DOM + Modern JS - 4

* What is an API ?

API stands for Application Programming Interface. It is a particular set of rules and specifications that software program can follow to communicate with each other.

It serves as an interface between different software programs and facilitates their interactions, similar to the way the user interface facilities interaction between humans and computers.

Methods are :- GET , POST, PUT, HEAD, DELETE , PATCH, CONNECT , TRACE etc

* Features of Async-Code

→ Clean & Concise

→ Better Error handling

→ Easier bugging

PROMISE

→ Promise is used for parallel execution in the background of Javascript.

→ A promise is a proxy for a value not necessarily known when the promise is created.

→ It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.

* A promise has three states :-

→ pending :- initial state, neither fulfilled nor rejected.

→ fulfilled :- Completed successfully. (fulfilled with value)

→ rejected :- Operation failed. (Rejected with error)

* We use Promise() constructor for creating a new promise object.

* Syntax :-

```
let p = new Promise ( callback function() );
                  function (resolve, reject)
                  {
                  ===
                  }
```

fulfilled ← Successfully

failed → rejected Error

Creating a Promise :-

```
let meraPromise = new Promise (function (resolve, reject) {
        setTimeout (function () {
        console.log ("I am inside Promise");
        }, 5000);

        resolve (2023);   // Resolve with any values.
        //reject (new Error ('Error Generated---');
    });
```

Output :-

```
I am inside Promise
```
→ (after 5 seconds)

Console :-

```
meraPromise;
Promise {<fulfilled> : 2023}
```

In Case of Rejected :-

```
reject (new Error ('Error Generated---'))
```

Output :-

```
Error :- 'Error Generated'
```

Console :-

```
meraPromise;
Promise {<rejected> : Error : 'Error Generated---'}
```

parallel execution of Promise :-

```
let promise-1 = new Promise (function (resolve, reject) {
        setTimeout (function (){
            console.log ("I am inside Promise 1 ");
        }, 5000} ;
    })

let promise-2 = new Promise (function (resolve, reject) {
        setTimeout (function (){
            console.log ("I am inside Promise 2 ");
        }, 3000 } ;
    })
```

---x---x---

Promise :- $\boxed{\text{then () and Catch ()}}$

$$\begin{tikzcd}
\boxed{\text{Promise}} \arrow[r] \arrow[d] & \text{fulfilled} \arrow[r] & \boxed{\text{then ()}} \\
\text{Rejected} \arrow[d] & & \\
\boxed{\text{Catch () error}} & &
\end{tikzcd}$$

Example of then () method

①
```
let Promise-1 = new Promise(function(resolve, reject) {
                    resolve (12345678);
                }

Promise1. then ( (value) => {console.log (value)} );
```

Output :-
```
12345678
```

explanation :- Resolve is handled by then () method here.

② **Example of Catch () method**

```
let Promise-2 = new Promise ( ~~function~~ (resolve, reject ) =>
                    reject (new Error ("Error Generated")));

Promise-2. Catch ( (error) => {console.log ("Receive an Error ")});
```

Output:-
```
Recieve an Error
```

explanation :- Reject is handled by catch () method here.

---x---

Chaining Promise

```
Promise1. then ((value) => { ___ } , (error) => { ___ });
```

* Promise Parallel execution inside then().

```
let P1 = new Promise(function(resolve, reject){
        SetTimeout(() => {
                console.log('Set Time 1 Started');
            }, 2000);

        resolve (true);
    })
```

2nd में 2sec बाद Print होगा।

```
let Output = P1.then(() => {

        let P2 = new Promise(function(resolve, reject) {

        setTimeout (() => {

              console.log('set Time 2 Started');

           }, 3000);
          resolve ("P2 resolved"); })
      return P2;

   })

Output.then ((value) => console.log(value));
```

3rd में 3sec बाद Print

सबसे पहले Print

Output :-

```
P2 resolved
Set Time 1 Started
Set Time 2 Started
```

Async & Await keyword (Avoid Multiple then())

* "async and await" make promises easier to write

→ async :- keyword async place before a function makes the function return a promise.

Example :-

```
async function ABCD(){
        console.log("Hello Async");
        return 2023;
    }

console.log(ABCD());
```

(Any Value Resolve) or (any String)

Output:-

```
Hello Async
Promise {<fulfilled> : 2023}
```

→ async return a Promise here.

