Lesson:

Promise Constructor

PW SKILLS

Topics

1. Introduction to promises
2. Importance of Javascript Promises
3. Promise Lifecycle
4. Promise constructor
5. Consuming the Promise Values
6. Promise methods
7. async and await
8. JSON.stringify
9. JSON.parse

Introduction to promises.

Promises are an important concept in modern Javascript programming. They allow us to handle asynchronous code and make it more efficient.

Javascript is single-threaded, it can execute only one code statement at a time. Some pieces of code execute immediately and some take time. The operation of fetching data from the server is not instantaneous. In that case, the execution thread would be blocked, leading to a bad user experience due to slow loading. We can solve these problems through Promises.

A promise is an object that represents a value that may not be available yet but will be available at some point in the future. Promises are a way of handling asynchronous code, which means code that runs in the background while another code is executing. With promises, we can write code that waits for the completion of an asynchronous task before moving on to the next task.

Importance of Javascript promises

Before looking into the implementation of promises it is important to know the importance of promises.

Here are some of the reasons why promises are important.

- Promises are an **effective way to handle asynchronous code in Javascript.** With promises, we can write code that **waits for the completion of an asynchronous task before moving on to the next task.** This leads to more efficient and responsive code, as well as a better user experience.
- In one of the previous lectures, we looked into callback hell, where multiple nested callbacks make code hard to read and debug. Promises provide a cleaner and more manageable way to handle asynchronous code than traditional callback functions.
- Promises can be chained together to handle multiple asynchronous tasks in a more readable and manageable way. This can lead to more efficient and maintainable code, making it easier to debug and improve over time.

- Promises come with an inbuilt error-handling mechanism, we can handle both expected and unexpected errors in a consistent way. This makes it easier to identify and debug errors in your code.
- Promises are widely used. This means that developers can use promises in their code regardless of the libraries or frameworks they are using.

The most effective way to understand javascript promises is by relating them to a real-life promise.

If you take a promise from your friend that he will get you chocolate, then here are the possible conditions.

1. You are waiting for your friend to come. This means the promise is in a **PENDING** state.
2. Your friend bought you chocolate. This means the promise is in a **FULFILLED** state.
3. Due to any reason, your friend failed to get a chocolate, maybe the shop was closed. This means the promise goes to the **REJECTED** state.

In the same way, if you are performing any asynchronous task that might get some resources, it is a promise. At first, the promise remains in the PENDING state because no response has been received. After some time, when the response is received, there are two possible cases:

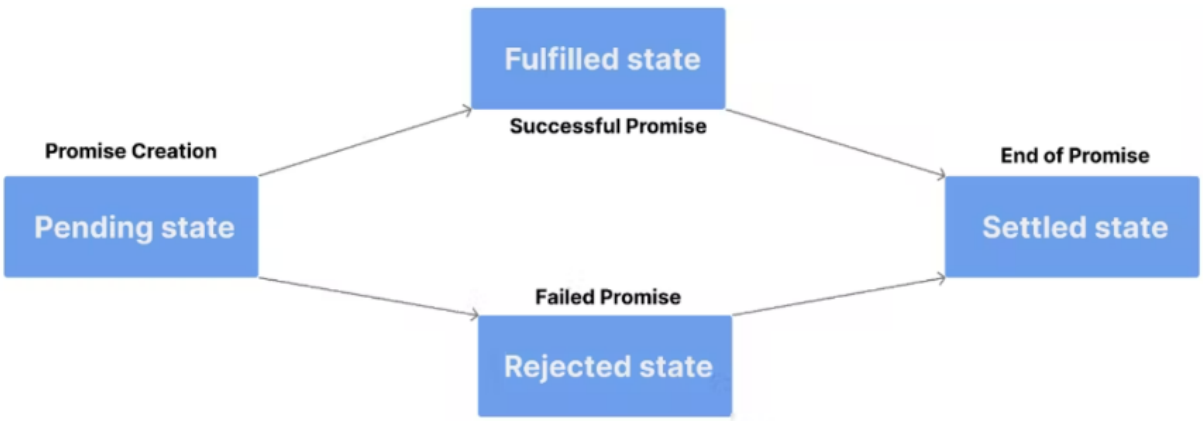
1. We got a response i.e., the promise goes to the FULFILLED state.
2. An error occurred, and the promise was REJECTED without receiving any response.

