19/02/24

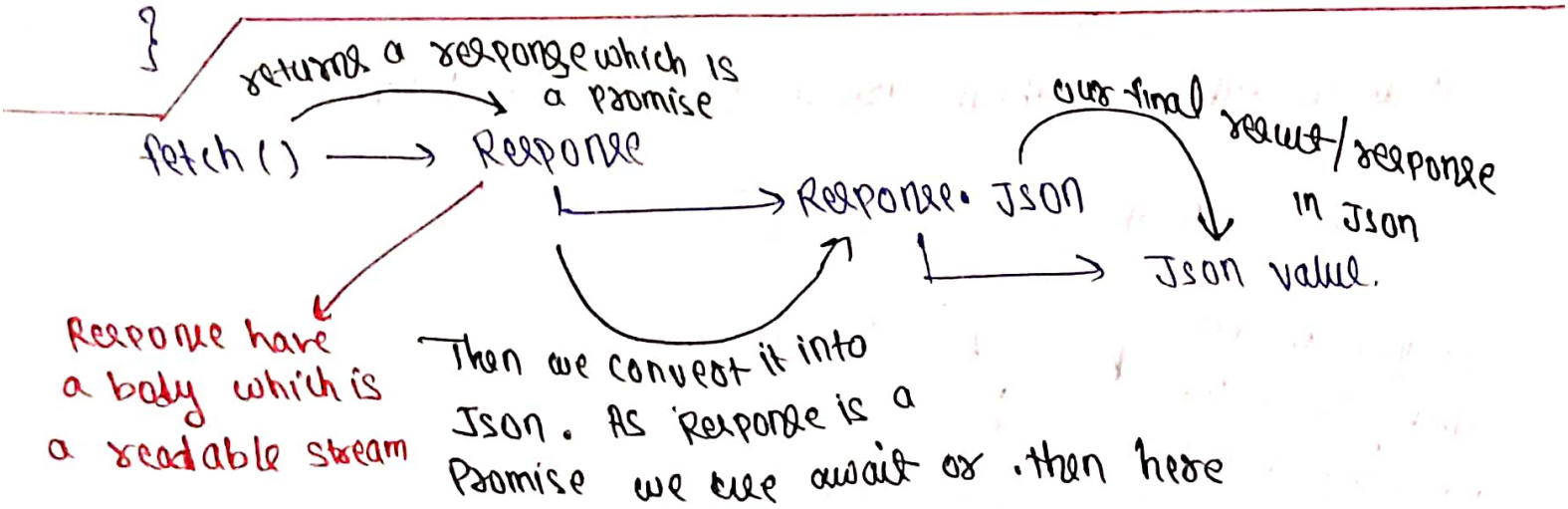
★ fetch.

→ It is an API which is provided by browser to make external call. It returns a Promise

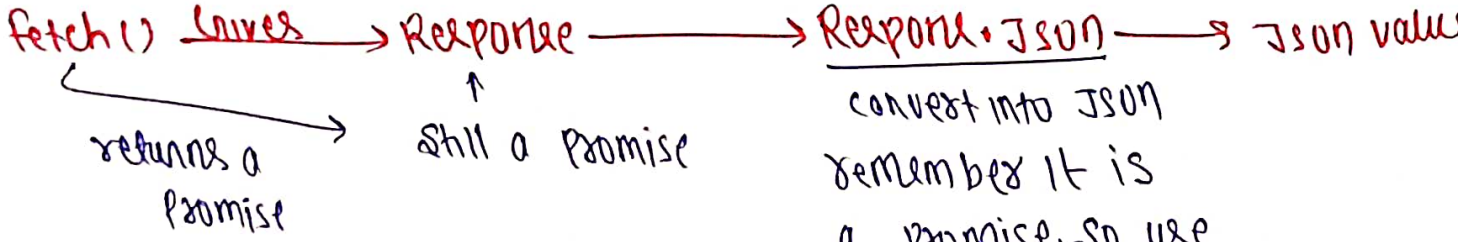
→ using async await

```
async fun-Name () {

    const data = await fetch (API_URL);

    const Json = await data.Json();
      console.log (Json);
// fetch().then((response) => response.Json()).then(

(response) => console.log (response));

}
```

As data.Json() is also promise that's why write await infront.



This is how fetch works....



If we use .then without arrow function to fetch.

```
const fetch().then( function (response) {
    return (response.json);
});
.then( function (response) {
    console.log (response);
});
```

If you are not using arrow func and doing promise chaining then you have to write return.

19/02/25

Callback Hell S-2 EP-01

→ Js is a Synchronous Single threaded language and it can just do one thing at a time.

