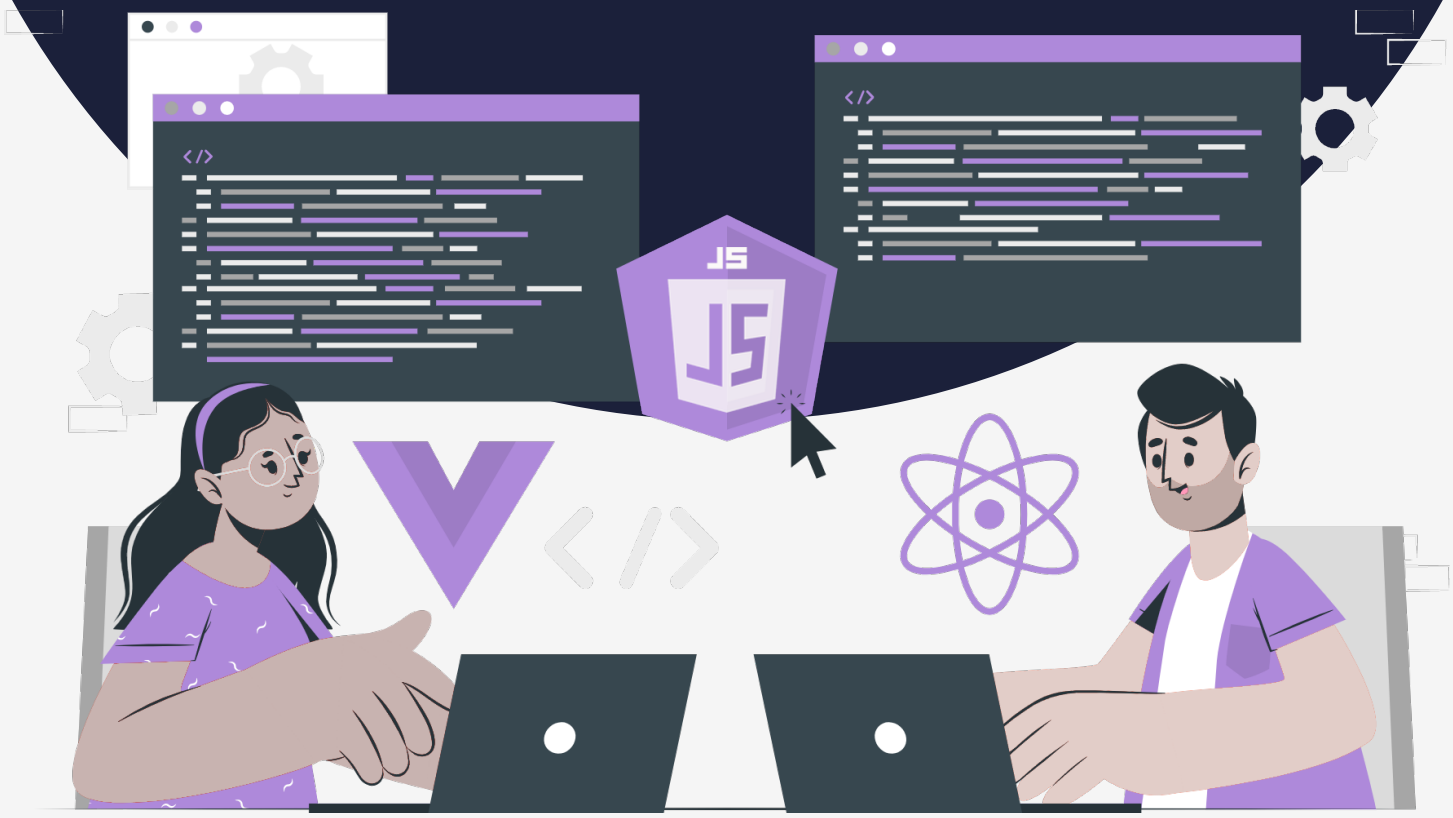
Lesson:

Promise Constructor



PW SKILLS

Topics Covered:

1. Introduction to promises.
2. Importance of Javascript Promises.
3. Understanding promises.
4. Promise Lifecycle.
5. Promise constructor.
6. Consuming the Promise values.

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

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 the previous lecture, 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 a 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. We got an error and no response the promise got REJECTED.

Promise Lifecycle.

The lifecycle of promises consists of 4 stages.

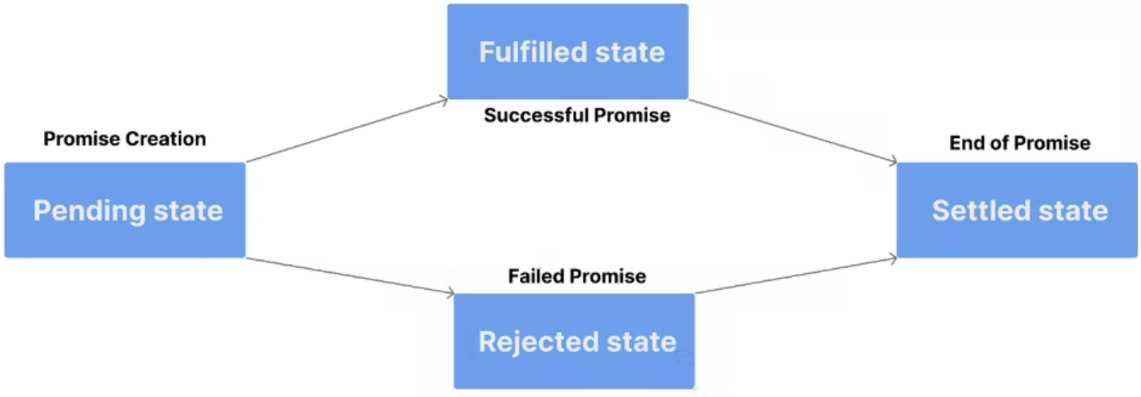
1. Pending.
2. Resolved.
3. Rejected.
4. Settled.

- 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:

```
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.

Math.random() is a built-in function in JavaScript that generates a random decimal number between 0 (inclusive) and 1 (exclusive). The function returns a random number each time it is called, which can be used for a variety of purposes.

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.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <!— JS Starts —>
    <script>
      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");
        }
      });

      console.log(newPromise);
    </script>

    <!— JS Ends —>
  </body>
</html>
```

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.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <!— JS Starts —>

    <script>
      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));
    </script>

    <!— JS Ends —>
  </body>
</html>
```

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 handle the error appropriately in our code.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <!— JS Starts —>

    <script>
      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));

    </script>

    <!— JS Ends —>
  </body>
</html>
```

```
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).

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <!— JS Starts —>

    <script>
      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"));

    </script>

    <!— JS Ends —>
  </body>
</html>
```

```
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
```