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. Promise methods
7. Consuming the Promise values
8. Interview point

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.

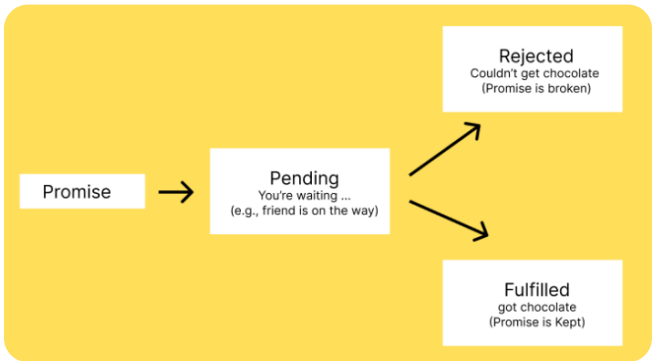
A promise is an object that represents a value that may not be available yet but will be available at some point in the future. Promises are a way of handling asynchronous code, which means code that runs in the background while another code is executing. With promises, we can write code that waits for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

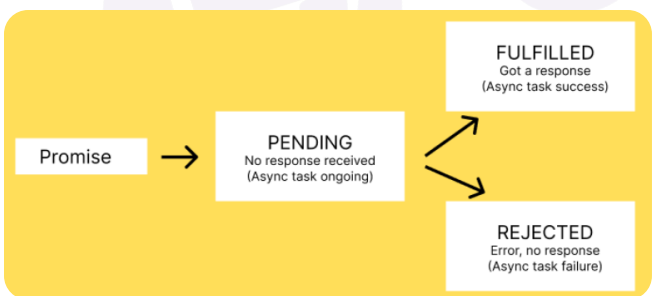


If you take a promise from your friend that he will get you chocolate, then here are the possible conditions.

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.

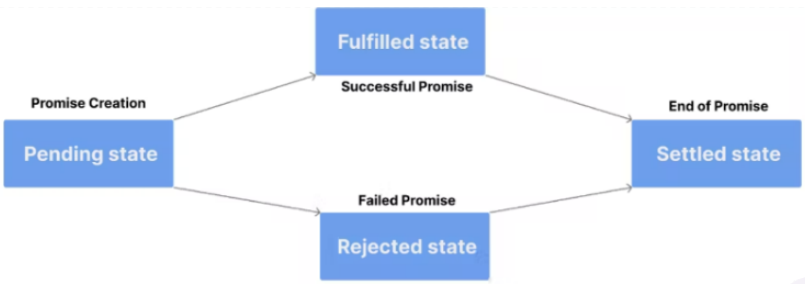


Promise Lifecycle

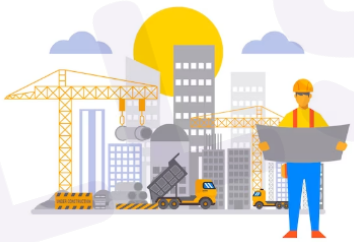
The lifecycle of promises consists of 4 stages.

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.

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.

```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
v0.5");
  }
});

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'} ⓘ
  ▶ [[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'} ⓘ
  ▶ [[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 lesser than 0.5   index.html:1
```

.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 lesser than 0.5   index.html:1
```

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 be resolved.

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);
})
```

```
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 lesser than 0.5   index.html:1
```

Promise Methods

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); // output - Hello world!
});
```

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"

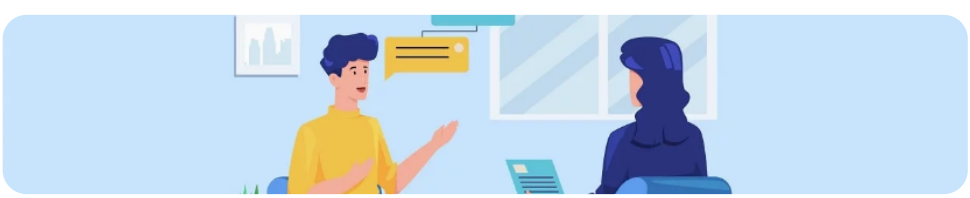
```javascript
const promise = Promise.reject("Error");
```

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); // output -  Error

});
```

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.

Interview Point



Q1.You have to create two promises, promise1, and promise2, and both will give you a number when they are ready or resolved. Your task is to create a new promise. This new promise should calculate the total sum of the two numbers from promise1 and promise2.

Ans.

sumPromise.js

```
function sumOfPromises(promise1, promise2) {
  return new Promise((resolve, reject) => {
    // Variables to store the resolved values of the promises
    let value1, value2;

    // Handler for promise1
    promise1.then(result => {
      value1 = result;
      // Check if both promises have resolved
      if (value1 !== undefined && value2 !== undefined) {
        resolve(value1 + value2);
      }
    }).catch(reject);

    // Handler for promise2
    promise2.then(result => {
      value2 = result;
      // Check if both promises have resolved
      if (value1 !== undefined && value2 !== undefined) {
        resolve(value1 + value2);
      }
    }).catch(reject);
  });
}

// Example usage:
const promise1 = new Promise(resolve => resolve(5));
const promise2 = new Promise(resolve => resolve(10));

sumOfPromises(promise1, promise2)
  .then(result => console.log("Sum:", result))
  .catch(error => console.error("Error:", error));
```

Q2. Let's say you have a special task to do, but sometimes it might not work on the first try. Imagine you have a helper, let's call it "retry." This helper takes two things:

- **A special task that you want to do (which is like a promise).**
- **Max number of times you want to try this task.**

So, you tell the helper to try this task. If it doesn't work, the helper will keep trying, but only up to the number of times you told it. Can you create this helpful function called "retry"?

Ans.

promise.js

```
// Function to retry a promise-returning function with a maximum number of attempts
function retry(promiseFn, maxAttempts) {
  // Return a new promise that will resolve or reject based on the retry logic
  return new Promise(async (resolve, reject) => {
    // Counter to track the number of attempts
    let attempts = 0;

    // Function to make attempts
    async function attempt() {
      try {
        // Attempt to execute the promise-returning function
        const result = await promiseFn();

        // Resolve the promise if successful
        resolve(result);
      } catch (error) {
        // Increment the number of attempts
        attempts++;

        // If attempts are within the limit, retry
        if (attempts < maxAttempts) {
          await attempt(); // Recursive call for the next attempt
        } else {
          // If maximum attempts reached, reject the promise with the last error
          reject(error);
        }
      }
    }

    // Start the initial attempt
    await attempt();
  });
}

// Example usage:
// retrying a function that always rejects for demonstration purposes
const exampleRetryFunction = retry(
  async () => {
    console.log("Attempting...");
    throw new Error("Simulated error");
  },
  3 // maximum attempts
);

exampleRetryFunction
  .then(result => console.log("Success:", result))
  .catch(error => console.error("Failed after multiple attempts:", error));
```

PW SKILLS

THANK YOU !