Promise Lifecycle.

- The lifecycle of promises consists of 4 stages.

1. Pending.
2. Resolved.
3. Rejected.
4. Settled.

- Once a promise is created, it enters the pending state. While a promise is in the pending state, the outcome of the asynchronous operation is still unknown. A promise remains pending until it is either resolved or rejected due to failure of the async operation.
- The resolved state specifies that the asynchronous operation has been completed successfully and the promise has a resolved value.
- The rejected state indicates that the asynchronous operation has failed and the promise has a rejected value.
- A promise's settled state refers to the final state of the promise after it has been fulfilled or rejected.



Promise constructor.

A promise constructor is used to create a new Promise.

Syntax:

```javascript
new Promise(function (resolve, reject) {
  // Asynchronous operation
});
```

The Promise constructor takes a function as its argument, which in turn takes two parameters, resolve and reject. These parameters are functions that are used to set the state of the Promise.

Let's look at an example demonstrating the states of Promise using the Promise constructor.

Now, let's write a promise which enters the resolved state if the random number generated is greater than 0.5 and the promise gets rejected if the random number generated is lesser than 0.5.

Let's write this inside an HTML document so we could visualize the states of promise more clearly.

JavaScript

```
const newPromise = new Promise((resolve, reject) => {
  let randomNumber = Math.random();
console.log(randomNumber);

  if (randomNumber > 0.5) {
    resolve("The Promise is resolved. The number is greater than
0.5");
  } else {
    reject("The Promise is rejected. The number is lesser than
0.5");
  }
});

console.log(newPromise);
```

OUTPUT:

- If the random number generated is greater than 0.5 the promise would be resolved.

```
0.9021438740851733                                    index.html:15
                                                      index.html:24
▼ Promise {<fulfilled>: 'The Promise is resolved. The number is greater than
0.5'} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: "The Promise is resolved. The number is greater than 0.5
```

- If the random number generated is greater than 0.5 the promise would be rejected.

```
0.2854349416352364                                    index.html:15
                                                      index.html:24
▼ Promise {<rejected>: 'The Promise is rejected. The number is lesser than 0.
5'} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "rejected"
    [[PromiseResult]]: "The Promise is rejected. The number is lesser than 0.5"


⊗ ▼Uncaught (in promise) The Promise is rejected. The number is   index.html:20
lesser than 0.5
    (anonymous) @ index.html:20
    (anonymous) @ index.html:13
```

Consuming the Promise Values.

Promises will for sure reach one state or the other of the promise lifecycle. Consuming a promise simply means taking the value obtained from the promise (the resolved or rejected value) to process another operation.

We have three methods to consume the promise values.

1. .then().
2. .catch().
3. .finally().

.then()

The then method allows you to specify a function that should be called when a Promise is fulfilled.

JavaScript

```
let newPromise = new Promise((resolve, reject) => {
  let randomNumber = Math.random();

  console.log(randomNumber);

  if (randomNumber > 0.5) {
    resolve("The Promise is resolved. The number is greater than
0.5");
  } else {
    reject("The Promise is rejected. The number is lesser than
0.5");
  }
});

newPromise.then((result) => console.log(result));
```

When the promise is resolved, it logs the resolve message onto the console.

```
0.7378537231286597                                    index.html:15
The Promise is resolved. The number is greater than 0.5   index.html:24
```

As the .then() method only handles the resolved state, when the promise is rejected the message will not be logged onto the console.

```
0.1766906542682738                                              index.html:15
▶Uncaught (in promise) The Promise is rejected. The number is   index.html:1
lesser than 0.5
```

.catch()

To handle the rejected or unsuccessful state, we have the .catch() method. It allows us to specify what should happen when a promise is rejected so that we can handle the error appropriately in our code.

