JavaScript Hoisting:

1. Variable Hoisting:

```
console.log(x); // Error: Cannot access 'x' before initialization
let x = 10;
```

In JavaScript, variables declared with `let` and `const` are hoisted to the top of their scope but not initialized. This is why trying to access them before the declaration results in an error.

2. Function Hoisting:

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

Functions, on the other hand, are fully hoisted, meaning you can use them before the declaration.

3. Arrow Functions and Variables:

```
getName2(); // Error: Cannot access 'getName2' before initialization
console.log(z); // Error: Cannot access 'z' before initialization
const getName2 = () => {
  console.log("Hello Anup");
};
let z = 10;
```

Arrow functions, being similar to variables, also exhibit hoisting behavior. Trying to access them before initialization results in an error.

Execution Context and Scope:

4. Function Scoped Variables:

```
let x = 10;
a(); // 100
b(); // 200
```

```
console.log(x); // 10
c(); // 300

function a() {
  var x = 100;
  console.log(x); // x is function scoped here
}

function b() {
  var x = 200;
  console.log(x); // x is function scoped here
}

function c() {
  var x = 300;
  console.log(x); // x is function scoped here
}
```

JavaScript has function-scoped variables. The variable `x` inside each function is local to that function.

5. Global Object and "this":

```
console.log(this === window); // true in the browser
let a = 10;

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

console.log(window.a); // 10
console.log(a); // 10
console.log(this.a); // 10
console.log(x); // Error: x is not defined
```

In a browser, the global object is `window`. The `this` keyword refers to the global object, and if a variable is not found in the local scope, JavaScript looks for it in the global scope.

Variable Declaration and Initialization:

6. Variable Declaration and Initialization:

```
console.log(a); // undefined
var a = 10;
console.log(a); // 10
console.log(b); // undefined
var b;
```

JavaScript is loosely typed, meaning variables can change types. If a variable is declared but not initialized, it's `undefined`.

```
var x;
console.log(x); // undefined
x = 10;
console.log(x); // 10
x = "Hi, this is x";
console.log(x); // Hi, this is x
```

```
x = true;
console.log(x); // true
```

Variables can change types dynamically, showcasing JavaScript's weakly typed nature.

Certainly! Below is a README file with your provided comments elaborated and organized:

Lexical Environment:

In JavaScript, a Lexical Environment represents the local memory and its environment hierarchy. When a Global Execution Context is created, it comes with a Lexical Environment, establishing a structured hierarchy.

Hoisting:

Variable Hoisting:

```
console.log(x); // Error: Cannot access 'x' before initialization
let x = 10;
```

In JavaScript, variables declared with `let` and `const` are hoisted to the top of their scope but not initialized, resulting in an error when accessed before declaration.

Function Hoisting:

```
console.log(this === window); // true in the browser
let a = 10;

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

console.log(window.a); // 10
console.log(a); // 10
console.log(this.a); // 10
console.log(x); // Error: x is not defined
```

Functions are fully hoisted, allowing them to be used before declaration. The `this` keyword in a browser environment points to the global object (`window`).

Arrow Functions and Variables:

```
getName2(); // Error: Cannot access 'getName2' before initialization
console.log(z); // Error: Cannot access 'z' before initialization
const getName2 = () => {
  console.log("Hello Anup");
};
let z = 10;
```

Arrow functions, being similar to variables, also exhibit hoisting behavior, resulting in an error if accessed before initialization.

Execution Context and Scope:

Function Scoped Variables:

```
let x = 10;
a(); // 100
b(); // 200
console.log(x); // 10
c(); // 300

function a() {
  var x = 100;
  console.log(x); // x is function scoped here
}

function b() {
  var x = 200;
  console.log(x); // x is function scoped here
}

function c() {
  var x = 300;
  console.log(x); // x is function scoped here
}
```

JavaScript has function-scoped variables. The variable `x` inside each function is local to that function.

