JavaScript Asynchronous Programming

readbytes

JavaScript Asynchronous Programming

From Fundamentals to Mastery

readbytes.github.io

2025-07-16

This page is intentionally left blank.

Contents

1 Introduction to Asynchronous Programming **12**
- 1.1 What Is Asynchronous Programming? . . . 12
- 1.2 Synchronous vs Asynchronous Execution . . . 12
- 1.3 Why Asynchronous Programming Matters in JavaScript . . . 14
- 1.4 Callback Functions: The Basics . . . 15
- 1.5 Practical Example: Simple Callback Usage . . . 16
  - 1.5.1 Explanation: . . . 17

2 Understanding the Event Loop **19**
- 2.1 JavaScript's Single Threaded Model . . . 19
- 2.2 The Call Stack, Callback Queue, and Event Loop Explained . . . 19
  - 2.2.1 Summary . . . 21
- 2.3 Macro Tasks vs Micro Tasks . . . 22
  - 2.3.1 Macro Tasks . . . 22
  - 2.3.2 Micro Tasks . . . 22
  - 2.3.3 Execution Order: Micro Macro . . . 22
  - 2.3.4 Example: . . . 23
  - 2.3.5 Why It Matters . . . 23
- 2.4 Practical Example: Visualizing Event Loop Behavior . . . 24
  - 2.4.1 Expected Output: . . . 24
  - 2.4.2 What's Happening? . . . 24
  - 2.4.3 Try It Yourself! . . . 25
  - 2.4.4 Why This Matters . . . 25

3 Callbacks and Callback Hell **27**
- 3.1 Writing and Using Callbacks Effectively . . . 27
  - 3.1.1 Use Descriptive Function Names . . . 27
  - 3.1.2 Keep Callbacks Pure and Focused . . . 27
  - 3.1.3 Separate Callback Logic into Reusable Functions . . . 28
  - 3.1.4 Avoid Deep Nesting Early . . . 28
  - 3.1.5 Summary . . . 29
- 3.2 Problems with Nested Callbacks (Callback Hell) . . . 29
  - 3.2.1 Why Does Callback Hell Happen? . . . 29
  - 3.2.2 A Contrived Example . . . 30
  - 3.2.3 The Bottom Line . . . 30
- 3.3 Error-First Callbacks and Handling Errors . . . 30
  - 3.3.1 What Is an Error-First Callback? . . . 31
  - 3.3.2 Why This Pattern Matters . . . 31
  - 3.3.3 Propagating Errors . . . 31
  - 3.3.4 Summary . . . 32
- 3.4 Practical Example: Callback Pyramid of Doom and Refactoring . . . 32
  - 3.4.1 Original: Callback Pyramid of Doom . . . 33

3.4.2 Refactored: Named Callback Functions . . . 33
3.4.3 Optional Teaser: Promise-Based Refactor (For Future Chapters) . . . 34
3.4.4 Summary . . . 35

4 Promises Fundamentals 37
4.1 Introduction to Promises . . . 37
4.1.1 What Is a Promise? . . . 37
4.1.2 Promise Lifecycle: The Three States . . . 37
4.1.3 Basic Example: . . . 37
4.1.4 Why Use Promises? . . . 38
4.2 Creating and Consuming Promises . . . 38
4.2.1 Creating a Promise . . . 38
4.2.2 Consuming a Promise . . . 39
4.2.3 What Happens Here? . . . 39
4.2.4 Creating Promises with Functions . . . 40
4.2.5 Summary . . . 40
4.3 Promise States and Chaining . . . 40
4.3.1 Promise States Recap . . . 40
4.3.2 How Chaining Works . . . 41
4.3.3 Example: Sequential Async Tasks Using Chaining . . . 41
4.3.4 Whats Happening? . . . 42
4.3.5 Visual Flowchart . . . 42
4.3.6 Why Chaining Solves Callback Hell . . . 42
4.4 Error Handling with Promises . . . 43
4.4.1 How Errors Propagate in Promise Chains . . . 43
4.4.2 Correct Error Handling Example . . . 43
4.4.3 What Happens Without Proper `.catch()` . . . 44
4.4.4 Incorrect Error Handling Example . . . 44
4.4.5 Multiple `.catch()` Handlers . . . 44
4.4.6 Summary . . . 45
4.5 Practical Example: Wrapping Async Operations in Promises . . . 45
4.5.1 Example: Wrapping `setTimeout` in a Promise . . . 45
4.5.2 Whats Happening Here? . . . 46
4.5.3 Why Wrap Async Callbacks? . . . 46
4.5.4 How to Run This Example . . . 46
4.5.5 Summary . . . 46

5 Advanced Promise Patterns 48
5.1 Promise.all, Promise.race, Promise.any, and Promise.allSettled . . . 48
5.1.1 `Promise.all` . . . 48
5.1.2 `Promise.race` . . . 48
5.1.3 `Promise.any` . . . 49
5.1.4 `Promise.allSettled` . . . 49
5.1.5 Summary Table . . . 50
5.2 Sequential vs Parallel Execution with Promises . . . 50