JavaScript

```
let newPromise = new Promise((resolve, reject) => {
   let randomNumber = Math.random();
   console.log(randomNumber);

   if (randomNumber > 0.5) {
     resolve("The Promise is resolved. The number is greater than 0.5");
   } else {
     reject("The promise is rejected. The number is lesser than 0.5");
   }
 });

 newPromise.then((result) => console.log(result))
           .catch((error) => console.log(error));
```

```
0.31937301866457557                                             index.html:15
The Promise is rejected. The number is lesser than 0.5          index.html:25
▶Uncaught (in promise) The Promise is rejected. The number is   index.html:1
lesser than 0.5
```

.finally()

The finally() method is used to specify a function that is executed when the promise is settled (i.e., either resolved or rejected).

JavaScript

```
let newPromise = new Promise((resolve, reject) => {
  let randomNumber = Math.random();
  console.log(randomNumber);
```

```
        if (randomNumber > 0.5) {
          resolve("The Promise is resolved. The number is greater
than 0.5");
        } else {
          reject("The Promise is rejected. The number is lesser
than 0.5");
        }
      });

      newPromise
        .then((result) => console.log(result))
        .catch((error) => console.log(error))
        .finally(() => console.log("The Promise is settled"));
```

```
0.9318476849437818                                    index.html:15
The Promise is resolved. The number is greater than 0.5  index.html:24
The Promise is settled                                index.html:26
```

```
0.3626415714708784                                    index.html:15
The Promise is rejected. The number is lesser than 0.5  index.html:25
The Promise is settled                                index.html:26
▶Uncaught (in promise) The Promise is rejected. The number is  index.html:1
lesser than 0.5
```

Promise Methods

promise.all()

The Promise.all method is used when you have multiple promises and you want to wait for all of them to resolve before performing some action. It takes an iterable of promises as input, such as an array, and returns a new promise that resolves when all of the input promises have resolved.

Here's a step-by-step explanation of how **Promise.all** works:

First, you need to create an array of promises that you want to wait for. Each promise in the array can represent an asynchronous operation, such as an API call or data fetch.

script.js

```javascript
const promise1 = fetch('https://fakestoreapi.com/products/1');
const promise2 =
fetch('https://fakestoreapi.com/products/categories');
const promise3 =
fetch('https://fakestoreapi.com/products/jewelery');

const promises = [promise1, promise2, promise]
```

Once you have created the array of promises, you can use the **Promise.all()** method to wait for all of the promises to resolve.

script.js

```javascript
Promise.all(promises).then(results=> {


     // code to execute when all promises are resolved
     console.log(results)
}).catch(error=>{
     // code to handle andy error
     console.log(error);
})
```

In the then block the results will contain an array of resolved values from all the promises. The order of the values in the results array corresponds to the order of the promises in the original array.

promise.any()

Promise.any() is a method that takes an iterable (such as an array) of promises and returns a new promise. This new promise is fulfilled with the value of the first promise in the iterable that successfully completes (fulfills).

script.js

```javascript
const promises = [
   new Promise((resolve, reject) => setTimeout(() =>
reject("Error 1"), 1000)),
   new Promise((resolve, reject) => setTimeout(() =>
resolve("promise 2"), 2000)),
   new Promise((resolve, reject) => setTimeout(() =>
resolve("promise 3"), 3000)),
];

const anyPromise = Promise.any(promises);

anyPromise.then(value => {
   console.log("The first fulfilled promise is", value);

})
```

Console

```
The first fulfilled promise is promise 2
```

When all the promises in the provided iterable are rejected, the resulting rejection is represented by an AggregateError. This AggregateError contains an array of rejection reasons in its errors property.

