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

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

**Execution Context**

**Code /  
Thread of Execution**

**Memory /  
Variable Environment**

Where whole code gets executed one line at a time in a specific order.

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

The 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 the 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 &* **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 the actual value.

Every time a new function is found a new **Local Execution Context** is push inside the call stack.

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

As soon as return is observed the execution context (global/ local) get destroy 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 JS 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 (Reference Error) that reference of variable is not defined (means variable is **not defined** into memory/ variable is nowhere initialized in the 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 execution context thus calling such function result TypeError.

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

1. **Global Execution Context**
2. **Global Object (In case of browser its Window)**

Even for empty JS file.

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

And every variable with var keyword gets attach to window object (global object).

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 = null; // object

a = undefined; // bad practice

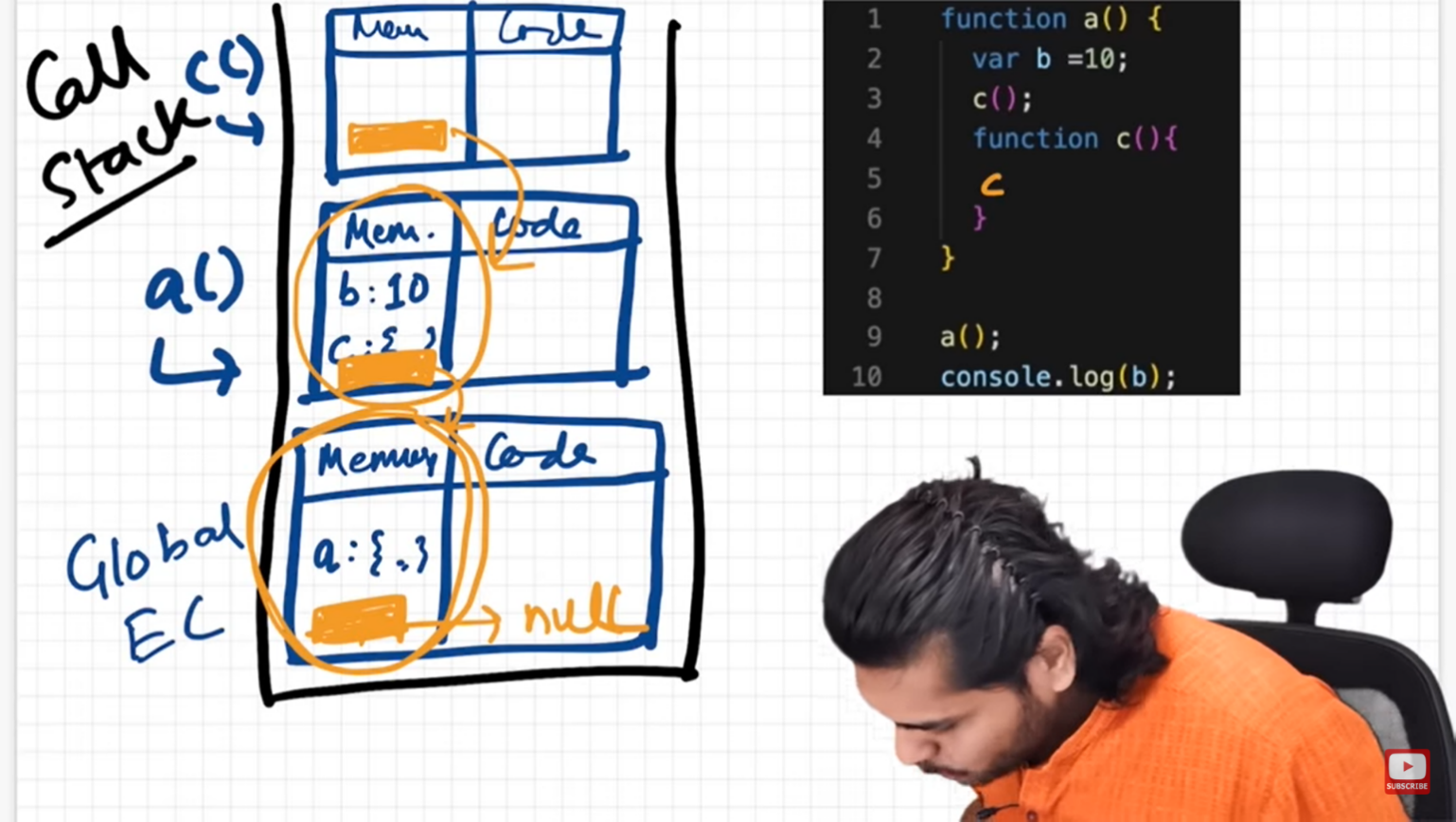
# Scope | Lexical Environment | Scope Chain

