**JavaScript Promises**

A promise is use to handle asynchronous operation elegantly, it is a object that represent the eventual completion or failure of a asynchronous operation.

A promises is a object that encapsulate the result of asynchronous operation.

**State of promises**

A promise object have 3 states

**Pending**

The initial state when the promise is created and the asynchronous operation is still going.

**Fulfilled**

The state when the asynchronous operation completed with a value.

**Rejected**

The state when the asynchronous operation is rejected for a reason.

**Creating a promise**

Use the ‘Promise’ constructor to create a new promise.

The constructor takes a function with two parameter ‘resolve’ and ‘reject’ which is provided by javascript at runtime

Inside this function, we call `resolve (value)` when it is successful or `reject (error)` when it is fails.

**Consuming promises**

Use **`.then ()`** to handle the fulfilled state and provide a callback function that takes the result.

Use **`.catch()`** to handle the rejected state and provide a callback function that takes the error.

**finally ()**

The finally () will be executed once the promise is settled whether it is fulfilled or rejected.

This means that the code inside the finally () block will run regardless of whether there was success or failure in the preceding promise chain.

**Common use cases**

**Cleanup operations**

We can use finally() to perform cleanup tasks, such as closing resources or releasing allocated memory, regardless of the outcomes of the asynchronous operations.

**Loading spinners**

In user interfaces, we might use finally() to stop loading spinners or perform other Ui-related actions that should regardless of success or failure.

**async/await**

Es2017 introduced the async/await keywords that build on top of promises, allowing us to write asynchronous code that looks more like synchronous code and is more readable.

The async keyword allows us to define a function that handles asynchronous operations. To define async function, we place the async keyword in front of the function keyword.

Asynchronous functions execute asynchronously via the event loop.it always return a promise.

The await keyword to wait for a promise to settle either in a resolved or rejected state. We can use the await keyword only inside async function.

**try{} catch{}**

try catch is use to handle error in controlled manner during the execution of a program

when javascript encounters error, it could lead to the termination of the entire application.

try/catch provides a way to catch and handle errors, preventing the application from crashing.

try {

// code may cause error

} catch(error){

// code to handle error

}

The try block contains code where the potential error may occurs

The catch block is executed if exception is thrown inside the try block. It takes error object as parameter allowing us to inspect and handle the specific error.

We can nest try/catch blocks to handle errors at different level of our code.

try {

// Outer try block

try {

// inner try block

// Code that may cause an error

} catch (innerError) {

// handle inner error

}

} catch (outerError) {

// handle outer error

}

try catch finally

the finally block contains code that will be executed regardless of whether exception occurred in the try block or not.

It is often used for cleanup operations, releasing resources, closing connections that should be performed whether error occurs or not.

The finally block is executed after the try or catch block regardless of whether a exception occurred.

try {

// Code that may cause an error

} catch (error) {

// Code to handle the error

} finally {

// Optional: Code that will run regardless of whether there was an error or not

}

throw

the throw statement allow us to throw exception. This can be helpful in scenarios where we , as a developer, detect a error or a exceptional condition in our code and want to disrupt the normal flow of execution.

throw expression

function divide(x, y) {

    if (y === 0) {

      throw "Cannot divide by zero!";

    }

    return x / y;

  }

  try {

    let result = divide(10, 0);

    console.log(result); // This line won't be executed if an exception is thrown

  } catch (error) {

    console.error("Error:", error); // Output: Error: Cannot divide by zero!

  }

where expression can be any value such as string,number,object or error object that we want to use to describe the exception.

1. Throw statement to throw exception
2. Throw statement to throw instance of the error class

Throw state to throw exception