**PART A**

**Experiment No. 03**

**A.1 AIM:**

Applying Callback Promise and Async Await to solve the issue of Callback hell in an Asynchronous Operation.

**A.2 Pre requisite:**

Basic knowledge Javascript ES6

**A.3 Outcome:**

After successful completion of this experiment students will be able to understand to concept of Form Asycn Communication solve the issue of Callback hell in an Asynchronous Operation.

**A.4 Theory**

A callback is a function passed into another function as an argument to be executed later.

* A high-order function is a function that accepts another function as an argument.
* Callback functions can be **synchronous or asynchronous.**

**Callback Hell** Callback Hell refers to a situation where callbacks are nested within other callbacks, resulting in code that is difficult to read and maintain.

**Solutions**:

* **Promises**: Handle asynchronous operations with cleaner syntax.
* **Async/Await**: Write asynchronous code that resembles synchronous code for better readability.

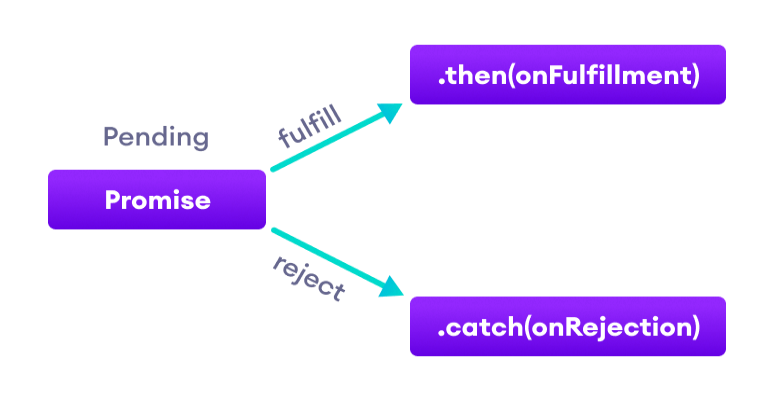
A Promise is a JavaScript **object** that represents **eventual completion (or failure)** of an **asynchronous operation** and its resulting value.

**setTimeout():** Executes a function after a specified delay.

setTimeout(function(){

console.log("Hello World");

}, 2000);



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

// Make an asynchronous call and either resolve or reject

});

promise

.then(result => {

console.log(result);

return 'Next step';

})

.then(nextStep => {

console.log(nextStep); // Output: Next step

})

.catch(error => {

console.error(error);

});

Aync/Await is a feature in JavaScript that **simplifies working with asynchronous code**, making it **more readable and easier to understand**.

async function fetchDataAsync() {

try {

const data = **await** fetchData();

console.log("Async/Await Result:", data);

} catch (error) {

console.error("Async/Await Error:", error);

}

}

// Using async/await

fetchDataAsync();

**A.5 Solving Callback Issue:**

**Given a food ordering process which** follows of sequential execution with callbacks

* Step 1: Placing the food order.
* Step 2: Processing the payment.
* Step 3: Preparing the order in the kitchen.
* Final Step: Delivering the order to the customer.

function step1(callback) {

setTimeout(function () {

console.log("Step 1: Order placed successfully");

callback();

}, 2000);

}

function step2(callback) {

setTimeout(function () {

console.log("Step 2: Payment processed");

callback();

}, 4000);

}

function step3(callback) {

setTimeout(function () {

console.log("Step 3: Order being prepared");

callback();

}, 3000);

}

function finalStep() {

console.log("Step 4: Order delivered! Enjoy your meal!");

}

// Using callbacks to ensure steps are executed in sequence

step1(() => {

step2(() => {

step3(() => {

finalStep();

});

});

});

console.log("Other services are running in the background... bye!");

**Problem1**: convert the above food ordering process using Promises to handle the asynchronous flow:

* Promises: Each step returns a Promise that resolves when the task is done.
* .then() chaining: Ensures that each step is executed in sequence once the previous step is complete.
* Error handling: a .catch() block is added to handle any potential errors in the promise chain.

**Problem2**: By using the solution of Promlem1 where promise was used now convert the food ordering scenario using async/await for better readability and easier handling of asynchronous tasks:

* async function: The processOrder function is declared as async so that we can use await inside it.
* await keyword: Ensures that each step is executed only after the previous step completes.
* Error handling: A try/catch block is used to catch any errors during the asynchronous operations.
* Cleaner flow: The use of async/await makes the code more synchronous-like, making it easier to follow and maintain.

**Problem3**:Refactor the **code given below** and make it more concise

1. Inlining the promise-returning functions.
2. Using async/await with direct definitions to reduce boilerplate.
3. Removing unnecessary intermediate variables and making the code more streamlined**.**

const delay = (message, time) => new Promise(resolve => setTimeout(() => {

console.log(message);

resolve();

}, time));

function step1() {

return delay("Step 1: Order placed successfully", 2000);

}

function step2() {

return delay("Step 2: Payment processed", 4000);

}

function step3() {

return delay("Step 3: Order being prepared", 3000);

}

function finalStep() {

console.log("Step 4: Order delivered! Enjoy your meal!");

}

async function processOrder() {

try {

await step1();

await step2();

await step3();

finalStep();

} catch (error) {

console.log("Something went wrong:", error);

}

}

processOrder();

console.log("Other services are running in the background... bye!");

**Hint:**

const delay = (message, time) => new Promise(resolve => setTimeout(() => { console.log(message); resolve(); }, time));

