Promise chaining is a technique used in JavaScript to handle asynchronous operations using Promises. A Promise represents a value that may not be available yet but will be resolved or rejected at some point in the future. Promise chaining allows you to execute multiple asynchronous operations sequentially and handle the results one after the other, making the code more readable and maintainable.

Here's an example of promise chaining:

Let's say we have a function `getUserData` that fetches user data from an API asynchronously using a Promise:

```javascript
function getUserData(userId) {
  return new Promise((resolve, reject) => {
    // Simulate an asynchronous API call to fetch user data
    setTimeout(() => {
      // Assuming data is fetched successfully
      const userData = {
        id: userId,
        name: "John Doe",
        age: 30,
        email: "john@example.com",
      };
      resolve(userData);
    }, 1000); // Simulating a delay of 1 second
  });
}
```

Now, let's use `getUserData` and chain multiple asynchronous operations to get the user data, update it, and then log the final result:

```javascript
// Using Promise chaining
getUserData(123)
  .then((userData) => {
    console.log("User data:", userData);
    return getUserOrders(userData.id); // Fetching user orders using another Promise-based function
  })
  .then((userOrders) => {
    console.log("User orders:", userOrders);
    return processOrders(userOrders); // Processing orders using another Promise-based function
  })
  .then((processedOrders) => {
    console.log("Processed orders:", processedOrders);
    console.log("All operations completed successfully.");
  })
  .catch((error) => {
    console.error("An error occurred:", error);
  });
```

In this example:

1. We call `getUserData(123)` to fetch the user data for a specific user ID.

2. In the first `.then()` block, we log the user data and then return a new Promise by calling `getUserOrders(userData.id)`. This allows us to chain the next operation, which is fetching the user's orders.

3. In the second `.then()` block, we log the user's orders and then return another Promise by calling `processOrders(userOrders)`. This allows us to chain the next operation, which is processing the orders.

4. In the third `.then()` block, we log the processed orders and handle the successful completion of all operations.

If any of the promises in the chain are rejected, the control will jump to the nearest `.catch()` block, and the error will be handled there.

Promise chaining helps avoid the "callback hell" and provides a more structured and organized way to handle asynchronous operations, making the code easier to read and maintain.