Class 05 – ES6

ADVANCE JAVASCRIPT

MERN STACK DEVELOPMENT - MODULE B

Today's Topics

- Callbacks
- Promises
- Async/Await
- Default Perimeter
- IIFE

callbacks

A callback in JavaScript is a function that is passed as an argument to another function. Instead of executing the callback immediately, the function "calls back" or invokes the callback later, typically after some asynchronous operation completes.

Callbacks are commonly used in asynchronous operations, like fetching data, handling events, or executing code after a certain task completes. They allow for more flexible and responsive programming in JavaScript.

example

```
const fetchData = (callback) => {
                setTimeout(() => {
                                const data = "Hello, Callbacks!"; // Invoke the callback with the fetched data
                                callback(data);
                }, 1000); // Simulating a delay of 1000 milliseconds (1 second) };
const handleData = (data) => {
                console.log("Received data:", data); }; // Call fetchData and pass handleData as the callback
fetchData(handleData);
```

Promises

A Promise is an object representing the eventual completion or failure of an asynchronous operation. Since most people are consumers of already-created promises, this guide will explain consumption of returned promises before explaining how to create them.

example

```
// Simulating an asynchronous operation with a Promise
const fetchData = () => new Promise((resolve, reject) => {
    setTimeout(() => {
        const isSuccess = true; isSuccess ? resolve("Hello, Promises!") : reject("Error fetching data");

}, 1000); }); // Using the Promise

fetchData()
.then(data => console.log("Received data:", data)) .
catch(error => console.error("Error:", error));
```

async/await

Async/Await is a feature in JavaScript that allows you to write asynchronous code using a more synchronous style. The async keyword is used to declare asynchronous functions, and the await keyword is used to pause the execution of a function until a Promise is resolved, making asynchronous code appear more linear and readable.

example

```
async function exampleAsyncFunction() {
    try {
        const result = await someAsyncOperation(); // Code here will wait until someAsyncOperation is complete
    } catch (error) {
        console.error('Error:', error);
    }
```

Default Parameters

Default parameters in JavaScript allow you to assign default values to function parameters in case they are not explicitly provided by the caller. This feature was introduced in ECMAScript 6 (ES6) to simplify function definitions and make them more flexible.

example

```
// Function with default parameters
function greet(name = 'Guest', greeting = 'Hello') {
    console.log(`${greeting}, ${name}!`);

}

// Calling the function without providing parameters
greet(); // Output: Hello, Guest!


//Calling the function with custom parameters
greet('John', 'Hi'); // Output: Hi, John!
```

IIFE

(Immediately Invoked Function Expression)

An IIFE is a JavaScript design pattern that involves defining and executing a function immediately after its creation. This pattern is used to create a private scope for variables, preventing them from polluting the global scope.

example

```
// Concise IIFE using an anonymous arrow function
(() => {
let message = "Hello from IIFE!";

console.log(message)
})()
```

```
// Concise IIFE using an anonymous function

const result = ( () => {
    let message = "Hello from IIFE!";
    return message;
})();

console.log(result); // Output: Hello from IIFE!
```

That's it for today!

Keep coding, stay curious, and enjoy your JavaScript journey!