5.2.1 Sequential Execution: One After Another . . . 51
5.2.2 Parallel Execution: Running at the Same Time . . . 51
5.2.3 Trade-offs Between Sequential and Parallel . . . 52
5.2.4 When to Use Which? . . . 52
5.2.5 Summary . . . 52
5.3 Practical Example: Fetching Multiple APIs Concurrently . . . 53
5.3.1 Example: Fetching Multiple Mock APIs . . . 53
5.3.2 How This Works . . . 53
5.3.3 Testing and Experimenting . . . 54
5.3.4 Why Use `Promise.all` Here? . . . 54
5.3.5 Summary . . . 54

6 Async/Await Syntax **56**
6.1 Introduction to Async/Await . . . 56
6.1.1 What Are Async Functions? . . . 56
6.1.2 What Does `await` Do? . . . 56
6.1.3 Simple Async/Await Example . . . 57
6.1.4 Why Use Async/Await? . . . 57
6.2 Converting Promises to Async/Await . . . 57
6.2.1 Promise-Based Code Example . . . 58
6.2.2 Refactoring Using Async/Await . . . 58
6.2.3 Key Changes Explained . . . 59
6.2.4 Why Refactor? . . . 59
6.2.5 Summary . . . 59
6.3 Error Handling with Try/Catch in Async Functions . . . 59
6.3.1 Why Use Try/Catch in Async Functions? . . . 60
6.3.2 Example: Handling Errors in Promise Chains . . . 60
6.3.3 Equivalent Code Using Async/Await and Try/Catch . . . 60
6.3.4 Best Practices for Try/Catch with Async/Await . . . 61
6.3.5 Example: Nested Try/Catch . . . 61
6.3.6 Summary . . . 61
6.4 Practical Example: Refactoring Promises to Async/Await . . . 62
6.4.1 Original Promise Chain Example . . . 62
6.4.2 Refactored Version Using Async/Await . . . 62
6.4.3 Side-by-Side Comparison and Benefits . . . 63
6.4.4 Key Takeaways . . . 63
6.4.5 Running the Example . . . 64
6.4.6 Summary . . . 64

7 Generators and Async Iteration **66**
7.1 Understanding Generators (`function*`) . . . 66
7.1.1 What Is a Generator? . . . 66
7.1.2 The Generator Object and `.next()` Method . . . 66
7.1.3 How Generators Pause and Resume Execution . . . 67
7.1.4 The Iterator Protocol . . . 67

7.1.5 Summary . . . 68
7.2 Yielding Values and Pausing Execution . . . 68
7.2.1 How `yield` Works . . . 68
7.2.2 Basic Example: Yielding Values . . . 68
7.2.3 Sending Values Back In . . . 69
7.2.4 Control Flow Advantages . . . 69
7.2.5 Practical Use: Representing Sequences Lazily . . . 70
7.2.6 Summary . . . 70
7.3 Using Generators for Async Control Flow . . . 70
7.3.1 The Idea Behind Generators for Async Control . . . 71
7.3.2 Manual Runner Example . . . 71
7.3.3 Using the Runner with a Generator . . . 72
7.3.4 How This Works . . . 72
7.3.5 Why This Was Important . . . 73
7.3.6 Summary . . . 73
7.4 Async Iterators and `for await...of` Loops . . . 73
7.4.1 What Are Async Iterators? . . . 73
7.4.2 The `for await...of` Loop . . . 74
7.4.3 How It Works Behind the Scenes . . . 74
7.4.4 Example: Async Generator Producing Data Chunks . . . 74
7.4.5 Key Differences from Regular Iteration . . . 75
7.4.6 Summary . . . 75
7.5 Practical Example: Fetching Paginated Data with Async Iterators . . . 75
7.5.1 Async Iterator for Paginated API . . . 75
7.5.2 Consuming Paginated Data with `for await...of` . . . 76
7.5.3 How This Works . . . 76
7.5.4 Benefits of This Approach . . . 77
7.5.5 Try It Yourself . . . 77

8 Event Emitters and Callbacks 79
8.1 Introduction to EventEmitter in Node.js . . . 79
8.1.1 What Is EventEmitter? . . . 79
8.1.2 The Event-Driven Programming Model . . . 79
8.1.3 How EventEmitter Differs from Traditional Callbacks . . . 80
8.1.4 Why EventEmitter Is Important for Scalable Apps . . . 80
8.1.5 Summary . . . 80
8.2 Emitting and Listening to Events . . . 80
8.2.1 Creating an EventEmitter Instance . . . 81
8.2.2 Registering Event Listeners with `.on()` . . . 81
8.2.3 Emitting Events with `.emit()` . . . 81
8.2.4 Multiple Listeners for the Same Event . . . 81
8.2.5 One-Time Listeners with `.once()` . . . 82
8.2.6 Removing Listeners . . . 82
8.2.7 Summary . . . 82
8.3 Practical Example: Custom Event-Driven Module . . . 83

