Promises in JavaScript - Detailed Notes

What is a Promise?

- A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- It provides a cleaner alternative to callbacks (avoiding "callback hell").
- Promises can be in one of three states:
  1. Pending: The initial state, not yet fulfilled or rejected.
  2. Fulfilled: The operation completed successfully.
  3. Rejected: The operation failed.

Basic Syntax:

```
let promise = new Promise(function (resolve, reject) {
    // asynchronous operation
});
```

- resolve(value): Marks the promise as fulfilled and returns value.
- reject(error): Marks it as rejected with an error.

States & Lifecycle:

- Pending: Created but not yet finished.
- Settled: Either fulfilled or rejected.

- Once a promise is settled, it cannot change again (immutable).

Example:

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Success!"), 1000);
});
```

Chaining with .then():

- .then() takes two arguments:
  - a success handler
  - an optional failure handler

Example:

```
promise
    .then((result) => {
        console.log(result);
    })
    .catch((error) => {
        console.log(error);
    });
```

- Chaining: Each .then() returns a new promise, allowing chaining.

```
fetchData()
    .then(processData)
    .then(displayData)
    .catch(handleError);
```

Error Handling with .catch():

- .catch() is syntactic sugar for .then(null, errorHandler).
- Always recommended to catch errors to avoid unhandled rejections.

Example:

```
promise
    .then((res) => {
        // success
    })
    .catch((err) => {
        // handle error
    });
```

Finalization with .finally():

- .finally() runs no matter what (fulfilled or rejected).
- Useful for cleanup tasks (like hiding loaders).

Example:

```
promise
    .then(result => console.log(result))
    .catch(error => console.log(error))
    .finally(() => console.log("Operation finished"));
```

Promise Resolution:

- You can return a value or a new promise inside .then().
- If you return a promise, the next .then() waits for it to resolve.

Example:

```
Promise.resolve(10)
    .then((num) => {
        return num * 2;
    })
    .then((result) => {
        console.log(result);
    });
```

Static Methods:

1. Promise.resolve(value):

   Returns a promise resolved with the given value.

2. Promise.reject(reason):

Returns a promise rejected with the given reason.

3. Promise.all(iterable):

Takes an array of promises. Resolves when all promises succeed. Rejects if any one promise fails.

Example:

```
Promise.all([promise1, promise2])
    .then(results => console.log(results))
    .catch(error => console.log(error));
```

4. Promise.race(iterable):

Resolves or rejects as soon as any one promise settles.

5. Promise.allSettled(iterable):

Waits for all promises to settle (fulfilled or rejected). Returns an array of results with {status, value/reason}.

6. Promise.any(iterable):

Resolves as soon as any one promise fulfills. If all reject, returns an AggregateError.

Why Use Promises?

- Cleaner asynchronous flow than nested callbacks.

- Better error propagation.

- Can compose multiple async operations.

- Standardized in ES6, well-supported.

Common Mistakes to Avoid:

- Forgetting to return a promise in .then() chains, which can break chaining.

- Using .catch() but not handling errors in inner chains properly.

- Mixing callbacks and promises-stick to one style per flow.

Async/Await (built on Promises):

Async/await is syntactic sugar over Promises for better readability.

Example:

```
async function fetchData() {
    try {
        let result = await fetch('https://api.example.com/data');
        let data = await result.json();
        console.log(data);
    } catch (error) {
        console.log('Error:', error);
    }
}
```