Intermediate-Hard: Custom Promise Control Flow

Objective: Understand how to control the flow of asynchronous operations using Promises.

Task: Write a JavaScript function named `processData` that simulates fetching data from an API and then processing it. This function should:

1. Simulate fetching data with a function `fetchData` that returns a Promise, which resolves with an array of numbers after a 2-second delay.
2. Write a function `analyzeData` that takes an array of numbers and returns a new Promise. This Promise should resolve with an object containing the sum and average of the numbers, but only if all numbers are positive. If any number is negative, the Promise should reject with an error message.
3. Use the `fetchData` function to get data, then process it with `analyzeData`, handling both success and failure cases properly.

Criteria for Success:

- Implement `fetchData` with a simulated delay using `setTimeout`.
- Correctly implement `analyzeData` to calculate sum and average, and handle negative numbers as described.
- Handle the resolved value and any potential errors properly.

Advanced: Sequential vs. Concurrent Execution

Objective: Compare sequential and concurrent execution patterns in handling Promises.

Task: Given an array of URLs (simulated as functions that return Promises), write two functions:

1. `fetchSequentially(urls)`: This function should fetch data from the URLs one after the other, waiting for each fetch to complete before starting the next. Measure and log the total time taken to complete all fetches.
2. `fetchConcurrently(urls)`: This function should initiate all fetches at once and wait for all of them to complete. Measure and log the total time taken to complete all fetches.

For both functions, simulate the fetch operation with a function that returns a Promise resolving after a random delay (1 to 3 seconds).

Criteria for Success:

- Implement both fetching strategies correctly.
- Use `console.time` and `console.timeEnd` to measure execution time.
- Understand and explain why the times differ between the two approaches.

Expert: Implementing a Promise Queue

Objective: Implement a queue system for handling Promises sequentially in a controlled manner.

Task: Write a class `PromiseQueue` that manages the execution of Promises sequentially. The class should:

1. Have a method `add(promiseFunction)` for adding new promises to the queue. Each `promiseFunction` is a function that, when called, returns a Promise.
2. Ensure that added Promises are executed one after the other, even if they are added at different times. A new Promise should only start executing once the previous Promise has resolved.
3. Optionally, implement a way to handle and log errors from any Promise without stopping the queue.

Criteria for Success:

- Ensure that Promises are executed in the order they are added.
- Successfully log or handle errors from the Promises.
- Demonstrate the functionality with a series of Promises that resolve after varying delays.