Variable Declaration and Initialization:

```
console.log(a); // undefined
var a = 10;
console.log(a); // 10
console.log(b); // undefined
var b;
```

JavaScript is loosely typed, allowing variables to change types. If a variable is declared but not initialized, it's `undefined`.

```
var x;
console.log(x); // undefined
x = 10;
console.log(x); // 10
x = "Hi, this is x";
console.log(x); // Hi, this is x
x = true;
console.log(x); // true
```

Variables can dynamically change types in JavaScript, showcasing its weakly typed nature.

Temporal Dead Zone (TDZ):

```
// let, const are not hoisted, this is not right. We can say they are in Temporal Dead

// console.log(a);   // Uncaught ReferenceError: Cannot access 'a' before initializati
let a = 10;
```

Variables declared with `let` and `const` are not hoisted in the traditional sense. They enter a Temporal Dead Zone where accessing them before declaration results in an error.

Duplicate Variable Declaration:

```
// syntax error: we cannot declare two values with the same variable name with let or
let x = 7;
let x = 10;
```

Declaring two variables with the same name using `let` or `const` results in a syntax error.

Variable Reassignment:

```
// this is valid
let y;
y = 12;
```

Reassigning a value to a declared variable is valid in JavaScript.

Const Declaration:

```
const t;
t = 100;
// syntax error: Missing initializer in const declaration
```

Declaring a constant variable without an initializer results in a syntax error.

Syntax Errors:

```
console.log("hello anup");

let x = 7;
let x = 10;
// It'll not execute the log!!! It'll simply throw a syntax error.
```

Syntax errors in JavaScript prevent the execution of the code.

```
const s = 1000;
s = 20;
// Type Error: Assignment to constant variable
```

Assigning a new value to a constant variable results in a Type Error. Constants cannot be reassigned.

Block Statements and Scope:

1. Block Statements:


```
  // we group multiple statements in a block to use it where JS expects one statement
  const a = 10;
  console.log(a);
}

if (true) {
  // This is a block where we group const and log statements for use in an if statemen
  const a = 10;
  console.log(a);
}
```

Blocks in JavaScript are used to group multiple statements, making them a single statement for contextual usage.

2. Scope Behavior of `let`, `var`, `const`:

```
{
  var x = 10; // Global scoped, can be accessed anywhere
  let y = 20; // Block scoped, cannot be accessed outside
  const z = 30; // Block scoped, cannot be accessed outside
}

console.log(x); // 10
console.log(y); // Reference error: y is not defined
console.log(z); // Reference error: z is not defined
```

`var` is not block-scoped and can be accessed globally. `let` and `const` are block-scoped, limiting their accessibility.

3. Shadowing in JavaScript:

```
var d = 10;

{
  var d = 100;
  console.log(d); // 100, var d in the block shadows the global var d
}

console.log(d); // 100, global var d gets shadowed by block-scoped d
```

`var` is not block-scoped; hence, it can be shadowed within a block, affecting the global variable.

4. Script Scope vs. Block Scope:

```
let x = 10; // Script scoped
const y = 10; // Script scoped

{
```

```
  let x = 20; // Block scoped
  const y = 20; // Block scoped
  console.log(x); // 20
  console.log(y); // 20
}

console.log(x); // 10
console.log(y); // 10
```

`let` and `const` have block scope, while `var` would have script (or function) scope.

5. Illegal Shadowing:

```
// let a = 10  // This will throw an error; it's called illegal shadowing
// {
//     var a = 10
// }
```

Attempting to declare a variable with `let` in a block where it was already declared with `var` results in an error.

6. Function Scoping:

```
let b = 20;

function x() {
  var b = 100; // No error; var is function-scoped
}
```

7. Lexical Scoping and Closures:

```
const a = 20;
{
  const a = 30;
  {
    const a = 40;
    console.log(a); // Lexical scoped, prints the nearest "a" value: 40
  }
  console.log(a); // Prints 30; the nearest value is 30
}
```

Lexical scoping keeps track of variables based on their nesting levels, forming a closure.