**PART B**

(PART B: TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)

|  |  |
| --- | --- |
| Roll No. : S020 | Name: Husain Chhil |
| Class : MBA.Tech DS Sem V | Batch : J1 |
| Date of Experiment :260924 | Date/Time of Submission : 260924 |
| Grade : |  |

**B.1 Code:**

**Problem1**

function step1(){

    return **new** *Promise*((*resolve*, *reject*) => {

        setTimeout(() => {

            console.log("Order placed successfully");

            resolve();

        }, 2000);

    });

}

function step2(){

    return **new** *Promise*((*resolve*, *reject*) => {

        setTimeout(() => {

            console.log("Payment processed successfully");

            resolve();

        }, 4000);

    });

}

function step3(){

    return **new** *Promise*((*resolve*, *reject*) => {

        setTimeout(() => {

            console.log("Order being prepared");

            resolve();

        }, 3000);

    });

}

function finalStep(){

    console.log("Order delivered successfully");

}

step1()

    .then(() => {

        return step2();

    })

    .then(() => {

        return step3();

    })

    .then(() => {

        return finalStep();

    })

    .catch((*err*) => {

        console.log(*err*);

    });

console.log("Other services are running in the background");

**Problem2**

function step1(){

    return **new** *Promise*((*resolve*, *reject*) => {

        setTimeout(() => {

            console.log("Order placed successfully");

            resolve();

        }, 2000);

    });

}

function step2(){

    return **new** *Promise*((*resolve*, *reject*) => {

        setTimeout(() => {

            console.log("Payment processed successfully");

            resolve();

        }, 4000);

    });

}

function step3(){

    return **new** *Promise*((*resolve*, *reject*) => {

        setTimeout(() => {

            console.log("Order being prepared");

            resolve();

        }, 3000);

    });

}

function finalStep(){

    console.log("Order delivered successfully");

}

async function main() {

    try {

        await step1();

        await step2();

        await step3();

        finalStep();

    } catch (err) {

        console.log(err);

    }

}

main();

console.log("Other services are running in the background");

**Problem 3**

const delay = (*message*, *time*) =>

**new** *Promise*((*resolve*) =>

    setTimeout(() => {

      console.log(*message*);

      resolve();

    }, *time*)

  );

async function processOrder() {

  try {

    await delay("Step 1: Order placed successfully", 2000);

    await delay("Step 2: Payment processed", 4000);

    await delay("Step 3: Order being prepared", 3000);

    console.log("Step 4: Order delivered! Enjoy your meal!");

  } catch (error) {

    console.log("Something went wrong:", error);

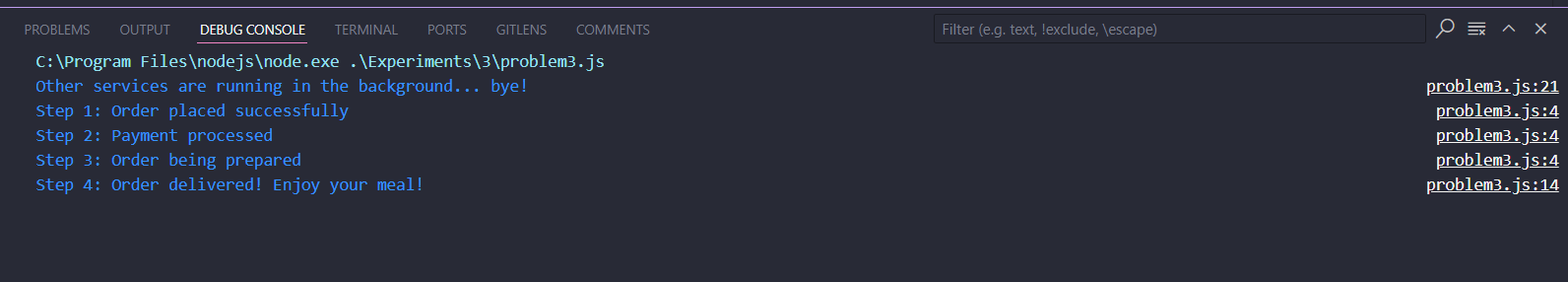
  }

}

processOrder();

console.log("Other services are running in the background... bye!");

**B.2 Output**

****

**B.3 Conclusion:**

Async-await is a modern way of handling asynchronous operations in JavaScript. It is built on top of promises and provides a more readable and concise syntax for handling asynchronous code. Async-await allows you to write asynchronous code that looks synchronous, making it easier to understand and maintain. It also provides a way to handle errors using try-catch blocks, which makes error handling more straightforward.

**B.3 Observations and Learning:**

async functions are a syntactic sugar over promises

async functions return a promise

await can be used only inside an async function

await pauses the execution of the async function until the promise is resolved

async functions can be used to write synchronous looking code for asynchronous operations

async functions can be used to handle errors in a cleaner way than using then and catch

**B.4 Question of Curiosity**

Q1. What are JavaScript Arrow Functions and HigerOrder function?

Ans. Arrow functions are a new way to write anonymous function expressions in JavaScript. They are more concise and easier to read than traditional function expressions. Higher-order functions are functions that take other functions as arguments or return functions as their results. They are a powerful feature of functional programming that allows you to write more flexible and reusable code.

Q2. What are the different possible state of promise object?

Pending, Fulfilled, Rejected

Q3. How to handle errors in Promise and Asyc Await?

Ans. We can use try-catch block to handle errors in async/await functions. We can also use .catch() method to handle errors in Promises.