→ It has just one call Stack and it can execute one thing at a time.

→ What ever code you will give to Js, it will be quickly executed by Js engine. It doesnot wait for anything.

→ But what if we want for Sometime / Something ?

we can do this using call back function.

Example :-

```
console.log ("Hellow")
console.log ("you")
console.log ("there")
```

→ Suppose we want to execute this after 5 seconds.

```
console.log ("Hellow")
setTimeout ( function () {
        console.log ("you");
}, 5000);
console.log ("there");
```

Higher order function

→ This is a callback function. This function will be called back by setTimeout function

→ We take a piece of code and it into another function as a call back function, which can be executed at later point of time.

→ In this way callback plays a very important role in asynchronous coding in Js

const cart = ["shirt", "pant", "kurta"]

Suppose we are creating an e-commerce website, where want to do following things:-

1) When someone buy something, create an order.
2) After creating order proceedToPayment

api.createOrder();
api.proceedToPayment();

→ we want createOrder to run first and then proceed to payment

↓

In such case we can use callback function

```
api.createOrder(cart, function() {
        api.proceedToPayment();
    });
});
```

→ Now this the responsibility of createOrder to call createOrder api first and then call the 2nd function which will call proceedTo payment.

→ createOrder is responsible for creating an order, when that process is completed it triggers the execution of proceedToPay.

→ This pattern is common in asynchronous Js programming, where a callback function is used to handle the next step after an asynchronous operation is finished.

→ Now Suppose after proceedToPayment we want 2 more things

1) showOrder Summary → After this execute updateWallet
2) updateWallet

```
api.createOrder(cart, function(){
    api.proceedToPayment(function(){
        api.showOrderSummary(function(){
            api.updateWallet();
        });
    });
});
```

callback Hell

This call back HELL is also Knows as Pyramid of DOOM

here our code starts growing horizontally not vertically

Here all 4 API will be called ~~one~~ one by one asynchronously after completion of each step.

createOrder ⟶ ProceedToPayment ⟶

↓

After completing this only next (ProceedToPayment) will be called. until then it will wait

→ When we have a callback inside a callback, another callback inside that callback & so on this is called callback Hell.

→ Above type of code is unreadable and un-manageable/unmaintainable

★ Inversion of control : When you loose your control over your ~~callback function~~ code while using callbacks.

```
api.sayHellow('Hello', function(){
    api.sayBye();
});
```

→ we have give this function to sayhellow api & we are sitting and blindly trusting on Sayhellow That at some point of time it will complete its work and will call Saybye.

" But this is RISKY "

Case Why it is risky ?

→ we have given our saybye function control to Sayhello, what if Sayhello never calls Saybye !!

→ There could be bugs in sayhello, you don't know what code is written inside sayhellow

→ what if Saybye is called twice ?

→ How to handle this ?? → promises

Conclusion : 1) we use callback functions for asynchronous Programming in JS.

2) callback exist that's why asynchronous Programming exists.

3) Issues with call back

→ 1) call back Hell

→ 2) Inversion of control.

↓

we are giving access or control of our code (callback function) to another api which we don't how it will work, How it will behave.

19/02/24

S2 - EP-02

Promises

→ Promises are used to handle async operations in JS.

→ Asynchronous :- We don't know how much time it will take, and it will run in parallel without blocking the execution of other code.

→ We can use call back function for asynchronous operation/coding.
But we have 2 problems → 1) call back Hell 2) Inversion of control with using api in callback.

→ we can solve this problem using promises

```
const cart = ["shoes", "shirt", "kurta"];
const promise = createOrder(cart);
```

→ In above code, createOrder api will be called and it will return an an object to promise variable. / or it will return an empty object called promise.

{ data : undefined }

This it will return an object with data field, whose value will be empty i.e, undefined.

→ After this createOrder api will take its time, until / during that time if they are some code lines below it, JS will execute it.

→ When it's time will be completed, it will return data back into that object.

when our JS engine see this line
→ At this point of time it return us an empty object { data : undefined } with undefined data field

→ const promise = createOrder(cart)

createOrder will take some time to get data / fetch data until then below written code will be executed.
when it will get the data, it will automatically fetch that data in that object (known as promise) asynchronously. from API.

* Promise is nothing just that object or data which we are getting.

```
Promise.then( function () {
                proceed to payment ( );
              });
```
↑ we are passing this func. as an argument.

.then is a function which we can run on the promise object.

→ promise object.

```
const promise = createOrder();
```

→ The point of time when promise object will be filled with the data which our api will give. This .then function will be executed.

→ And the callback function which we have written into .then function will be automatically called. (only once).

→ when we pass any function as an argument into another function then it is called callback function.

