get expire.

A picture containing text, person

Description automatically generated

Task Queue

Microtask queue takes the higher priority task (callback functions from promises, mutation observer).

Function inside microtask queue is push before callback queue to call stack as soon as call stack get empty

JavaScript is a dynamically typed language. In a dynamically typed language, the type of a variable is checked during **run-time**in contrast to a statically typed language, where the type of a variable is checked during **compile-time.**