beforeAll() = runs code before All tests

afterAll() = runs code after All tests.

---

(16)

Recap Part 2

Callback Function :-

(*) When we pass Functions as the value of the Parameter is called Callback Function.

eg

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

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

→ Calling the function in Parameter.

async

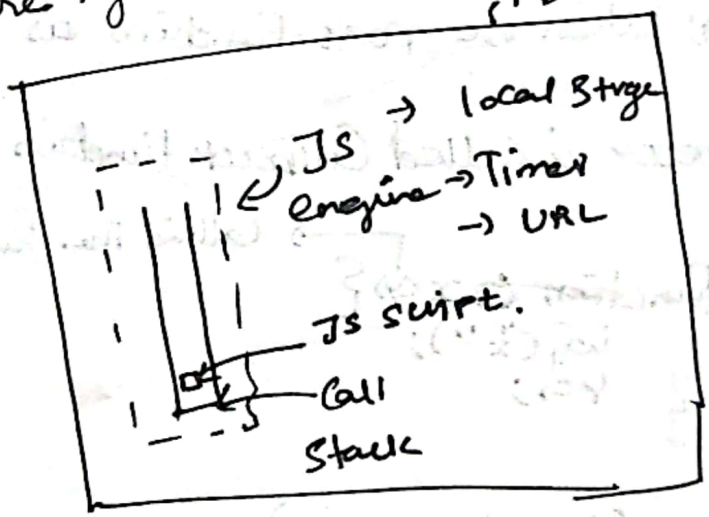
```
setTimeout(function(){
    log("timer");
}, 5000);
```

(*) If any operations blocks the call stacks means
lock "main thread".

(*) Whenever, call back functions is called, it creates
opes of its parents (closure) and Global.

Event listeners

(*) It is always heavy in memory as it remembers
its closure for future reference calls.



(*) We need webAPI In order to make connection with the
Browser.

WebAPI's (Global object of webAPI => window)

(*) setTimeout() (*) Local Storage
(*) DOM API's (*) Console.
(*) fetch() (*) location.

setTimeout
Console
Console
When
Timeout
Even
(*) After
setTimeo
It will
(*) If
Event Loo
Event
and
will

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

function cb() Stores in webAPI environment

* When the code reach SetTimeout it will run the Timer feature through the SetTimeout API to the given milliseconds.

(*) After the Timer end, The function Inside the SetTimeout will move from webAPI to Callback Queue and It will check by Event Loop.

(*) If any value present in Callback Queue, the EventLoop will Push the value to the Call Stack.

Event Loop :

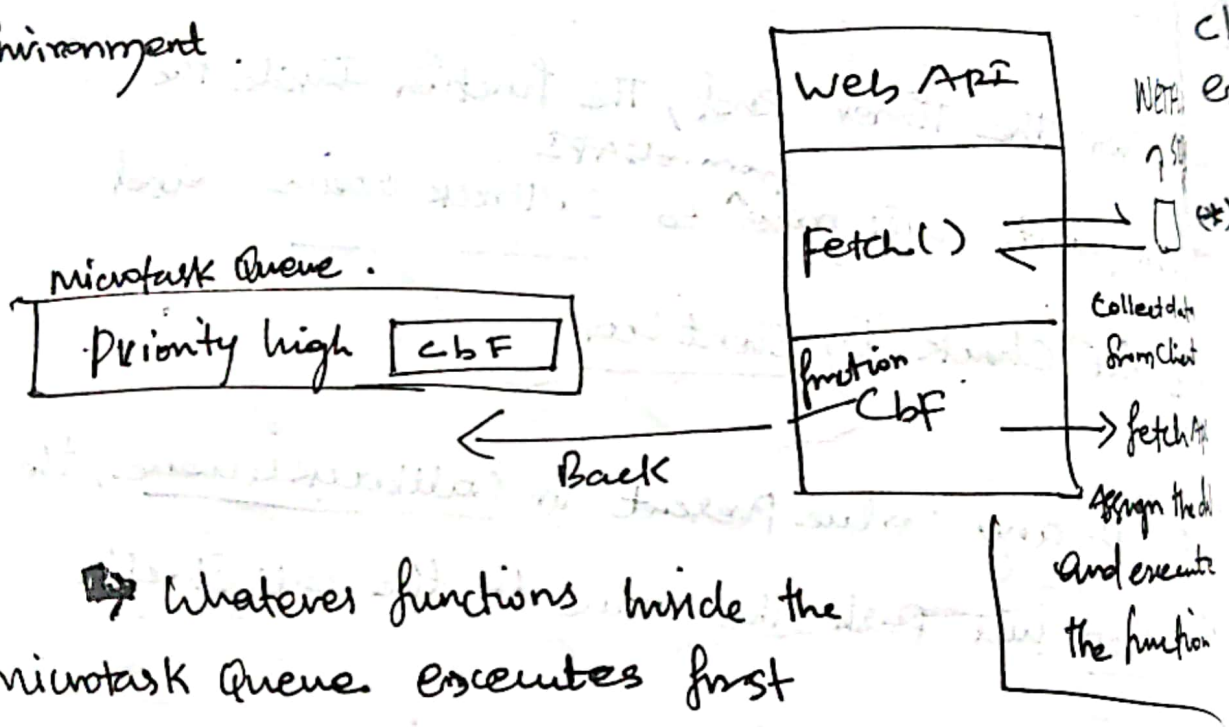
(*) Event loop continously checks the Call Stack. and the Callback Queue.