→ In this case we have control over our callback function/program

★ PromiseState: It tells us the state of the promise, when initially we didnot have any data into our promise object it will be pending.

→ After sometime when we will get data from api it will turn into 'fulfilled state'. we can also have rejected state.

★ PromiseResult: It will have the data which we will get from api. Initially it will be undefined, after getting data, it will be shown here.

You can see promiseState & PromiseResult in ~~Network~~ sources. → scope.

```
promise.then ( function (data) {
                 console.log (data);
               });
```
→ data which we will get from api.

★ promise object is im-mutable. (you/nobody can't edit it)

→ promise is an object representing the eventual completion or ~~failure~~ of an asynchronous operation.

→ through .this function we have solved the problem of inversion of control.

→ Another problem with callback code pyramid of doom or callback Hell

↳ This is solved by promise chaining

Callback Hell

```
const cart=["shoe","pant","shirt"];

createObje

createOrder(cart, function(orderId){
    proceedToPayment(orderId, function(paymentInfo){
        showOrderSummary(paymentInfo, function(){
            updateWallet();
        });
    });
});
```

→

```
const promise = createOrder();

promise.then(proceedToe
    function(orderId){
        proceedToPayment();
    });
);
```

Both are Same

```
createOrder().
then(function(orderId){
    proceedToPayment();
});
```

→ Promise chaining :-

In this way we can do Promise chaining.

```
createOrder(cart)
  .then(function(orderId){
    return proceedToPayment(orderId);
  })
  .then(function(paymentInfo){
    return showOrderSummary(paymentInfo);
  })
```

very imp to receive the data and not to loose any data.

In Promise chaining we can also use arrow functions :-

```
createOrder(cart)
.then((orderId) => proceedToPayment(orderId))
.then((paymentInfo) => showOrderSummary(paymentInfo))
.then((paymentInfo) => updateWallet(paymentInfo));
```

Just see the syntax. 😊 don't worry about what are these functions and these arguments.

Conclusion :-

Inversion of control problem is solved by using Promise object and .then function.

Callback Hell problem is resolved using Promise chaining.

* .catch() . Just like .then we can also use .catch to catch any error.

```
promise.catch(function(error) {
    console.log(error);
});
```

→ If the promise gets resolved then control of program will go to .then() block, if the promise gets rejected meaning it encounters an error, .catch() function will be invoked and .then() function will not be executed in this case.

13/02/24

★ Async Await

→ What is async?

↳ Async is a keyword, that it used before a function to create an async function.

→ What is async function & how it is different from normal function.?

★ Async function will always return a promise

→ Either you can ~~return~~ return a promise

```
async function getData () {
    return Promise;
}
```
case ①

OR if you return any other non-promise value

```
async function getData(){
    return "Namaste";
}
```
In this case "Namaste" will be wrapped inside a promise

case ②

→ If you will do console.log (getData) from case 2, you will get a promise object and "Namaste" will be written inside PromiseResult.

→ If you ~~written~~ return any normal value inside async function it will be automatically wrapped inside a promise. After calling async function, you can use .then function, to get the returned data.

→ ★ Async & Await combo is used to handle promises.

→ use the word await in front of promise. Await keyword can only be used inside async function

"Time, tide & Js wait for none"!

→ we can use both .then or async await to ~~resolve~~ Handle promises

→ If we use .then, then it will not wait for the Promise to be resolved and Js engine will execute the code of below line.

→ once the promise is resolved then only .then function will be called.

→ Resolved :- Resolved basically means when we get our data from api. We know that we immediately don't get the data from API. It takes some time after some time we get our Promise (result). Thus resolved means getting data from API

** When we use asyn await, our JS engine actually waits for the promise to resolved, once the promise has been resolved then only it will execute the code of below line.

```
                    → promise
const p = new Promise((resolved, reject) => {
    setTimeout(() => {
        resolved("Promise is resolved");
    }, 10000);
});
```

creating a dummy promise which which gives us "Promise is resolved" as a promise

→ using .then

Both are same

```
function getData() {
    .then(function (result) {
        console.log(result);
        console.log("Hello");
    });
}
getData();
```

```
.then((result) => console.log
    (result));
console.log("Hello");
);
```

calling function

output :
Hello → It will be instantly printed
Promise is resolved → It will be printed after 10 sec.

→ using async await

```
async function getData() {
    const val = await p;
    console.log("Hello");
};
```

output :
Promise is resolved → printed after 10 sec
Hello → After then hello gets printed

→ our whole code will be blocked at `const val = await p.` until the promise is resolved and then our code with start executing. The program will wait.

