# Promises in JavaScript

## Introduction

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation. It provides a cleaner, more manageable way to handle asynchronous tasks compared to callbacks.

## Key Features

1. States of a Promise:  
 - Pending: Initial state; neither fulfilled nor rejected.  
 - Fulfilled: The operation completed successfully.  
 - Rejected: The operation failed.  
2. Chaining: Use `.then()`, `.catch()`, and `.finally()` to handle promise results and errors.  
3. Avoids Callback Hell: Promises enable cleaner code by eliminating deeply nested callbacks.

## Syntax

const promise = new Promise((resolve, reject) => {  
 // Asynchronous operation  
 if (/\* operation success \*/) {  
 resolve('Success message');  
 } else {  
 reject('Error message');  
 }  
});  
  
promise  
 .then(result => console.log(result)) // Handle success  
 .catch(error => console.error(error)) // Handle error  
 .finally(() => console.log('Done')); // Executes regardless of the result

## Example

Fetching data asynchronously with Promises:  
function fetchData() {  
 return new Promise((resolve, reject) => {  
 setTimeout(() => {  
 const success = true; // Simulate success  
 if (success) {  
 resolve('Data fetched successfully!');  
 } else {  
 reject('Failed to fetch data.');  
 }  
 }, 2000);  
 });  
}  
  
fetchData()  
 .then(data => console.log(data)) // Logs: Data fetched successfully!  
 .catch(error => console.error(error))  
 .finally(() => console.log('Fetching process completed.'));

## Case Study: User Authentication System

Scenario:  
You need to build a user authentication system where:  
1. The user's credentials are verified using an asynchronous call.  
2. If successful, the user's profile data is fetched.  
3. If either step fails, an appropriate error message is displayed.

## Solution Using Promises

Code Implementation:  
function verifyCredentials(username, password) {  
 return new Promise((resolve, reject) => {  
 setTimeout(() => {  
 if (username === 'admin' && password === '1234') {  
 resolve('User authenticated');  
 } else {  
 reject('Invalid credentials');  
 }  
 }, 1000);  
 });  
}  
  
function fetchUserProfile() {  
 return new Promise((resolve, reject) => {  
 setTimeout(() => {  
 resolve({ name: 'John Doe', role: 'Admin' });  
 }, 1000);  
 });  
}  
  
function authenticateUser(username, password) {  
 verifyCredentials(username, password)  
 .then((message) => {  
 console.log(message); // Log: User authenticated  
 return fetchUserProfile();  
 })  
 .then((profile) => {  
 console.log('User Profile:', profile); // Log user profile  
 })  
 .catch((error) => {  
 console.error('Error:', error); // Log any errors  
 })  
 .finally(() => {  
 console.log('Authentication process completed.'); // Always executed  
 });  
}  
  
// Test the workflow  
authenticateUser('admin', '1234');

## Conclusion

Promises in JavaScript provide an effective way to handle asynchronous operations with cleaner syntax and better error handling. They are ideal for tasks like API calls, file operations, and other asynchronous workflows.