8. Closures:

```
function x() {
  var a = 10;
  function y() {
    console.log(a);
  }
  return y; // Function x() returns function y()
}

var z = x();
```

```
console.log(z); // Prints the function definition of y
z(); // 10  // Invoking y() where it's stored, remembering its lexical scope reference
```

Uses of closures include Module Design Pattern, Currying, Functions like once, memoize, Maintaining state in Async world, setTimeout(), and iterators.

9. Loop Issue and Solution:

```
function x() {
  for (var i = 1; i <= 5; i++) {
    setTimeout(() => {
      console.log(i);
    }, i * 1000);
  }

  console.log("Hello Anup");
}

x();
```

In this code, `var` causes an issue as it is globally scoped. It prints 6 for every iteration.

10. Fixing Loop Issue with `let`:

```
function y() {
  for (let i = 1; i <= 5; i++) {
    setTimeout(() => {
      console.log(i);
    }, i * 1000);
  }

  console.log("Hello Anup");
}

y();
```

Using `let` solves the loop issue by creating a new lexical scope for each iteration.

11. Fixing Loop Issue with `var` and Closure:

```
function z() {
  for (let i = 1; i <= 5; i++) {
    function closure(i) {
      setTimeout(() => {
        console.log(i);
      }, i * 1000);
    }
    closure(i); // Creates a new closure with a new copy of i
  }

  console.log("Hello Anup");
}

z();
```

Creating a closure for each iteration solves the loop issue with `var`. Each closure gets a new copy of `i`.

Function Statement vs. Function Expression:

1. Function Statement:

```
// Function statement can be hoisted
function a() {
  console.log("a called");
}
a();
```

Function statements can be hoisted, allowing them to be called before the declaration in the code.

2. Function Expression:

```
// Function expression cannot be hoisted
const b = function () {
  console.log("b called");
};
b();
```

Function expressions, stored in variables, cannot be hoisted. They need to be declared before usage.

3. Function Declaration:

```
// Function declaration and statement are the same thing
function a() {
  console.log("a called");
}
a();
```

Function declaration and statement are interchangeable; both can be hoisted.

4. Anonymous Function:

```
// Anonymous function doesn't have a name or identity
// Used in places where functions are used as values
// Example: callback functions
function() {
  //...
}
```

Anonymous functions are often used as values, such as in callback functions.

5. Named Function Expression:

```
const c = function xyz() {
  console.log("c called");
};
c(); // c called
xyz(); // Reference error; xyz is in the local scope of c
```

Named function expressions have a name, but the name is only accessible within the function itself.

6. Parameters vs. Arguments:

```
const d = function xyz(param1, param2) {
  console.log(param1 + param2);
};
d(1, 2); // Arguments: 1 and 2
```

Parameters are placeholders in the function definition, and arguments are the values passed during the function call.

First Class Functions:

7. First Class Functions:

```
const e = function xyz(param1) {
  console.log(param1);
};

function x() {
  console.log("x is called");
}

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

JavaScript treats functions as first-class citizens, allowing them to be used as values, parameters, and returned in other functions.

Callback Functions:

8. Callback Functions:

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

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

setTimeout(() => {
  console.log("timer");
}, 5000);

document.getElementById("clickMe").addEventListener("click", () => {
  console.log("button clicked");
});
```

Callback functions are functions passed as arguments to other functions. They enable asynchronous behavior in synchronous code.

JavaScript Single-Threaded Nature:

9. JavaScript is Single-Threaded:

- JavaScript is a single-threaded and synchronous language.
- Blocking the main thread can lead to performance issues.

Event Listeners and Closures:

10. Event Listeners and Closures:

- Deep dive into event listeners and closures.
- Understanding scope and closures in the context of event listeners.