script.js

```javascript
const promises = [
   new Promise((resolve, reject) => setTimeout(() =>
reject("Error 1"), 1000)),
   new Promise((resolve, reject) => setTimeout(() =>
reject("Error 2"), 2000)),
   new Promise((resolve, reject) => setTimeout(() =>
reject("Error 3"), 3000)),
];
```

```
const anyPromise = Promise.any(promises);

anyPromise.then(value => {
   console.log("The first fulfilled promise is", value);
}).catch((aggregationError) => {
   console.log(aggregationError.errors)
});
```

Console

```
▶ Array(3) [ "Error 1", "Error 2", "Error 3" ]
```

promise.resolve()

Promise.resolve() is a method that creates a promise that is already resolved. This means that the promise will not be pending, and it will not be rejected. Instead, it will immediately resolve with the value that is passed to Promise.resolve().

For example, the following code creates a promise that is resolved with the value "Hello, world!":

JavaScript

```
const promise = Promise.resolve("Hello, world!");
```

This promise can then be used with the then() method to chain together asynchronous operations. For example, the following code uses the then() method to log the value of the promise to the console:

JavaScript

```
const promise = Promise.resolve("Hello, world!");

promise.then(value => {
  console.log(value);
});
```

When the promise is resolved, the then() method will be called with the value of the promise. In this case, the value of the promise is "Hello, world!", so the then() method will log this value to the console.

promise.reject()

Promise.reject() is a method that creates a promise that is already rejected. This means that the promise will not be pending, and it will not be resolved. Instead, it will immediately reject with the reason that is passed to Promise.reject().

For example, the following code creates a promise that is rejected with the reason "Error"

```javascript
const promise = Promise.reject("Error");
```

This promise can then be used with the catch() method to handle the rejection. For example, the following code uses the catch() method to log the reason of the rejection to the console:

```javascript
const promise = Promise.reject("Error");

promise.catch(reason => {
  console.log(reason);
});
```

When the promise is rejected, the catch() method will be called with the reason for the rejection. In this case, the reason for the rejection is "Error", so the catch() method will log this reason to the console.

async and await

We all know that handling asynchronous tasks without blocking the main thread is very important. However, writing and understanding asynchronous code can be difficult. Async and await is a syntax provided to make asynchronous code more readable and in a way that looks and behaves more like synchronous code.

With async/await, we can mark a function as asynchronous using the "async" keyword, and use the "await" keyword to wait for the completion of asynchronous tasks. This makes it easier to read and write asynchronous code, as well as handle errors more effectively.

Let's look at the async/await in depth in this lecture.

What is the need for async-await?

We know that writing asynchronous code using callbacks or promises can be complex, hard to read, and error-prone, particularly when dealing with complex tasks that involve multiple asynchronous operations. Additionally, nested callbacks or chained promises can lead to what's known as "callback hell" which can be difficult to debug and maintain.

Here are some of the benefits of using the async and await keywords:

- Async and await make asynchronous code more readable and maintainable.
- Async and await make it easier to reason about the flow of control in an asynchronous program.
- Async and await make it easier to test asynchronous code.

So, it is very important to understand how to use async-await to write more easy asynchronous code.

Async

The Async keyword is used to mark a function as asynchronous. An asynchronous function is a function that returns a promise, which represents the eventual completion of the operation performed by the function.

Understanding the async keyword is easier linked to the day-to-day tasks we do such as laundry. Imagine you need to do laundry. You can start the washing machine and then do other things while the machine is running. The washing machine is performing a task asynchronously in the background, while you are free to do other tasks.

This is exactly how asynchronous tasks are done in Javascript as well. Async is to make a function work without the need of freezing the complete program.

The async keyword is used before the function keyword in the declaration.

```
Unset
async function functionName() {
  // Asynchronous operation
  // ...
  // Return a promise/value
}
```

The "async" keyword is used to indicate that the function is asynchronous, and the function body can contain one or more asynchronous operations. The function should return a promise that will resolve the result of the asynchronous operation.

```
JavaScript
async function printResult() {
  return "Hello";
}

console.log(printResult());
```

```
▼ Promise i                                          index.html:17
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: "Hello"
```

When we run the above code, it will log a Promise object to the console instead of the string "Hello". This is because the "printResult" function is declared as an asynchronous function and therefore it returns a Promise that resolves to the value "Hello".

