CallBacks, Promises, Async/await and Fetch

CallBacks

```
//functions as first class citizens
//higher order function  --> can accept a function as an argument


function greet(callback){
    console.log("hi");
    callback();

}

function greetnxt()
{
    console.log("good Morning");
}

greet(greetnxt);

const numbers = [4,2,1,-2,-2,-1,5]

const posNumbers  =  removeNeg(numbers, (num) => num >=0)

function removeNeg (numbers,callback){
    const arr = [];
    for(const num of numbers)
    {
        if(callback(num))
        {
            arr.push(num)
        }
    }
    return arr;
}

console.log(posNumbers)


const double = numbers.map((num) => num * 2)
```

```
console.log(double);


console.log("Start");
setTimeout(() =>
{
    console.log("Timer completed ")
}, 3000);
console.log("Timer Initiated ")


// callback hell --> when callbacks are deeply nested

function task1(callback) {
    setTimeout(() => {
    console.log("Task 1 is completed ")
    callback();
    }, 1000)
}

function task2(callback) {
    setTimeout(() => {
    console.log("Task 2 is completed ")
    callback();
    }, 1000)
}

function task3(callback) {
    setTimeout(() => {
    console.log("Task 3 is completed ")
    callback();
    }, 1000)
}


task1(() =>{
    task2(() =>
    {
        task3(() => {
            console.log("Task3 is completed ")
        })
    })
})
```

Promises

```
Promises

// 3 states --> pending --> initial state , promise has neither been
fulfilld nor rejected
// Fulfilled ---> The operation is successful and you are ggetting a
resolved value
// Rejected --> the operation failed

let promise = new Promise((resolve, reject) =>{
        setTimeout(() =>{
          let success  = false;
          if(success){
            resolve("Promise resolved");
          }
          else{
            reject("promise rejected");
          }
        },3000)
});


promise
  .then((success) => {
    console.log(success)
  })
  .catch((error) => {
    console.log(error)
  })


console.log("Step 1: Start");  // Synchronous

let promise2 = new Promise((resolve, reject) => {
    console.log("Step 2: Inside Promise");  // Synchronous
    setTimeout(() => {
        resolve("Step 4: Promise Resolved");  // Asynchronous
    }, 2000);
});
```

```
promise2.then(result => console.log(result));

console.log("Step 3: End");  // Synchronous

```

Async/Await

```
let promise = new Promise((resolve, reject) =>{
        setTimeout(() =>{
          let success  = false;
          if(success){
            resolve("Promise resolved");
          }
          else{
            reject("promise rejected");
          }
        },3000)
});

async function funncAsync() {
    console.log("before try catch")
    try {
        let success = await promise;
        console.log("Result Fetched", success)
    }
   catch(error){
    console.log("error occured ")
   }
   console.log(" After try catch")
}

console.log("Before funcasync")

funncAsync();

console.log("After funcasync")
```

Fetch

Arguments for `fetch()` in JavaScript

The `fetch()` function takes two arguments:

1 **URL (Required)** – The resource to fetch.
2 **Options (Optional)** – An object that configures the request method, headers, body, etc.

1️⃣ Basic Syntax

```
fetch(url, options);
```

- `url` → The endpoint (string) from which to fetch data.
- `options` → An object with configuration properties (optional).

2️⃣ Common Arguments for `fetch()`

Argument	Type	Default	Description
`method`	`string`	`"GET"`	HTTP method (`GET`, `POST`, `PUT`, `DELETE`, etc.)
`headers`	`object`	`{}`	HTTP headers (e.g., `Content-Type`)
`body`	`string`	`null`	Data to send with `POST`, `PUT`, `PATCH` requests
`mode`	`string`	`"cors"`	Mode of request (`cors`, `same-origin`, `no-cors`)
`credentials`	`string`	`"same-origin"`	Handle cookies (`same-origin`, `include`, `omit`)
`cache`	`string`	`"default"`	How caching should be handled (`default`, `no-cache`, `reload`, `force-cache`, `only-if-cached`)
`redirect`	`string`	`"follow"`	Handle redirects (`follow`, `error`, `manual`)

3️⃣ Example: `GET` Request

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error("Error:", error));
```

Uses the default `GET` method.

4️⃣ Example: `POST` Request with Headers & Body

```
fetch("https://jsonplaceholder.typicode.com/posts", {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify({
        title: "New Post",
        body: "This is a new post.",
        userId: 1
    })
})
.then(response => response.json())
.then(data => console.log("Created:", data))
.catch(error => console.error("Error:", error));
```

Sets `method`, `headers`, and `body`.

5️⃣ Example: Sending Credentials (Cookies, Auth)

```
fetch("https://example.com/api/user", {
    method: "GET",
    credentials: "include" // Ensures cookies are sent
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error("Error:", error));
```

`"include"` allows cross-origin cookies.

6️⃣ Example: Handling Redirects

```
fetch("https://example.com/api", {
    redirect: "error" // Will throw an error if redirected
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error("Redirect Error:", error));
```

✅ `"error"` prevents automatic redirects.

7️⃣ Example: Disabling Cache

```
fetch("https://example.com/data", {
    cache: "no-cache" // Forces fresh response
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error("Error:", error));
```

`"no-cache"` ensures fresh data instead of cached results.