Garbage Collection and Remove Event Listeners:

11. Garbage Collection and Remove Event Listeners:

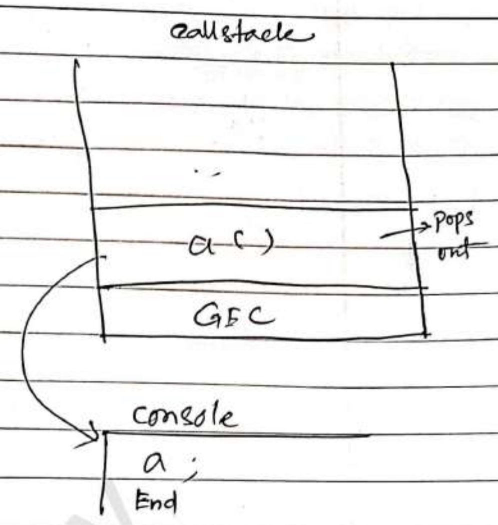
- Adding event listeners is a heavy operation that uses memory.
- Exploring the importance of removing event listeners to avoid memory leaks.

Event Loop

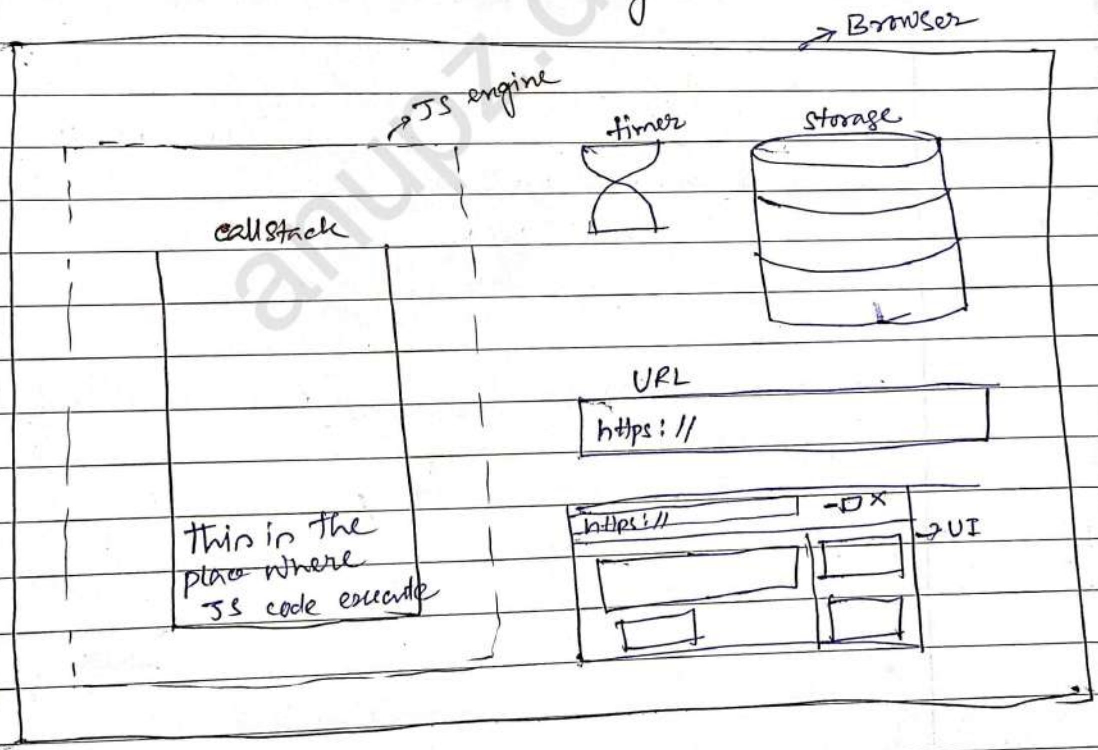
Page-1

Event Loop

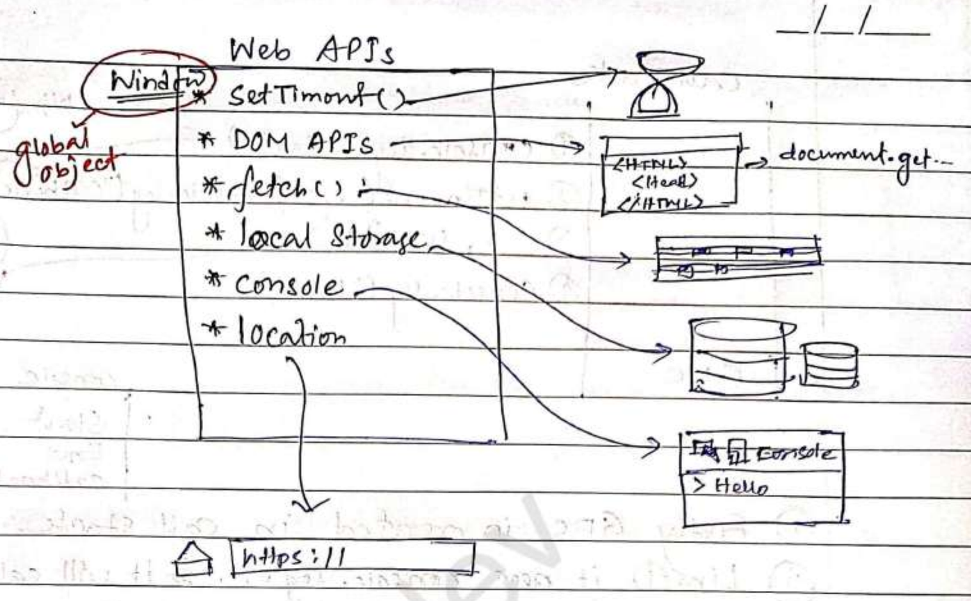
```
function a() {
    console.log("a");
}
a();
console.log("End")
```



