📌 Creating a Promise, Chaining & Error Handling in JavaScript

🧠 What is a Promise?

A **Promise** in JavaScript is like a **contract** that says:

> "I promise to return a value in the future. It might succeed (resolve) or fail (reject), but I'll let you know either way."

Promise States:

- `pending`: The operation is still ongoing.
- `fulfilled`: The operation completed successfully.
- `rejected`: The operation failed.

🎯 Use Case: Ordering Items Online

```
const cart = ["shoes", "pants", "kurta"];
```

Imagine you're buying clothes online. You:

1. **Create an order**
2. **Proceed to payment**
3. **Receive confirmation**

Let's implement this flow using **Promises**.

✨ Step 1: Consuming a Promise

```
const promise = createOrder(cart); // returns a Promise

promise.then(function (orderId) {
  proceedToPayment(orderId); // executes once order is created
});
```

☑ What's happening here?

- We call `createOrder(cart)` and expect it to return a **Promise**.
- If successful, the `.then()` callback is triggered with the `orderId`.

🏗 Step 2: Creating a Promise (Producer Code)

```
function createOrder(cart) {
  return new Promise(function (resolve, reject) {
    // Step 1: Validate cart
    if (!validateCart(cart)) {
      return reject(new Error("Cart is not Valid"));
    }

    // Step 2: Simulate DB call
    const orderId = "12345"; // Mock DB-generated ID

    // Step 3: Fulfill the promise
    resolve(orderId);
  });
}
```

💡 Key Concepts:

- `resolve`: Call this when the operation succeeds.
- `reject`: Call this when something goes wrong (e.g., invalid cart).
- `validateCart(cart)`: A hypothetical function that checks if the cart is valid.

🔍 What Happens When You Log the Promise?

```
console.log(promise); // Output: Promise {<pending>}
```

Why pending?

> Because the Promise is **asynchronous**. It hasn't completed yet. Once it's resolved or rejected, the appropriate `.then()` or `.catch()` runs.

🚨 Step 3: Handling Errors with `.catch()`

If the cart is invalid, you should **handle the failure gracefully**:

```
createOrder(cart)
  .then(function (orderId) {
    // Success
    return proceedToPayment(orderId);
  })
  .catch(function (err) {
    // Failure
    console.error("Error occurred:", err.message);
  });
```

🔗 Step 4: Promise Chaining (One After Another)

In real-world scenarios, multiple operations depend on each other — this is where **chaining** shines:

```
createOrder(cart)
  .then(function (orderId) {
    return proceedToPayment(orderId); // returns another promise
  })
  .then(function (paymentInfo) {
    console.log("Payment Response:", paymentInfo); // logs: "Payment Successful"
  })
  .catch(function (err) {
    console.error("Something went wrong:", err.message);
  });
```

> 💬 Each `.then()` receives the return value from the previous `.then()`.

🔧 Helper Functions

```
function proceedToPayment(orderId) {
  return new Promise(function (resolve, reject) {
    // Simulate success
    resolve("Payment Successful");
  });
}

function validateCart(cart) {
  // Simulate cart validation
  return cart.length > 0;
}
```

🛑 What If We Want to Continue Even After an Error?

Sometimes, we want to catch errors but continue the chain:

```
createOrder(cart)
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .catch(function (err) {
    console.warn("Early error caught:", err.message);
    return "FallbackOrderId"; // Continue with default/fallback
  })
  .then(function (paymentInfo) {
    console.log("Continuing after error:", paymentInfo);
  });
```

> ✅ The key here is: **return something** from the `.catch()` to resume the chain.

⚙️ Advanced: Multiple `.catch()` Blocks

You can place multiple `.catch()` blocks at different levels to handle errors in specific segments:

```
createOrder(cart)
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .catch(function (err) {
    console.log("Error in order creation or payment:", err.message);
  })
  .then(function () {
    // This still runs
    console.log("Executing final step regardless of previous error.");
  });
```

🧠 Final Notes

Concept	Description
`Promise`	Handles async operations in a structured way
`resolve(value)`	Call when the async task is successful
`reject(error)`	Call when the async task fails
`.then(callback)`	Handles successful results
`.catch(callback)`	Handles errors
Promise Chaining	Sequence of `.then()` calls
Error Propagation	Errors flow down the chain until caught

✅ Output Recap

```
> console.log(promise)
Promise {<pending>}

> When resolved:
"Payment Successful"

> When cart invalid:
Error: Cart is not Valid
```