(i) If the Call Stack is empty. the event loop

(ii) if the Call Stack is not empty, the event loop waits until it becomes empty before moving the next task from the Queue to the Stack.

fetch WebApi

(*) Whenever using fetch, it will get the data from the Server and stores with functions (register) in the Web Environment.



➡ Whatever functions inside the microtask Queue executes first

↳ Call back Queue waits and execute next.

↳ After Compiler finish executing whole code GEC will be empty then the microtask Queue functions will be in Global Execution Context.

↳ After Executing the microtask Queue, the Callback Queue will be in Call Stack and

Here, cbF is function of fetch

And cbT is the of SetTimeout

After if GEC microtask Qu cbf to the execute

(*) When the

Scanned with CamScanner

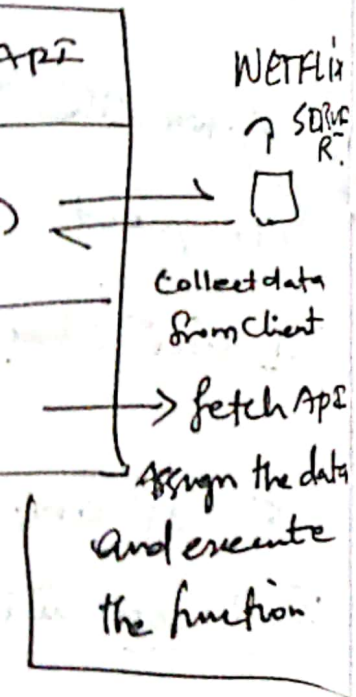
the Event loop

oing to the

Stack.

t The data from

) In the WebAPI



te next.

whole code

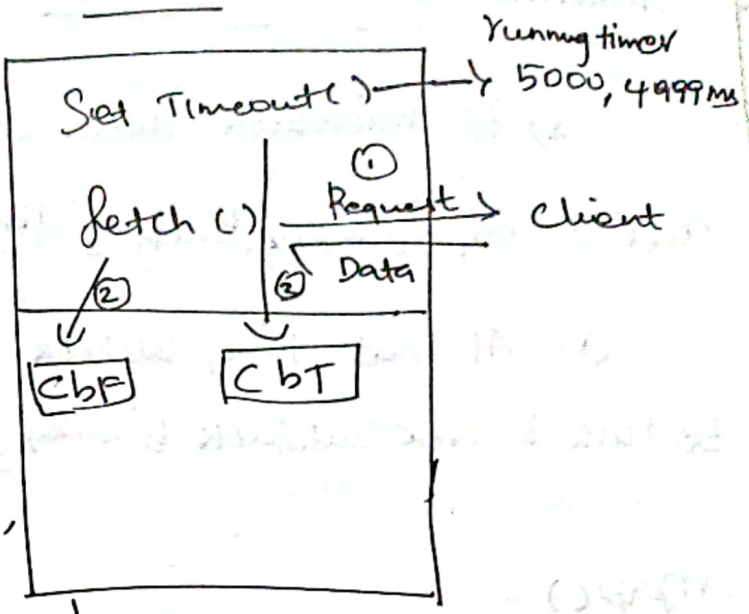
ueue functions

Queue, so

and exec

Here, CbF is the function of fetch().

And CbT is the function of SetTimeout.



(*) After it (Event loop check) CbF is empty, microtask Queue push CbF to the Call Stack and execute

CbF will be in microtask Que

CbT will be in Callback Queu

(*) When the microtask is empty, The Callback Queue ~~function Queue~~ are pushed to Call Stack.

microtask Queue

- → Promises
- → Mutuation Observes.

(*) In General, Microtask Queue contains promises and mutuation Observe

(*) It is prority high Queu

Call back Queue

- → SetTimout()
- → DomAPI
- → Console.

(*) Priority less

(*) Other name "Task Qu

STARVATION OF TASK INSIDE CALLBACK QUEUE.

(*) if microtask Queue creates another function and so on, we may plenty of pending in the Microtask Q

(*) At that time, callback Queue never get chance to Push to GEC Call Stack is called Starvation

.map()

(*) It creates a new array by applying a provided function to each element of the original Array.