call stack is the place where JS codes are executed.
call stack is inside the JS engine.



To connect all these features, the V8 JS engine need ~~to~~ something. That is Web APIs

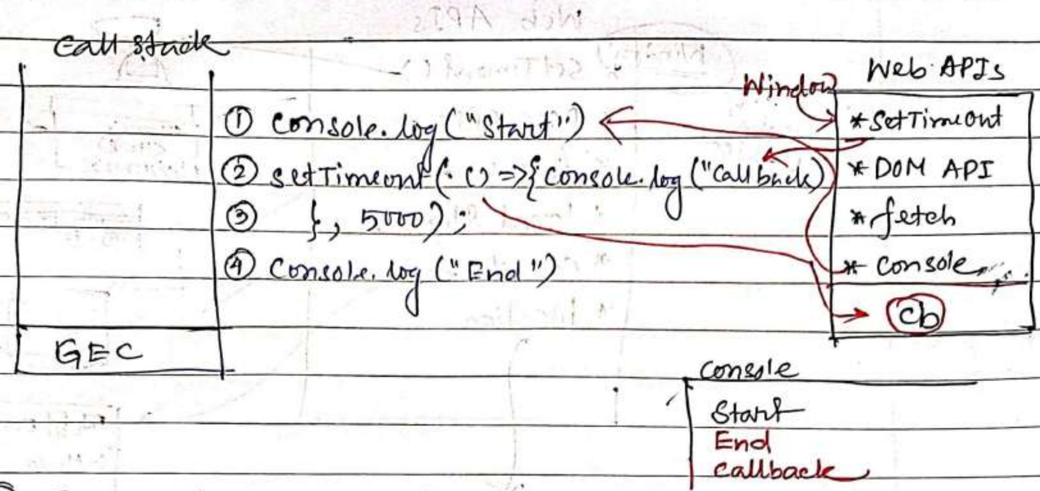


All the functionalities are not part of JavaScript.
These powers gives us by the Browser.

To access all the Web APIs, We can do →
window.setTimout
window.localStorage
⋮

Whenever we need all these Web APIs, the come to the Call stack / JS V8 engine.

