Introduction to JavaScript Promises

A **Promise** in JavaScript is a way to handle **asynchronous operations**. It lets you write code that runs **after something finishes**, without getting stuck in messy nested callbacks.

Think of a Promise like a **placeholder for a value** that will be available in the future.

Why Do We Need Promises?

With callbacks, things can quickly become hard to read and maintain, especially when we have to wait for multiple things.

Example of **callback hell**:

```
doTask1(function (result1) {
  doTask2(result1, function (result2) {
    doTask3(result2, function (result3) {
      console.log("All tasks done");
    });
  });
});
```

This kind of nested code becomes difficult to manage. **Promises solve this** by allowing a cleaner, more readable structure.

Basic Promise Syntax

```
const promise = new Promise(function (resolve, reject) {
  // Do some work...
```

```
  // Call resolve(result) if successful
  // Call reject(error) if there's an error
});
```

Once a Promise is created, we can handle its result using `.then()` and `.catch()`:

```
promise
  .then(function (result) {
    // This runs if the promise was resolved
  })
  .catch(function (error) {
    // This runs if the promise was rejected
  });
```

A Simple Example: Fake Async Task

Let's create a Promise that waits for 2 seconds and then resolves.

```
function waitTwoSeconds() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve("Done waiting");
    }, 2000);
  });
}

console.log("Start");

waitTwoSeconds()
  .then(function (message) {
    console.log(message); // "Done waiting"
  })
  .catch(function (error) {
    console.log("Something went wrong");
  });
```

```
console.log("End");
```

Output:

```
Start
End
Done waiting
```

Even though the Promise is written earlier, it runs **after** the rest of the synchronous code — just like with callbacks.

Solving Callback Hell with Promises

You can chain multiple `.then()` calls instead of nesting:

```
doTask1()
  .then(function (result1) {
    return doTask2(result1);
  })
  .then(function (result2) {
    return doTask3(result2);
  })
  .then(function (result3) {
    console.log("All tasks done");
  })
  .catch(function (error) {
    console.log("Something failed", error);
  });
```

This is **much cleaner** than deeply nested callbacks.

Summary

- A **Promise** is an object representing a value that may be available now, later, or never.
- It has three states:
  - **Pending**: not yet finished
  - **Resolved**: finished successfully
  - **Rejected**: finished with an error
- Use `.then()` to handle success and `.catch()` to handle errors.
- Promises help write **cleaner async code**, especially when chaining tasks.