

## JavaScript Promises - Full Notes

### 1. Introduction to Promises

A Promise in JavaScript represents a value that might be available now, or in the future, or never. It is used to handle asynchronous operations, making it easier to work with APIs, databases, or any operation that takes time.

### 2. Why Promises?

Before Promises, JavaScript developers relied on callbacks, which often led to 'callback hell'

Example of callback hell:

```
function getData(callback) {  
  setTimeout(() => {  
    console.log("Data fetched");  
    callback();  
  }, 2000);  
}
```

```
getData(() => {  
  console.log("Next process...");  
});
```

### 3. Creating a Promise

A Promise is created using the Promise constructor, which takes a function with two parameters: resolve (on success) and reject (on failure).

```
let myPromise = new Promise((resolve, reject) => {  
  let success = true;  
  if (success) {  
    resolve("Promise resolved successfully!");  
  } else {  
    reject("Promise rejected!");  
  }  
});
```

Promises in JavaScript have three states:

1. **Pending** – The initial state when the promise is neither fulfilled nor rejected. It is waiting for the asynchronous operation to complete.
2. **Fulfilled (Resolved)** – The state when the asynchronous operation is successful, and the promise returns a value.
3. **Rejected** – The state when the asynchronous operation fails, and the promise returns an error.

## 4. Handling Promises

### 4.1 Using .then()

Used to handle the resolved value.

```
myPromise.then((message) => {  
  console.log(message); // Output: Promise resolved successfully!  
});
```

### 4.2 Using .catch()

Used to handle rejection.

```
let failedPromise = new Promise((resolve, reject) => {  
  reject("Something went wrong!");  
});  
  
failedPromise.catch((error) => {  
  console.log(error); // Output: Something went wrong!  
});
```

### 4.3 Using .finally()

Executes whether the Promise is resolved or rejected.

```
myPromise  
  .then((message) => console.log(message))  
  .catch((error) => console.log(error))
```

```
.finally(() => console.log("Execution completed."));
```

## 5. Chaining Promises

Chaining helps avoid nested callbacks and makes code more readable.

```
new Promise((resolve) => {  
  setTimeout(() => resolve("Step 1 complete"), 1000);  
})  
  .then((result) => {  
    console.log(result);  
    return "Step 2 complete";  
  })  
  .then((result) => {  
    console.log(result);  
    return "Step 3 complete";  
  })  
  .then(console.log);
```

## 6. Promise Methods

JavaScript provides utility methods to work with multiple promises.

### 6.1 Promise.all()

Runs multiple promises in parallel and resolves when all promises are fulfilled.

```
let p1 = Promise.resolve("One");  
let p2 = new Promise((resolve) => setTimeout(() => resolve("Two"), 2000));  
let p3 = Promise.resolve("Three");
```

```
Promise.all([p1, p2, p3]).then(console.log);
```

### 6.2 Promise.race()

Resolves or rejects as soon as the first promise settles.

```
let fastPromise = new Promise((resolve) => setTimeout(() => resolve("Fast"), 1000));  
let slowPromise = new Promise((resolve) => setTimeout(() => resolve("Slow"), 3000));
```

```
Promise.race([fastPromise, slowPromise]).then(console.log);
```

### 6.3 Promise.allSettled()

Returns the result of all promises, regardless of whether they resolve or reject.

```
let pA = Promise.resolve("Resolved A");
let pB = Promise.reject("Rejected B");

Promise.allSettled([pA, pB]).then(console.log);
```

### 6.4 Promise.any()

Resolves with the first fulfilled promise and ignores rejections.

```
let pX = Promise.reject("Error X");
let pY = new Promise((resolve) => setTimeout(() => resolve("Success Y"), 2000));

Promise.any([pX, pY]).then(console.log);
```

## 7. Using Promises with async / await

`async` / `await` simplifies promise handling.

### 7.1 async function

```
async function fetchData() {
  return "Data received";
}

fetchData().then(console.log);
```

### 7.2 await

```
async function getData() {
  let response = await new Promise((resolve) =>
    setTimeout(() => resolve("Fetched Data"), 2000)
  );
  console.log(response);
}

getData();
```

### 7.3 Handling Errors with try...catch

```
async function fetchData() {  
  try {  
    let data = await Promise.reject("Network Error");  
    console.log(data);  
  } catch (error) {  
    console.log(error);  
  }  
}  
  
fetchData();
```