**Working with Callbacks, Promises, and Async/Await in JavaScript**

**1. Callbacks**

A callback is a function that is passed as an argument to another function and is executed after some operation has been completed. Callbacks are often used in asynchronous operations, like reading files, making API requests, or handling user interactions.

**Example: Simple Callback**

function greet(name, callback) {

console.log('Hello ' + name);

callback();

}

function askQuestion() {

console.log('How are you today?');

}

greet('Alice', askQuestion);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Explanation:**

* The `greet` function takes a name and a callback function as arguments.
* After greeting the user, it calls the `askQuestion` function.

**Example: Callback with Asynchronous Code**

function fetchData(callback) {

setTimeout(() => {

console.log('Data fetched!');

callback();

}, 2000); // Simulate a 2-second delay

}

function processData() {

console.log('Processing data...');

}

fetchData(processData);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Explanation:**

* `fetchData` simulates fetching data with a delay and calls `processData` after the data is fetched.

**2. Promises**

A Promise is an object representing the eventual completion or failure of an asynchronous operation. It allows you to attach callbacks to handle the success or failure of the operation.

**Creating a Promise**

const fetchData = new Promise((resolve, reject) => {

const success = true; // Simulate success or failure

setTimeout(() => {

if (success) {

resolve('Data fetched successfully!');

} else {

reject('Error fetching data.');

}

}, 2000);

});

fetchData

.then((message) => {

console.log(message); // Handle success

})

.catch((error) => {

console.error(error); // Handle failure

});

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Explanation:**

* A `Promise` is created with two arguments: `resolve` and `reject`.
* If the operation is successful, `resolve` is called; otherwise, `reject` is called.
* The `then` method handles the resolved value, while the `catch` method handles any errors.

**Chaining Promises**

You can chain multiple `then` methods to handle a sequence of asynchronous operations.

fetchData

.then((message) => {

console.log(message);

return 'Next step';

})

.then((nextMessage) => {

console.log(nextMessage);

})

.catch((error) => {

console.error(error);

});

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**3. Async/Await**

`async` and `await` are modern JavaScript features that provide a more readable and intuitive way to work with Promises. They allow you to write asynchronous code that looks synchronous.

**Example: Basic Async/Await**

async function fetchData() {

try {

const response = await new Promise((resolve, reject) => {

const success = true;

setTimeout(() => {

if (success) {

resolve('Data fetched successfully!');

} else {

reject('Error fetching data.');

}

}, 2000);

});

console.log(response);

} catch (error) {

console.error(error);

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

fetchData();

Explanation:

* The `async` keyword before a function means that the function will always return a Promise.
* The `await` keyword is used to wait for a Promise to resolve. It can only be used inside an `async` function.
* `try/catch` blocks handle any errors that occur during the asynchronous operation.

**Example: Chaining Async/Await**

async function fetchAndProcessData() {

try {

const data = await fetchData();

console.log(data);

const processedData = await processData(data);

console.log(processedData);

} catch (error) {

console.error(error);

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Explanation:**

* This example demonstrates chaining asynchronous operations using `async/await`. It waits for `fetchData` to complete before moving on to `processData`.

**Summary**

* Callbacks are functions passed as arguments to be executed later.
* Promises provide a more structured way to handle asynchronous operations, with `then` and `catch` for handling success and failure.
* Async/Await allows for writing asynchronous code in a more synchronous and readable manner.

Next Steps

Practice these concepts by creating projects that involve API requests, data fetching, and processing. Try converting callback-based code to use Promises and async/await to see the differences in code structure and readability.