PROMISES

Promises are a way to handle asynchronous operations in JavaScript in a more organized and manageable manner compared to using plain callbacks. They provide a cleaner syntax and a more structured approach for dealing with asynchronous tasks.

A promise represents the eventual completion or failure of an asynchronous operation, and it allows you to attach callbacks to handle the success or failure of that operation. Promises have three states: **pending, fulfilled, or rejected.**

How promises work:

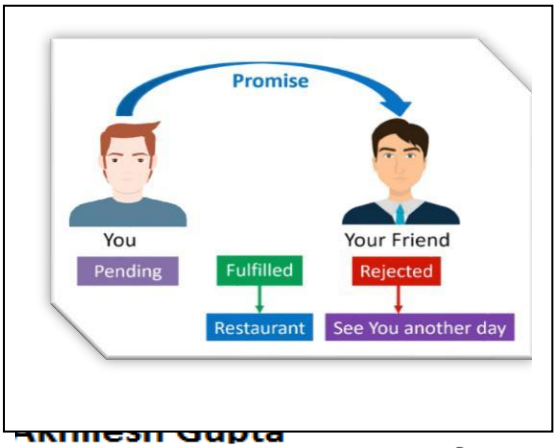
- **Creation:** You create a promise using the Promise constructor. Inside the constructor, you pass a function that takes two arguments: resolve and reject. These are functions that you call to indicate whether the asynchronous operation succeeded (resolve) or failed (reject).
  e.g.
  const myPromise = new Promise**((resolve, reject) => {**
  Perform Asynchronous operation
  If successful, call `resolve(value)`
  If failed, call `reject(error)`
  });

- **Asynchronous operation:** Inside the promise constructor function, you perform your asynchronous operation (like fetching data from a server or reading a file).

- **Resolution:** When the asynchronous operation completes successfully, you call the resolve function with the result of the operation.

- **Rejection:** If the asynchronous operation fails, you call the reject function with an error indicating the reason for the failure.

- **Consumption:** After creating the promise, you can attach callbacks using the **.then method**. The .then method takes two optional arguments: a success callback and a failure callback. These callbacks will be invoked when the promise is resolved (successfully completed) or rejected (encountered an error), respectively.



Note:

Additionally, promises can be chained together, allowing you to perform multiple asynchronous operations sequentially.

A promise has three states:

e.g.-1
```
<script>
let prom = new Promise((resolve,reject)=>{
});
console.log(typeof (prom));
</script>
```

.then() method: The .then() method in JavaScript is used to handle the result of a promise. When you create a promise, you can attach .then() to it to specify what should happen after the promise is resolved (fulfilled) or rejected.

Syntax

promise.then(**onFulfilled**, **onRejected**)

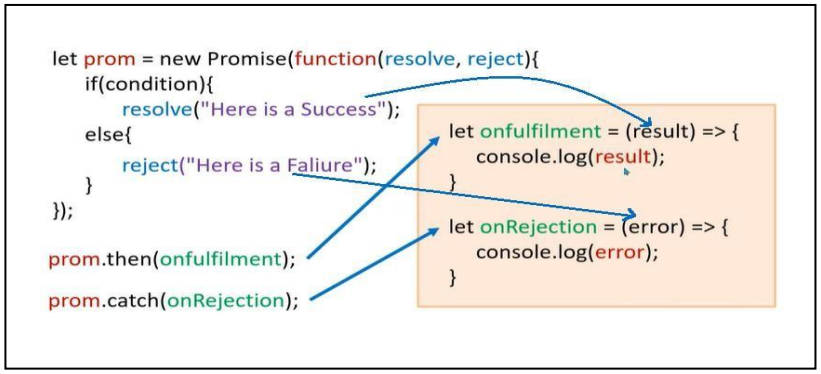
- **promise:** This is a promise object to which you're attaching the .then() method.
- **onFulfilled:** The onFulfilled function, also known as a success handler, is a function passed as the first argument to the .then() method. It gets executed if the promise is successfully resolved. **The onFulfilled function takes one argument, which is the resolved value of the promise.**

  e.g.
  ```
  let onfulfilment = (result) => {
      console.log("fulfilment");
  }
  ```
- **onRejected:** The onRejected function, also known as a rejection handler, is an optional function passed as the second argument to the .then() method. It gets executed if the promise is rejected, meaning the asynchronous operation associated with the promise encountered an error. **The onRejected function takes one argument, which is the reason for the promise rejection.** This reason is typically an error object or an error message.

  e.g.
  ```
  let onRejection = (error) => {
      console.log("Rejection");
  }
  ```

e.g.
```
<script>
  let condition = true ;
  let prom = new Promise(function (resolve, reject) {
    if (condition) {
      resolve("Here is a success");
    }
    else {
      reject("Here is a failure");
    }
  });

  let onfulfilment = (result) => {
    console.log("fulfilment");
  }
  let onRejection = (error) => {
    console.log("Rejection");
  }
  prom.then(onfulfilment);
```



```
  prom.catch(onRejection);
</script>
```

.catch() method: The .catch() method is used to handle errors or rejections that occur in a promise chain. It's a shorthand for specifying only the rejection handler (onRejected function) without the need to explicitly provide a success handler (onFulfilled function) using the .then() method.

```
<script>
  let condition = true ;
  let prom = new Promise(function (resolve, reject) {
    if (condition) {
      resolve("Here is a success");
    }
    else {
      reject("Here is a failure");
    }
  });

  let onfulfilment = (result) => {
    console.log("fulfilment " + result );
  }
  let onRejection = (error) => {
    console.log("Rejection " + error);
  }

// prom.then(onfulfilment);
  // prom.catch(onRejection);

  prom.then(onfulfilment).catch(onRejection);
</script>
```

```
x =  fetch('https://jsonplaceholder.typicode.com/users')
```

e.g.

```
<!DOCTYPE html>
<html lang="en">

<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>Callback Example</title>
</head>

<body>
 <h1>User Data</h1>
 <button id="fetchButton">Fetch User Data</button>
 <div id="userData"></div>

 <script >
  function displayUserData(user) {
   const userDataDiv = document.getElementById('userData');
   userDataDiv.innerHTML = `
```

```
    <p>User ID: ${user.id}</p>
    <p>Name: ${user.name}</p>
    <p>Username: ${user.username}</p>
    <p>Email: ${user.email}</p>
  `;
  }

  function fetchUserDataFromServer(userId, callback) {
    // Replace with the actual API URL
    fetch(`https://jsonplaceholder.typicode.com/users/${userId}`)
      .then(response => {
        if (!response.ok) {
          throw new Error(`HTTP error! Status: ${response.status}`);
        }
        return response.json();
      })
      .then(data => {
        callback(data);
      })
      .catch(error => {
        console.error("Fetch error:", error);
      });
  }

  // Event listener for the "Fetch User Data" button
  const fetchButton = document.getElementById('fetchButton');
  fetchButton.addEventListener('click', function () {
    const userId = 2; // User ID from JSONPlaceholder
    fetchUserDataFromServer(userId, displayUserData);
  });
 </script>
</body>
</html>
```

Akhilesh Kumar Gupta
(Technical Training specialist)
TechCarrel LLP