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 global execution context is referring to the null

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

# Temporal Dead Zone | let & const

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

But they are in temporal dead zone.

***Temporal dead zone (TDZ) is the time between let and const is hosted and till it is initialized with 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 (block 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.

An inner function always has access to the variables & parameters of its outer function.

Even after the outer function has returned.

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 generatedA 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 automatically garbage collected till the program expires

Currying using bind

A screenshot of a computer

Description automatically generated

Currying using closure

A screenshot of a computer

Description automatically generated

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

Callback

Functions are first-class citizen.

A callback is a function passed as an argument to another function

This technique allows a function to call another function

A callback function can run after another function has finished.

Asynchronous programming happens due to callback functions.

Asynchronous operation in JS happens due to callback.

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

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

Problems with Callback

**Callback Hell**

**Solution- Promise Chaining**

Code starts growing horizontally instead of vertically also known as pyramid of DOM. Because of nesting of callbacks. (Callbacks inside callback)

**Inversion of control**

**Solution- Promise**

Losing control of code. As we pass callback function to other function, and we had provided control of callback function to other function.

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 API

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

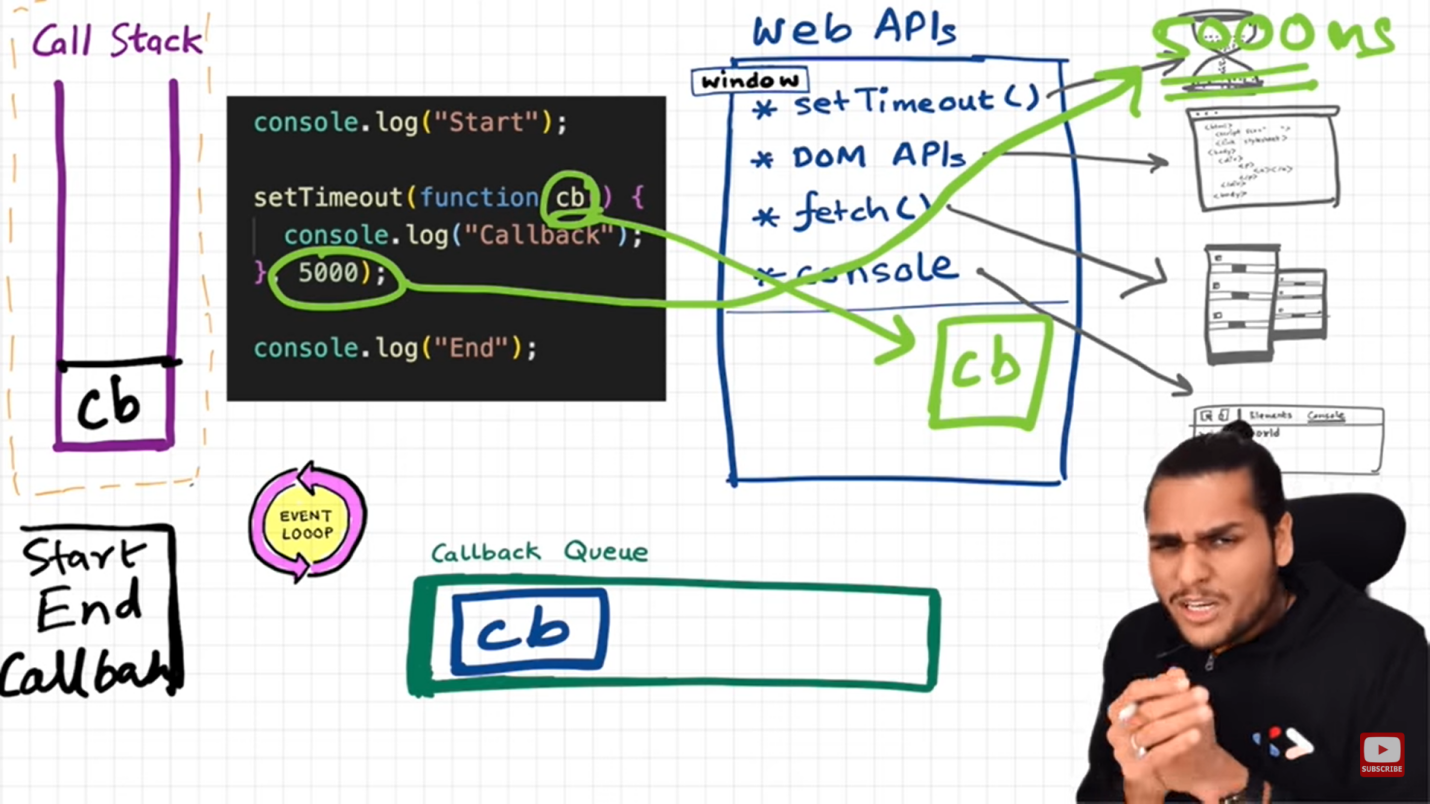
Browser gives access of Web API in call stack (i.e., JavaScript engine) using global object (window) to use all the feature of Web API.

