Promises

Promises in JavaScript provide a cleaner and more structured way to handle asynchronous operations compared to traditional callbacks.It has three states: pending, fulfilled, or rejected.

1. **Creating a Promise(producing)** : You create a new Promise object using the **Promise** constructor. This constructor takes a function as an argument, which in turn takes two parameters: **resolve** and **reject**. Inside this function, you perform your asynchronous operation, and when it's completed, you call **resolve** with the result or **reject** with an error if it fails.

```
//promises creation
var promises=new Promise(function (resolve,reject){
var a=100;
if(a==10){
resolve("a is 10")
}
else{
reject("a is not 10")
}
});
```

2. **Consuming a Promise**: You consume a promise using the **then** method, which takes two optional parameters: a callback function to handle the resolved value, and a callback function to handle any errors.

```
//print the response
promises.then((val)=>{
console.log(val)
}).catch((err)=>{
```

```
  console.log(err)
  })
```

Promises also forming chain method which inturns make code readability difficult in order to avoid this

```
let add = (val) =>
  new Promise((resolve, reject) => {
    resolve(val + 10);
  });

let sub = (val) =>
  new Promise((resolve, reject) => {
    resolve(val - 10);
  });

let mul = (val) =>
  new Promise((resolve, reject) => {
    resolve(val * 5);
  });

let div = (val) =>
  new Promise((resolve, reject) => {
    resolve(val / 2);
  });

add(10)
  .then((addres) => sub(addres))
  .then((subres) => mul(subres))
  .then((mulres) => div(mulres))
  .then((divres) => console.log(divres))
  .catch((error) => console.error(error));
```

Promises aysnchronous

```
let promise1= new Promise((resolve,reject)=>{
  console.log("promise 1");
  setTimeout(resolve, 2000, "promise 1 success")
})

let promise2= new Promise((resolve,reject)=>{
  console.log("promise 2");
  setTimeout(resolve, 1500, "promise 2 success")
})

let promise3= new Promise((resolve,reject)=>{
  console.log("promise 3");
  setTimeout(resolve, 1800, "promise 3 success")
})

let promise4= new Promise((resolve,reject)=>{
  console.log("promise 4");
  setTimeout(resolve, 500, "promise 4 success")
})



promise1.then((resolve)=>{console.log(resolve)})
promise2.then((resolve)=>{console.log(resolve)})
promise3.then((resolve)=>{console.log(resolve)})
promise4.then((resolve)=>{console.log(resolve)})
```

//convert synchronous to aynchronous

```
promise1
```

```
    .then((result) => {
      console.log(result);
      return promise2;
    })
    .then((result) => {
      console.log(result);
      return promise3;
    })
    .then((result) => {
      console.log(result);
      return promise4;
    })
    .then((result) => {
      console.log(result);
    });
```

Error handling methods is js

In JavaScript, try...catch statements are used to handle exceptions (errors) that occur in your code. By using these statements, you can ensure that your code continues to run even if an error occurs.

Error – It is an object that is created to represent a problem that occurs offten with userinput or establishing a connection

Why we need try catch

```
// case -1 ------> both statements will execute
console.log("hi hello");


console.log("you have reached the end");
```

```
// case -2 ------>It interupts the program from 163 line to end because console.lag is not a method it
console.lag("hi hello");


console.log("you have reached the end");
```

To overcome this we need try catch methods

Syntax

```
try {
  // Code that may throw an error
} catch (err) {
  // Code to handle the error
} finally {
  // Code that will always run, regardless of error
}
```

Explanation

try block: Contains the code that may throw an error. If no errors occur, the code inside the catch block is skipped.

catch block: Contains code that will execute if an error occurs in the try block. The err parameter contains the error object.

finally block (optional): Contains code that will run after the try and catch blocks, regardless of whether an error was thrown or not. This block is often used for cleanup operations.

```
//try block contains error catch block takes one parameter err means error it will store the error caused by try
try{

    console.lag("hi hello");
}
catch(err){
    console.log(err);
}
```

We can also use

console.error(err)

```
try{

    console.lag("hi hello");
}
catch(err){
   console.error(err);
}
finally{
    console.log("this will always run");
}

console.log("you have reached the end");
```

Throw statement in js

The throw statement is used to raise an exception in JavaScript. When an exception is thrown, the normal flow of code execution is stopped, and control is passed to the nearest enclosing catch block. If no catch block is found, the script will terminate.

```
num = window.prompt("entry");
try {
  if (num <= 0) {
    throw new Error("The number must be positive.");
  }
} catch (err) {
  console.error(err);
}

console.log("you have reached the end");
```

This program prompts the user to enter a number, checks if the number is positive, and handles errors if the number is not positive. It uses a try...catch block to manage potential exceptions and ensures that a final message is logged to indicate the end of the program.

Prompting the User:

num = window.prompt("entry");

Error Handling with try...catch:

try {

  if (num <= 0) {

    throw new Error("The number must be positive.");

  }

```
} catch (err) {
   console.error(err);
}
```

Try Block:

- if (num <= 0) { throw new Error("The number must be positive."); }
    - **Condition**: Checks if the input num is less than or equal to 0.
    - **Error Handling**: If the condition is true, an error is thrown with the message "The number must be positive."

Catch Block:

- catch (err) { console.error(err); }
    - **Function**: Catches the error thrown in the try block.
    - **Error Logging**: Uses console.error(err) to log the error object to the console, which includes the error message and stack trace.

Final Log Statement:

console.log("you have reached the end");

Purpose: To indicate that the program has reached its end, regardless of whether an error occurred.

Types of errors

1. Syntax Errors

- **Description**: Occur when the code contains invalid syntax, which prevents the script from being parsed correctly.

```
if (true {
  console.log("This is a syntax error");
}
```

2. Reference Errors

- **Description**: Occur when trying to reference a variable that is not declared.

```
console.log(nonExistentVariable);
```

3. Type Errors

- **Description**: Occur when an operation is performed on a value of an inappropriate type

```
let num = 5;
num.toUpperCase(); // TypeError: num.toUpperCase is not a function
```

4. Range Errors

- **Description**: Occur when a numeric variable or parameter is outside its valid range

```
function createArray(size) {
  if (size > 100) {
    throw new RangeError("Array size is too large");
  }
  return new Array(size);
}
createArray(101);
```

5. Eval Errors

- **Description**: Occur due to improper use of the eval() function. This error type is rarely encountered and is primarily included for backward compatibility.

```
eval("alert('Hello)");// This example won't necessarily throw an EvalError in modern browsers but shows misuse of eval.
```

6. URI Errors

- **Description**: Occur when global URI handling functions are used incorrectly, such as encodeURI(), decodeURI(), encodeURIComponent(), and decodeURIComponent().

```
decodeURIComponent("%");
```