→ await :- 'await' keyword can only be used inside an async function.

It makes the function pause the execution and await for a resolved promise before it continues.

~~Ex:- async function ABCD(){~~
~~return "Completed";~~

P.T.O →

example of await :-

```
async function Utility(){

  let delhi = new Promise((resolve, reject) => {
        setTimeout(() => {
              resolve("Delhi is Capital");}, 1000);
        });

  let mumbai = new Promise((resolve, reject) => {
        setTimeout(() => {
              resolve("Finance Capital"):}, 3000);
        });

  let d = ~~delhi~~ await delhi;        // first Completely execute this.
  let m = await mumbai;                 // after 'd' Completion,.

  return [d, m];
}
```

Fetch API :-

* The fetch api provides a JS interface for accessing and manipulating parts of the protocal such as requests and responses.

* It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.

→ fetch () methods ~~s~~ starts the process of fetching a resource from a server.

→ fetch() method returns a Promise that resolves to a response object.

→ List of some freely available API :-

a.) 7 timer! (weather forecasts)
b.) Dogs (Random dog images)
c.) Joke Api (Jokes)
d.) JSON placeholder (fake Rest API for testing
   |
   |
  etc.

Retrieving data through fetch api (Get Call) using API

```
async function utility(){
      let content = await fetch('https://jsonplaceholder.
                   typicode.com/posts/1');
      let output = await content.json() ;
      console.log(output);
   }
utility();
```

→ converting content into json format

Output:-

```
{ userId: 1,
  id : 1 ,
  title : 'sunt aut facere -----',
  body: 'quet------------' }
```

we can check
- ✓ content.status
- ✓ content.ok
- content.json()
- ✓ content.text()
- etc.

JSON :- JavaScript Object Notation

JSON :- JavaScript Object Notation

→ JSON is a lightweight format for storing and transporting data.

→ JSON is often used when data is sent from a server to a web page.

JSON Syntax Rule :-

* key value pairs
* separated by commas.
* within { } hold objects
* [ ] holds array of objects.

example :-

```
{
  "employees":[
          {"firstname" : "John" , "lastName" : "Doe"},
          {"firstname" : "Peter" , "lastName" : "Jones"}
          ]
}
```

Send Data through Fetch Api (POST Calls)

fetch ('url') → Get ✓

Now,

fetch ('url', (options)) → POST

↓

Object for authenticity / POST Method.

```
async function helper () {
    let options = {
            method: 'POST',
            body: JSON.stringify({
                title: 'foo',
                body: 'bar',
                userId: 91, weight: 100,
            }),
          headers: {
              'Content-type': 'application/json;
                                  Charset=UTF-8',
              },
        };
  let content = await fetch ('https://jsonplaceholder.typicode.com/posts',
                        options);
  let response = content.json();
  return response;
}
```

POST Calls available on free API

```
async function utility () {
        let ans = helper;
        console.log (ans);
}
```

Output :- { title: 'foo', body: 'bar', userId: 91, weight: 100, id: 101 }

Note:- stringify () :- convert JS object into JSON String.

———x——————x———

CLOSURES

A closure is the combination of a function bundled together with references to its surrounding state (the lexical environment).

In other words, A closure gives you access to an outer function's scope from an inner function.

In JavaScript, Closures are created every time a function is created, at function creation time.

Problem in Normal Case

```
function init(){
      var name = "Mozilla";          //local variable
      function displayName(){
            console.log(name);
      }
      displayName();
  }
 init();
```

Note:- Variable name Cannot have accessed Outside init() function.
Outside the init(), variable name will be dead.

But Closures makes it accessible with the references.

```
function makeFunc(){
      const name = "Mozilla";        // name is surrounding
      function displayName(){        //here Making Closure
            console.log(name);          function
      }
      return displayName;
  }

const myFunc = makeFunc();           //myFunc is reference
myFunc();
```

In this case, As functions in JS forms Closures.

* myFunc is a reference to the instance of the function displayName() that is created when makeFunc() is run.

* The instance of displayName() maintains a reference to its lexical environment, within the Variable name exists.