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. If call stack is not empty, then callback function will wait inside callback Queue waiting for call stack to get empty.

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.**

How JavaScript run inside JS Engine

1. The Script gets loaded as a UTF-16-byte stream from either the network/ server, cache, or a service worker and passed to a byte stream decoder.
2. The **byte stream decoder decodes the bytes into tokens**. The tokens are sent to the parser.
3. The **parser** generates nodes based on the tokens and **create an AST** (Abstract Syntax Tree).
4. The **interpreter** walks through the AST and **generate byte code**.
5. The byte code and type feedback are sent to the **optimizing compiler**, which generates highly optimized machine code.

***Reference*** [How JavaScript Works ? | Javascript - Lecture 3 | Web Development Course - YouTube](https://www.youtube.com/watch?v=2lRQTdpwhFk)

Diagram

Description automatically generated

Map/ Filter and reduce

Higher Order function in JavaScript

**Chaining**

const user = [1, 2, 3, 4, 56, 5];

const multipleOddBy2 = user.filter(i => i%2).map(i => i\*2)

console.log(multipleOddBy2);

Debouncing

If the difference between two key pressed events has time greater than certain time, then make an API call.

Calling API when pause.

Throttling

If the difference between two API call has time greater than certain time, then make an API call

Calling API after some time interval.

***JSON.parse()***

* Used to receive data from a webserver
* To convert text into JS object

***JSON.stringify()***

* Used to send data to a webserver
* Convert JS object into string

# Promises

Promise is like container to store the future value.

"Producing code" is code that can take some time

"Consuming code" is code that must wait for the result

A Promise is a JavaScript object that links producing code and consuming code

|  |  |
| --- | --- |
| ***myPromise.state*** | ***myPromise.result*** |
| "pending" | undefined |
| "fulfilled" | a result value |
| "rejected" | an error object |

Promises is used to handled async operations in JavaScript

With callback we pass a function to other function

With promises we attach a function to promise object.

Promise object are immutable.

A Promise is an object representing the eventual completion or failure of asynchronous operation.

***// Creating Promise***

const p = new Promise((resolve, reject) => {

if(true){

resolve('Saranj');

}else{

reject('Bule');

}

});

***// Consuming Promise***

p.then(data => console.log(data)); // true => Saranj

p.catch(err => console.log(err)); // !true => Bule

Prototype

* Inheritance is possible due to prototype property
* The prototype property of an object is where we put methods & properties that we wanted another object to inherit.
* The constructor prototype is the prototype of all instances that are created through it
* Use prototype to inheritance