(*) It doesnot modify the original array and returns a new Array with the results of the function applied to each element.

Syntax:-

```
array.map function ( currentValue, index, array)
    // return transformed value.
});
```

eg:- const num = [1, 2, 3, 4, 5];

const doubled = num.map((num) => num * 2);

console.log(doubled);

output:
[2, 4, 6, 8, 10]

.filter();

(*) Used to filter the value inside the Array.

eg:

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

const output = arr.filter(isOdd);

function isOdd(x){
    return x%2;
}
```

.reduce() :-

(*) Higher Order function that reduces an array to a single value by applying a function to each element of the array.

eg:

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

const output = arr.reduce(function(acc, curr){
    acc = acc + curr;
    return acc;
}, 0);
```

To find max by using reduce :-

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

Const output = arr.reduce(function (max, curs){

    if (curs > max){

        max = curs;

    }

    return max; },0);

Console.log(output); => 6.
```

another eg :-

```
Const users =[

{firstname :"akshay", lastname: "saini", age: 26},

{firstname : "donald", lastname : "trump", age : 75},

{ firstname : "elon", lastname : "musk", age: 50},

{firstname : "Jaimin", lastname : "Athath", age : 10}];
```

```
const output = Users.filter((x) => x.age<30).
                .map((x) => x.Firstname);

console.log(output);
```

(*) above map works based on the output of the filter here.

Challenge

```
const output = users.reduce(function(acc,curr){
    if(acc[curr.age] < 30){
        acc.push(curr.Firstname);}
    return acc;
},[]);

console.log(Output);
```

Callback Hell:

(*) Using Callback Function Inside callback function is called Callback Hell.

(*) Code grows horizontally.

(*) Its unreadable & unmaintainable.

CH2 Inversion of Control:-

0. Important of Callbacks
1. Issues with Callbacks
   a. Callback Hell.
   b. Inversion of Control.

Inversion of Control :-

(*) losing control cause of passing callback Function to the another function.

(*) we couldn't confirm whether the function will execute or not.

(4) Promises :-

(*) Its an Object that represent the eventual completion of an asynchronous operation and its resulting value.

(*) Promises are used to handle asynchronous code in a more manageable way compared to traditional callbacks, allowing you to write cleaner and more readable code.

Before promises :-

```
CreateOrder(cart, function(orderId){
    ProceedToPayment(orderId);
```

5. Eg:- Const Cart = ["shoes", "pants", "kurta"];

Promises

Const Promise = CreateOrder(Cart);

Promise.then(function(orderId){

Proceed to payment(orderId);

});

Promises initally = Pending

After some time = Fulfilled.

State of Promises

1) Pending
2) fulfilled
3) Rejected.

(*) Promises Objects are Immutable.

Definition 1 :-

(*) Promise Objects are Placeholder which will filled with value later.

Definition 2 :-

(*) Promise Objects are Placeholders for certain period of time until fulfilled with Asynchronous Operation.

(*) It's a Container

Promises Chaining :-

(*) Return a Promises in Promises when chaining

eg: CreateOrder(cart)

.then(function(orderId){

return ProceedToPayment(orderId);

})

.then(function(paymentInfo){

return ShowOrderSummary(paymentInfo);

});

(or) Using arrow function.

Creat Order(cart)

.then(orderId => ProceedToPayment(orderId));

.then(paymentInfo => ShowOrderSummary(paymentInfo));

By Using Promises Chaining, we will avoid Callback He

Creating an promises :-

```
const cart = [ "shoes", "Pants", "Kurta" ];

const promise = createOrder (cart);

promise.then(function () {

  Proceed to payment (orderId);

})
```

creating

```
function createOrder (cart) {

const pr = new promise (function (resolve, reject ) {

  if( !validateCart (cart)){

  const err = new Error ("Cart is not valid");

  reject (err);

  }

const orderId = "12345";

if (orderId) {

  resolve (orderId);

});

  return pr;
```

if we get reject in Promises :-

Use,

```
Promise
  .catch( function (err) {
  Console.log (err.message);
  });
```

```
Create order(Cart)
  .then (function (orderId) {
  Console.log (orderId);
  return orderId;
  })
  .catch (function (err) {
  Console.log(err.message);
  .then (function (orderId) {
  return ProceedToPayment (orderId);
  })
  .then (function (paymentInfo) {
  Console.log (
```

.catch → Catch will consider the error Before the Catch Statement.

Once Catch is used above onto another function, It will called and Catch won't handle

Eg:-