_/_/__



① Firstly GEC is created in call stack
(ii) Line ① it sees console.log(), it will call the web Api console and print 'start' in console.
(iii) Line ② it sees setTimeout & call the Web Api to get the setTimeout.
(iv) Line ② it sees a callback, it register the cb in the web Api
(v) Line ③, it sees there's time, it sets the timer and go to the next line
(vi) Line ④, it ~~just~~ call console web api & ~~is~~ simply logs "End"

(vii) After that GEC will be popped out of the call Stack

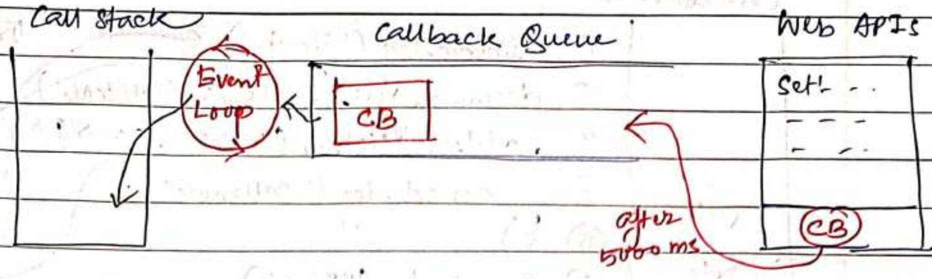
(viii) But the timer is still running.

(ix) As soon as 5000 ms is passed the call back function ~~will be~~ needs to put into the call stack

But How this CB function will go to the

call stack ??

__/__/__



(X) So, after 5000 ms ends, the CB fn will go through the call back Queue.

(XI) After 5000 ms, CB will be send to the callback Queue.

(XII) Then the Event loop will act as a gate keeper. The Event loop will check whether there's anything inside the callback Queue or not.

(XIII) As soon as the Event loop finds out there's CB in the CallBack Queue, it will send the CB to the Call Stack.

(IX) And the call stack will Quickly executes the call back function.

Create a GEC in callstack

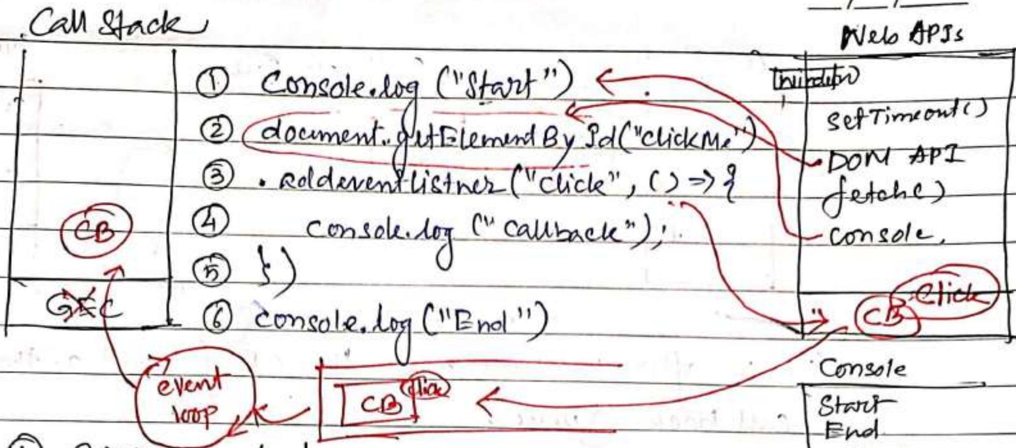
↓

Execute the code line by line

↓

After Execution GEC deleted/popped out

Event loop has one task → to constantly monitor callback Queue & Call Stack



(i) GEC created

(ii) Access the Web API → Console → Print "Start"

(iii) Line ② → Access the DOM API, it will basically goes to the DOM element & get the button id named "clickMe".

(iv) Line ③ → A call back will be registered in the Web API. And with the call back the "click" event will be attached to it. and it moves ~~no~~ on to the next line.

(v) Line ⑥ → same as step (ii)

(vi) GEC is popped out of Call Stack.

(vii) The callback wait in the web APIs for the Button click.

(viii) When the Button clicked, the CB + "click" will be pushed in the Callback Queue.

(ix) Then the Event loop ~~find the~~ will check if the call Stack is empty or not. If its empty it'll take the CB & push to the Call stack.