★ fetch() : It is an API given by browser (not by JS.) to us, to make an external call. It returns us a promise.

```
const promise = fetch(API-URL);
```

★ Remember :- It seems like JS engine is waiting to execute, but actually it is not. If it would be waiting then our page would get freezed. But it don't, if there are some other functionality, like clicking on the page/button (function). It will work fine.

→ How it works behind the scenes :-

→ ① JS have only one call stack & it executes the code synchronously line by line.

→ ② See in code we have 2 functions showData(); & then timepass();

→ ③ We have also created 2 promises P_1 with 10 sec & P_2 with 20 sec time to get resolved.

→ ④ So behind the scenes, JS engine (engineer) counter showData function. So showdata func. goes into JS stack

→ ⑤ As it sees an await, it immediately suspend or put out this function from call stack, and the other function timepass comes into call stack. and executing timepass fun goes & out of function call stack.

→ ⑥ After 10 sec showData fun comes back into call stack and executes * from where it left the function after await line.

(7) Now another asynch needs 20 sec to be resolved. During 10 sec when P_1 (promise) was getting executed P_2 will be half resolved in those 10 sec.

(8) Now as fun call stack sees another async statement, it again suspend that function & ShowData function goes out of stack.

(9) After other 10 secs (seconds) when P_2 get resolved, showData func. will be again come into func. call stack & it be executed from where it left. i.e, next line from async statement.

→ Suppose in our code we had P_1 = 10 sec time & P_2 = 5 sec. Then after 10 sec both Promises will be executed because:-

↳ By the time of 10 sec when P_1 was getting resolved, in between that span of 10 sec. P_2 will be ~~executed~~ resolved in 1st 5 sec.

Note :- The point of time when a promise is created it starts ~~to~~ getting executed.

★ Error Handling

```
async fun_name() {
    try {                          → This block will be executed when there is no error
        const data = await fetch(URL);
        const json = await data.json();
        console.log(json);
    }
    catch(err) {
        console.log(err);          ↓ This block will be executed when there will be any error like API_URL is wrong etc.
    }
}
```

```
async fun_name(){
    const data = await fetch(URL);
    const json = await data.json();
    console.log(json);
};

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

Because async fun. always returns a promise so we can use .catch() here

→ what to use async await or .then.

→ Rember async await is just a Syntactic Sugar over .then is .catch.

→ It means behind the scenes if will write async await, JS will treat is as same as how it treats .then & .catch

→ It is just change in Syntax, behind the scenes it is using its own promise.then and promise.catch.

Que which one is better?

Both are good, async await is just a new Syntax!!!
and Sometimes it makes a little sense to write like await

→ No promise chaining. 

I'd be glad to explain the new Promise keyword and the resolve keyword in JavaScript:

new Promise:

- **Purpose:** Used to create a new Promise object, which serves as a placeholder for an asynchronous operation's eventual outcome (success or failure).
- **Syntax:**

JavaScript

```
new Promise((resolve, reject) => {
  // Asynchronous operation here
  // Call resolve(value) if successful, reject(reason) if unsuccessful
});
```

- **Explanation:**
    - Takes an executor function as an argument, where resolve and reject are callback functions provided by the Promise itself.
    - Inside the executor function, perform your asynchronous operation (e.g., network request, file I/O, setTimeout).
    - If the operation succeeds, call resolve(value) to indicate success and provide the resolved value.
    - If the operation fails, call reject(reason) to indicate an error and provide a reason (optional).

Example:

JavaScript

```
new Promise((resolve, reject) => {
  const value = Math.random();
  if (value > 0.5) {
    resolve("It's greater than 0.5!");
  } else {
    reject("It's not greater than 0.5.");
  }
})
.then(result => {
```

```
  console.log("Fulfilled:", result);
})
.catch(error => {
  console.error("Rejected:", error);
});
```

resolve Keyword:

- **Purpose:** Signals that the asynchronous operation associated with the Promise has successfully completed and provides the resolved value.
- **Usage:** Called within the executor function to indicate a successful outcome and provide a value:

JavaScript

```
new Promise((resolve, reject) => {
  // ... (asynchronous operation)
  if (condition) {
    resolve("success value"); // Promise resolves successfully
  } else {
    // Handle unsuccessful cases or call reject() if appropriate
  }
});
```

- **Chaining Promises:** Subsequent then() methods receive this resolved value to continue the promise chain.

Key Points:

- new Promise creates a Promise object that represents the eventual outcome of an asynchronous operation.
- resolve indicates a successful outcome and provides a value within the executor function.
- reject indicates an error or failure and provides a reason (optional).
- Promises help manage asynchronous operations in a more structured and readable way.