```
CreateOrder(Cart)
  .then(function(orderId){
    console.log(orderId);
    return orderId;
  })
  .catch(func.(err){
    console.log(err.message);
  })
  .then(function(orderId){
    console.log("no matter");
  })
```

Here Catch will handle the error only for above then function.

Then will surely called regardless of the error.

(16) Async functions :-

(*) Use "Async" keyword before Functions Creations.

<< It always returns a promise.

Eg: async function getData(){

return new Promise(function(resolve, reject){

resolve("Hello World");

});

}

(or)

asyn function getData(){

return "namaste"; → Here, the value which we return wraps as promise when we return.

}

① const data = getData(); → promises

console.log(data); → console Promises

② datapromises.then((res) => console.log(res));

↳ console :

namaste.

Eg:2

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

resolve("promise Resolved Value!!");

});

```
    return p;
}
const datapromise = getData();
datapromise.then((res) => console.log(res));
```

output:-

Promise Resolved value !!

Awaits :-

(*) Async and Awaits are used to handle Promise.

(*) Await is a keyword that only can be used Inside the Async function.

(*) Write await infront of the promise that is resolved.

Eg
```
const p = new Promise((resolve, reject) => {
    setTimeout(() => {
    resolve("promise Resolved value.!!");
    }, 1000);
});

async function handlepromise(){
    console.log("Hello world");
// Js Engine waiting for Promise to be resolved
    const val = await p;
```

```
console.log(val);
}
handlePromise();
```

Output:

```
Hellaworld
NamasteJavascript
Promise Resolve value !!
```

Things to note:-

(*) Wherever execution not at awaits the particular fun Suspended and moved out of call Stack.

(*) Once the Promises resolved in the awaits, then the Function will be Pushed into Call Stack and Starts Over from where it left.

```
Const P1 = new Promise((resolve, reject) => {
    SetTimeout(() => {
    resolve("Promise Resolved Value.!!");
    }, 20000);
});

Const P2 = new Promise((resolve, reject) => {
    SetTimeout(() => {
    resolve("Promise Resolved Value");
    }, 4000); });
```

async

```
console.log ("Helloworld!!");

const val = await p1;
Console.log ("namaste Javascript");
Console.log (val);
const val2 = await p2;
console.log ("namaste Javascript-2");
Console.log (val2);
}
handlePromise();
```

Real world Example of async/await :-

Fetch

(*) when the promise is resolved, gives you the response (object)

(*) Response body is the readable stream. (.json())

// Fetch() => Response.json() => Jsonvalue.

```
const API = "API URL";

async function handlePromises(){
  const data = await fetch(API);
  const jsonValue = await data.json()
```

fetch().then(res => res.json()).then(

Error handling while using await :-

```
Const API = "API URL";
async function handlePromise() {
  try {
    Const data = await fetch(API URL);
    Const jsonValue = await data.json();
    console.log(jsonValue);
  } catch(err){
    console.log(err);
  }
}

handlePromise();
```

(or) other ways

(i) Avoid catch in above code

```
handlePromise().catch((err) => console.log(err));
```

In Short :

In Promise	In async function (Preferable)
We use .then to resolve and .catch to reject	We use "await" before the promise and use "try" and "catch" for error handling (better)

⑦ "This" keyword :-

(i) "this" in Global Space.

```
console.log(this);
```

In Browser, The Global Object is "window"

In node JS, The Global Object is "Global".

(ii) "This" Inside a function

```
function x(){
  // The value depends on Strict / non Strict mode.
  console.log(this);
}
```

Output

Strictmode	non-Strict mode
Undefined	Window .

(*) (This Substitution) This in Strict mode :-

» If the value of this keyword is undefined or null, this keyword will be replaced with Global Object

Only in ^non-strict Strict mode.

(iii) "This" Keyword value depends on how the function is Called (Window)

```
function x(){
    console.log(this)
}
x();  → undefined
window.x() → window Object.
```

(iv) "This" Inside a Objects method

```
Const Obj = {
    a = 10,
    x: function (){
        console.log (this.a);
    },
};
Obj.x();
```

↳ "This" point Obj

So (this.a =) 10.

Call apply bind (Sharing methods)

Call

```
Const Student = {
    name : "AlcShay",
    Printname : function (){
        Console.log (this.name);
    },
};

Student.Print Name ();

Const Student2 = {
    name : "Deepika",
};
Student.Printname.Call (Student 2);
```

output
↳ Deepika.

(V) "this" Inside arrow function;

```
Const Obj = {
    a :10,
    x: () => {
```

Here, Obj present in Global object & The value of "This" Global Object i.e win

(*) Enclosing Lexical Context (*)

(*) Arrow functions doesn't provide its own binding

(iii) "this" inside nested arrow functions.

```
Const obj2 = {
  a:20,
  x: function (){
(//enclosing lexical context)
  Const y = () => {
    Console.log(this);
  };
  y();
  },
};
obj2.x();
```

(vii) "this" inside Dom elements

=> Reference to HTML element.

```
<button onClick = 'alert(this)'> Click me </button>
```