(x) And this CB quickly executed.

(xi) ~~Then the G~~ Prints the "callback" in Console.

(xii) CB will be popped off from the Call Stack.

(*) (*) But why do we need the Callback Queue?

Ans: So, there's multiple eventListeners in the webpage. If the user clicks, scroll or anything, that will be added to the Queue. To maintain the order of execution of each CB, we need callback Queue.

&

Event loop will constantly check the call stack is empty or not & callback Queue has functions or not. So, if ~~its~~ ~~cm~~ Stack is empty, it will send the CB from the Queue one by one.

Fetch

WebAPIs

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

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

console.log("End")
```

CBT
CBN
GEC

window
* SetTimeOut()
* DOM API
* fetch()
* Console

5000 ms

CBTimeout 5000 ms
CBN 50ms

Console
Start
End
CB Netflix
CB Timeout

② ①

Event loop

CB Queue

CBT

Microtask Queue

CBN

Higher Priority

(I) CBT is waiting for the timer 5000 ms to end.

(II) CBN is waiting to get the data from NF server → Once it get the data from the server, CBN() is ready to be executed.

(III) There also ~~Micro Task Queue~~, Function which ~~will be~~ is in the Microtask Queue, will be executed first, But ~~of~~ callback Queue later.

(IV) So, the Event loop will check if the Call stack is Empty or not. And Event loop will check the Microtask Queue & the send the CBN function to the Call stack. & executes

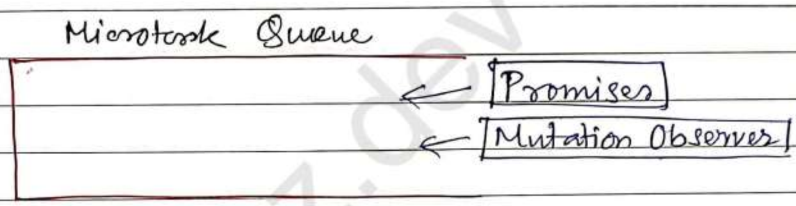
(V) Then send the CBT function from CB Queue.

- What will be in the Microtask Queue?

⇒ All the codes which ~~to~~ have Promises, will be pushed into the Microtask Queue.

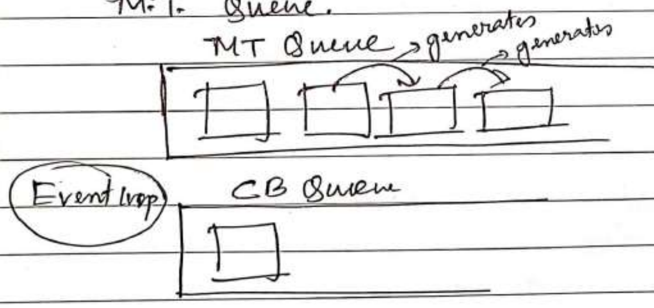
Mutation observer ⇒ Its something which keep looking for if there's any changes in the DOM or not.

So, DOM changing ~~functi~~ CB functions also send into the Microtask Queue.



- Starvation of Callback Queue.

⇒ So, ~~At~~ In the Microtask Queue, there may be some functions which will create more functions in the ~~Fu~~ M.T. Queue.



So, in this case the function in CB Queue never ~~ex~~ get a chance to execute. Because ~~ev~~ M.T. Queue have higher priority.

This is called Starvation of CB Queue.

1. Higher-Order Functions and Callbacks

Code Example:

```
// Callback function
function x() {
  console.log("hi");
}

// Higher-order function
function y(callback) {
  callback();
}
```

Explanation:

- **Higher-Order Functions:** Functions that take other functions as arguments or return functions.
- **Callback Functions:** Functions passed as arguments to other functions.

2. Area, Circumference, and Diameter Calculations

Code Example:

```
const radius = [2, 4, 5, 6];

const areaFormula = (r) => 4 * 3.14 * r * r;
const circumferenceFormula = (r) => 2 * 3.14 * r;
const diameterFormula = (r) => 2 * r;

