Basic JS Concept Revision for BACKEND:

Understanding map and filter in JavaScript

The map and filter methods in JavaScript are powerful tools for working with arrays. They are widely used in backend development to process data efficiently. Let's explore these methods with examples and real-life scenarios.

---

1. 'map' Method

The map method creates a new array by applying a function to each element in an existing array. It doesn't modify the original array but instead returns a new one with transformed values.

Real-life Example: Calculating Prices with Tax

Let's say you have an array of prices (e.g., prices of books) and want to add a 10% tax to each. You can use map to create a new array with the updated prices.

```
const prices = [100, 200, 300];

const pricesWithTax = prices.map((price) => price * 1.1); // Adds 10% tax


console.log(pricesWithTax); // Output: [110, 220, 330]
```

- **Explanation**: prices.map((price) => price * 1.1) iterates through each price, multiplies it by 1.1, and stores the result in pricesWithTax.

---

2. 'filter' Method

The filter method creates a new array with elements that pass a condition, leaving out any elements that don't meet that condition.

Real-life Example: Filtering Books Above a Certain Price

Imagine you want to filter out books that cost more than $150.

```
const prices = [100, 200, 300];
```

```
const expensiveBooks = prices.filter((price) => price > 150);


console.log(expensiveBooks); // Output: [200, 300]
```

- **Explanation**: `prices.filter((price) => price > 150)` checks each price, keeping only those greater than 150 in the expensiveBooks array.

Combining 'map' and 'filter'

You can chain map and filter to apply multiple transformations in one go!

Example: Calculate Price with Tax and Filter Expensive Books

Say you want to:

1. Add a 10% tax to each price.
2. Filter only the prices above 200.

```
const prices = [100, 200, 300];

const finalPrices = prices

 .map((price) => price * 1.1) // Adds 10% tax

 .filter((price) => price > 200); // Keeps prices > 200


console.log(finalPrices); // Output: [220, 330]
```

Summary

- **map** transforms each element (e.g., applying a 10% tax).
- **filter** selects elements based on a condition (e.g., only keeping prices > 200).

Using these methods, you can handle complex array transformations and filtering efficiently. Let me know if you need more examples or further clarification!

Understanding ‘Async/Await’, ‘Promises’, ‘then’, ‘try’, and ‘catch’ in JavaScript

Asynchronous programming in JavaScript enables you to handle tasks that take time (like fetching data from a server) without blocking the main thread. Here’s an explanation of these concepts with real-life examples.

---

1. Synchronous vs. Asynchronous

- **Synchronous**: Tasks happen one after another. Imagine you’re waiting in line to buy a coffee. You have to wait until the person in front finishes before you can order.
- **Asynchronous**: Tasks can happen independently, without waiting for the previous task to finish. It’s like placing your coffee order, then waiting at a table while the barista prepares it. You’re free to do other things, and you’ll get notified when your coffee is ready.

JavaScript’s **asynchronous nature** lets it handle multiple tasks without blocking the main thread, essential for a smooth user experience in web apps.

---

2. Promises

A **Promise** in JavaScript represents a task that will complete in the future, either successfully or with an error.

Example: Promise for Ordering Coffee

Let’s say you place a coffee order. The coffee shop promises to prepare it, and you’ll get notified when it’s ready or if something goes wrong.

```
const orderCoffee = new Promise((resolve, reject) => {
  let isCoffeeReady = true; // Simulating coffee preparation status
  if (isCoffeeReady) {
```

```
    resolve("Your coffee is ready!"); // Task is successfully completed
  } else {
    reject("Sorry, we're out of coffee."); // An error occurred
  }
});
```

Here:

- **resolve** represents a successful outcome (coffee ready).
- **reject** represents an error (coffee isn't available).

---

3. .then() and .catch() for Handling Promises

When we have a Promise, we can use .then() and .catch() to handle the results.

```
orderCoffee
  .then((message) => console.log(message)) // If coffee is ready
  .catch((error) => console.log(error)); // If there was an error
```

- .then() runs when the Promise is fulfilled (coffee is ready).
- .catch() runs when the Promise is rejected (coffee isn't available).

---

4. Async/Await for Cleaner Code

Using **async/await** makes handling asynchronous code easier and more readable than .then() chaining.

Example: Ordering Coffee with async/await

Let's use async/await to make the code more readable, like waiting for coffee without needing .then() or .catch().

1. **async** function: An async function is a wrapper for promises, allowing you to use await inside it.

2. **await**: await pauses the code until the Promise resolves or rejects.

```
const orderCoffeeAsync = async () => {
 try {
  const message = await orderCoffee; // Wait for coffee to be ready
  console.log(message);
 } catch (error) {
  console.log(error); // Handle any errors
 }
};

orderCoffeeAsync();
```

---

5. try and catch for Error Handling

In the async/await syntax, we use try and catch blocks:

- **try**: Place the code you want to attempt to execute.
- **catch**: Catches and handles any errors that occur in the try block.

Here's a recap of the example above:

- **try** block: await orderCoffee waits for the coffee order to be fulfilled or rejected.
- **catch** block: Runs if an error occurs, such as the coffee being unavailable.

Real-life Scenario Recap:

Imagine you:

1. Place a coffee order.
2. await lets you wait for the order without blocking you from other activities.

3. You either get notified when the coffee is ready (resolve) or if there's an issue (reject).
4. try and catch handle both outcomes.

Summary

- **Promises** represent tasks that will finish in the future.
- **.then()** and **.catch()** handle fulfilled and rejected promises, respectively.
- **async/await** makes asynchronous code more readable.
- **try and catch** are used for structured error handling.

Let me know if you need more examples or have specific scenarios in mind!