What is a Promise?

A **Promise** in JavaScript is an object that represents the **eventual completion (or failure)** of an **asynchronous operation**, and its resulting **value**.

✅ **Syntax:**

```
const promise = new Promise((resolve, reject) => {
  // async operation
  if (success) {
    resolve(value);   // fulfilled
  } else {
    reject(error);    // rejected
  }
});
```

🔷 **Promise States**

1. **Pending** – Initial state.
2. **Fulfilled** – Operation completed successfully.
3. **Rejected** – Operation failed.

Once a promise is **settled (fulfilled or rejected)**, it **cannot change** state.

🔷 **Basic Example:**

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


promise.then(result => console.log(result)); // Success!
```

🔷 **Chaining Promises**

You can return values from .then() to chain multiple steps.

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve(2), 1000);
})
.then(data => data * 3)
```

```
.then(data => data + 1)
.then(result => console.log(result));  // 7
```

🔷 Error Handling with .catch()

```
new Promise((resolve, reject) => {
  reject("Something went wrong");
})
.then(res => console.log(res))
.catch(err => console.log("Caught error:", err));
```

🔷 .finally() Method

Runs **no matter what** (resolved or rejected). Useful for cleanup tasks.

```
doSomething()
  .then(result => {})
  .catch(error => {})
  .finally(() => console.log("Done!"));
```

🔷 Real-World Analogy

A **promise** is like ordering a pizza:

- You **place an order** (promise is pending).
- Pizza is **delivered** (fulfilled → resolve()).
- Or they say **sorry, we can't deliver** (rejected → reject()).

🔷 Async/Await (based on Promises)

Syntactic sugar to handle promises more cleanly.

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function run() {
  console.log("Start");
  await delay(1000);
```

```
  console.log("After 1 sec");
}

run();
```

🔷 Promise Combinators

✅ Promise.all([])

Waits for **all promises** to resolve, or rejects if any fail.

```
Promise.all([p1, p2])
  .then(results => console.log(results))
  .catch(err => console.log(err));
```

✅ Promise.race([])

Returns the result of the **first settled** promise (resolved or rejected).

✅ Promise.allSettled([])

Waits for **all promises** to settle, and returns results with their status.

✅ Promise.any([])

Returns the **first fulfilled** promise (ignores rejections unless all fail).

🔷 Common Mistakes

Mistake	Why it happens
Forgetting to return inside .then()	Breaks the chain
Mixing callbacks and promises	Leads to messy code
Not handling errors	Causes unhandled promise rejections
Using await outside async	SyntaxError

🔷 Custom Promise Example

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      success ? resolve("Data loaded") : reject("Error loading data");
```

```
  }, 1000);
 });
}

fetchData()
 .then(data => console.log(data))
 .catch(err => console.log(err));
```

🔁 **Summary Table**

Concept	Description
new Promise()	Creates a promise with resolve & reject
.then()	Handles resolved value
.catch()	Handles errors
.finally()	Runs regardless of outcome
async/await	Cleaner syntax for promises
Promise.all()	Waits for all to succeed
Promise.race()	Resolves/rejects with first one
Promise.any()	Resolves with first success
Promise.allSettled()	Gives all outcomes

✅ **What is Promise Chaining in JavaScript?**

Promise chaining is a way to execute **multiple asynchronous operations in sequence**, where each .then() receives the result of the previous one.

🧠 **Basic Syntax:**

```
doSomething()
 .then(result1 => {
  return doSomethingElse(result1);
 })
 .then(result2 => {
  return doThirdThing(result2);
 })
```

```
 .catch(error => {
   console.log("Caught an error:", error);
 });
```

Each .then() returns a **new promise**, so chaining works by **passing the resolved value down the chain**.

🔁 **Example: Chaining with setTimeout**

```
new Promise((resolve) => {
 setTimeout(() => resolve(1), 1000);
})
.then((result) => {
 console.log(result); // 1
 return result * 2;
})
.then((result) => {
 console.log(result); // 2
 return result * 2;
})
.then((result) => {
 console.log(result); // 4
});
```

Every .then() receives the **return value** from the one before it.

❓ **How does JavaScript know a Promise is resolved?**

🔍 **Internals of Promise Resolution:**

1. When you create a promise:

```
new Promise((resolve, reject) => {
 // some async operation
});
```

2. The resolve() function tells JavaScript:

“This promise has completed successfully and holds a result.”

Example:

```
resolve("done");
```

3. Internally, the **JavaScript engine maintains a job queue (microtask queue)**.
    - When you resolve(), your .then() callback is **queued** to run **after the current call stack** is cleared.
4. The event loop sees the promise has settled and **executes .then() handlers**.

🔁 **Flow Diagram (Simplified):**

Promise created → pending

↓

resolve("value") called

↓

Promise moves to "fulfilled"

↓

.then() callback gets executed via microtask queue

📌 **Important Points:**

- A .then() always returns a **new promise**, which lets you chain more.
- If you return a **value**, it's passed to the next .then() as a resolved promise.
- If you return a **promise**, the next .then() waits for it to settle.
- If you **throw an error** or return Promise.reject(), the next .catch() handles it.

⚠️ **Common Mistake:**

```
doSomething()
 .then(result => {
  doSomethingElse(result); // ❌ No return!
 })
 .then(next => {
  // Won't wait for doSomethingElse to finish
 });
```

✅ Fix:

```
.then(result => {
 return doSomethingElse(result); // ✅ return the promise
})
```