If we need to print the value then we must consume the promise that we have seen in the previous lecture.

```javascript
async function printResult() {
  return "Hello";
}

printResult().then((result) => console.log(result));

// OUTPUT: Hello
```

Await

The "await" keyword is used to wait for the completion of an asynchronous operation inside an asynchronous function. It can only be used inside an asynchronous function that is marked with the "async" keyword.

When we use the "await" keyword, the function execution is paused until the Promise returned by the asynchronous operation is resolved or rejected. The resolved value of the Promise is then returned, allowing us to continue executing the function with the resolved value.

```javascript
async function printHelloAfterThreeSeconds() {
  let data = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Printing: Hello");
    }, 3000);
  });

  let result = await data;
// Wait until the asynchronous operation is resolved : 3 seconds

  console.log(result);
}

printHelloAfterThreeSeconds();

// OUTPUT: [After 3 seconds] Printing: Hello
```

In the above code, the "printHelloAfterThreeSeconds" creates a Promise using the "setTimeout" function to simulate an asynchronous operation that takes 3 seconds to complete. The Promise is then awaited using the "await" keyword to wait for the operation to complete.

When the Promise resolves after 3 seconds, the resolved value "Printing: Hello" is stored in the "result" variable. Finally, the value of "result" is logged to the console, which outputs "Printing: Hello" after 3 seconds.

JSON.stringify

JSON.stringify() is a built-in JavaScript method used to convert a JavaScript object or value into a JSON-formatted string.

```JavaScript
let objectIs = { name: "nasikh", age: 20 };
let val = JSON.stringify(objectIs);
console.log(val)
typeof(val)


//output:

‘{"name":"nasikh","age":20}’
'string'
```

As you can see in the above example, the object has been converted into a string.Common usecase,

- **Data Serialization:** When you want to send data from a client (e.g., a web browser) to a server, you need to convert JavaScript objects or data structures into a JSON string for transmission.

- **Storing Data:** You can use JSON.stringify to convert complex data structures, such as user preferences or settings, into a JSON string and store it in a file, localStorage or a database.

JSON.parse

JSON.parse is a built-in JavaScript function that is used to parse a JSON-formatted string and convert it into a JavaScript object. It essentially does the opposite of JSON.stringify, which converts JavaScript objects to JSON strings. JSON.parse is commonly used when you receive data in JSON format, and you need to work with it as a JavaScript object.

```javascript
// previously stringified object.
let stringifiedObject = '{"name":"nasikh","age":20}';

let val = JSON.parse(stringifiedObject);
console.log(val)
typeof(val)


//output:

{name: 'nasikh', age: 20}
'object'
```

The stringified object has been converted back to object.

Common use case,

- **Data Deserialization:** When you receive data from a server or another source in JSON format, you can use JSON.parse to convert the JSON string into a JavaScript object that you can work with in your code.
- **Reading Data from Files or Databases:** When you store data as JSON strings in files, localStorage or databases, you can use JSON.parse to read and convert that data into usable JavaScript objects.
- **Working with API Responses:** Many web APIs return data in JSON format. You can use JSON.parse to convert the JSON response into a JavaScript object that you can manipulate and display in your web application.

Usecase along with Promise

```javascript
// Simulate an asynchronous data retrieval operation
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = '{"name": "John", "age": 30}';
      resolve(data);
    }, 1000); // Simulating a delay of 1 second
  });
}
```

```
 let responseData;
 function parseData(data){
   responseData = JSON.parse(data) //parsing the data
   console.log(responseData)
   console.log('user Name is', responseData.name)
   console.log('user Age is', responseData.age)
 }

 fetchData().then((data)=> parseData(data))
```

```
//output
{ name: 'John', age: 30 }
user Name is John
user Age is 30
```