// Using higher-order function to calculate values
const calculateValues = (radiusArray, logicFunction) => {
  return radiusArray.map((r) => logicFunction(r));
};

let area = calculateValues(radius, areaFormula);
let circumference = calculateValues(radius, circumferenceFormula);
let diameter = calculateValues(radius, diameterFormula);
```

Explanation:

- Demonstrates higher-order functions and callback functions for calculating circle properties.

3. Map Method

Code Example:

```
const arr = [1, 2, 3, 4, 5];

const double = (element) => element * 2;
const triple = (element) => element * 3;

const outputDouble = arr.map(double);
const outputTriple = arr.map(triple);
```

Explanation:

- Utilizes the `map` method to transform array elements based on provided functions.

4. Filter Method

Code Example:

```
const array = [12, 34, 55, 87, 56, 43];

const oddArray = array.filter((element) => element % 2 !== 0);
const evenArray = array.filter((element) => element % 2 === 0);
const greaterThanThirty = array.filter((element) => element > 30);
```

Explanation:

- Uses the `filter` method to create new arrays with specific conditions.

5. Reduce Method

Code Example:

```
const list = [12, 34, 55, 87, 56, 43];

const sumOfList = list.reduce(
  (accumulator, current) => accumulator + current,
  0
);
const maxVal = list.reduce(
  (max, current) => (current > max ? current : max),
  0
);
```

Explanation:

- Demonstrates the `reduce` method for aggregating values in an array.

6. User Data Operations

Code Example:

```
const users = [
  { firstName: "Anup", lastName: "Jana", gender: "male", age: 25 },
  { firstName: "Arpita", lastName: "Biswas", gender: "female", age: 25 },
  { firstName: "Swarup", lastName: "Jana", gender: "male", age: 22 },
  { firstName: "Gargi", lastName: "Mosha", gender: "female", age: 17 },
  { firstName: "Irani", lastName: "Chowdhury", gender: "female", age: 17 },
];

// * get the full name of every user

const fullName = users.map((user) => user.firstName + " " + user.lastName);
console.log(fullName);

// // get user firstName who have age below 20

const belowTwentyUsers = users
  .filter((user) => user.age < 20)
```

```
    .map((user) => user.firstName);

  console.log(belowTwentyUsers);

  // // get the count of ages {17:2, 25:2, 22:1}

  const countAges = users.reduce((acc, curr) => {
    if (acc[curr.age]) {
      acc[curr.age] += 1;
    } else {
      acc[curr.age] = 1;
    }
    return acc;
  }, {});

  console.log(countAges);

  // // Get the user first names who have the same last name

  const sameLastNameUsers = users.reduce((acc, curr) => {
    const existingUser = acc.find((user) => user.lastName === curr.lastName);
    if (existingUser) {
      existingUser.firstNames.push(curr.firstName);
    } else {
      acc.push({ lastName: curr.lastName, firstNames: [curr.firstName] });
    }
    return acc;
  }, []);
  console.log("Users with the Same Last Name:", sameLastNameUsers);

  // // Sort the users by age

  const sortedUsersByAge = users.sort((a, b) => a.age - b.age);
  console.log("Sorted Users by Age:", sortedUsersByAge);

  // // Get the user who has the same number of letters in the first name

  const sameLetterCountUsers = users.filter((user) => {
    const firstNameLength = user.firstName.length;
    return users.some(
      (u) => u.firstName.length === firstNameLength && u !== user
    );
  });
  console.log(
    "Users with the Same Letter Count in First Name:",
    sameLetterCountUsers
  );

  // // Get the users separated by gender

  const maleUsers = users.filter((user) => user.gender === "male");
  const femaleUsers = users.filter((user) => user.gender === "female");
  console.log("Male Users:", maleUsers);
  console.log("Female Users:", femaleUsers);
```

Feel free to explore and modify the code for your own learning and understanding of higher-order functions and array methods in JavaScript.

Feel free to use this README for your documentation!