8.3.1 Creating the Custom Module . . . 83
8.3.2 Using the Module and Subscribing to Events . . . 83
8.3.3 What Happens Here? . . . 84
8.3.4 Benefits of This Event-Driven Design . . . 84
8.3.5 Try It Yourself . . . 84

9 Timers and Scheduling 87
9.1 Using `setTimeout()`, `setInterval()`, and `clearTimeout()` . . . 87
9.1.1 `setTimeout()`: Delayed Execution . . . 87
9.1.2 `setInterval()`: Repeated Execution . . . 87
9.1.3 Canceling Timers with `clearTimeout()` and `clearInterval()` . . . 87
9.1.4 Common Pitfalls: Timer Drift and the Event Loop . . . 88
9.1.5 Practical Tips . . . 88
9.1.6 Summary . . . 88
9.2 Throttling and Debouncing Techniques . . . 89
9.2.1 What Is Throttling? . . . 89
9.2.2 Throttling Example . . . 89
9.2.3 What Is Debouncing? . . . 89
9.2.4 Debouncing Example . . . 90
9.2.5 Key Differences . . . 90
9.2.6 Why Use These Techniques? . . . 90
9.2.7 Summary . . . 91
9.3 Practical Example: Debounced Input Validation . . . 91
9.3.1 Example: Debounced Input Validation . . . 91
9.3.2 How It Works . . . 93
9.3.3 Suggestions for Further Enhancements . . . 93
9.3.4 Summary . . . 94

10 Web APIs and Asynchronous Browser Features 96
10.1 Fetch API Basics and Streaming Responses . . . 96
10.1.1 Why Use Fetch? . . . 96
10.1.2 Basic Syntax . . . 96
10.1.3 GET and POST Requests . . . 96
10.1.4 Handling Streaming Responses . . . 97
10.1.5 Example: Fetching and Processing Streamed Data . . . 97
10.1.6 Summary . . . 98
10.2 XMLHttpRequest vs Fetch . . . 98
10.2.1 XMLHttpRequest: The Old Standard . . . 98
10.2.2 Fetch: The Modern Alternative . . . 99
10.2.3 Browser Compatibility and Use Cases . . . 99
10.2.4 Summary . . . 100
10.3 Using Service Workers and Background Sync . . . 100
10.3.1 What Is a Service Worker? . . . 100
10.3.2 Lifecycle of a Service Worker . . . 101
10.3.3 Caching Strategies . . . 101

10.3.4 Background Sync . . . 101
10.3.5 Browser Support . . . 102
10.3.6 Conceptual Overview Diagram . . . 102
10.3.7 Summary . . . 102
10.4 Practical Example: Building a Progressive Web App Offline Cache . . . 102
10.4.1 Step 1: Register the Service Worker . . . 103
10.4.2 Step 2: Create the Service Worker (`sw.js`) . . . 103
10.4.3 Step 3: Test Offline Functionality . . . 104
10.4.4 Step 4: Cache API Responses (Optional) . . . 104
10.4.5 Debugging Tips . . . 104
10.4.6 Summary . . . 105

11 Handling Streams 107
11.1 Introduction to Streams: Readable, Writable, Duplex . . . 107
11.1.1 Why Streams Matter . . . 107
11.1.2 Types of Streams . . . 107
11.1.3 Conceptual Diagram . . . 108
11.1.4 Summary . . . 108
11.2 Stream Events and Backpressure . . . 109
11.2.1 Common Stream Events . . . 109
11.2.2 What Is Backpressure? . . . 109
11.2.3 Managing Backpressure: Pause and Resume . . . 110
11.2.4 Summary . . . 110
11.3 Using Streams with Promises and Async Iterators . . . 110
11.3.1 Streams and Async Iteration . . . 111
11.3.2 Converting Legacy Streams to Async Iterators . . . 111
11.3.3 Example: Streaming and Writing with Promises . . . 112
11.3.4 Summary . . . 112
11.4 Practical Example: Reading and Writing Files with Streams . . . 112
11.4.1 Goal . . . 113
11.4.2 Complete Example . . . 113
11.4.3 How It Works . . . 113
11.4.4 Error Handling . . . 114
11.4.5 Performance Benefits . . . 114
11.4.6 Conclusion . . . 114

12 Error Handling and Debugging Asynchronous Code 116
12.1 Common Pitfalls and Anti-Patterns . . . 116
12.1.1 Summary . . . 117
12.2 Debugging Async Code with Breakpoints and Logs . . . 117
12.2.1 Use Console Logs Effectively . . . 118
12.2.2 Set Breakpoints in Async Code . . . 118
12.2.3 Trace Async Call Stacks . . . 119
12.2.4 Advanced Tools . . . 119
12.2.5 Summary . . . 119

12.3 Handling Promise Rejections Gracefully . . . 119
12.3.1 Local Error Handling with `.catch()` . . . 120
12.3.2 Error Handling in `async/await` with `try/catch` . . . 120
12.3.3 Global Error Handling in Node.js . . . 120
12.3.4 Recovery and User Experience . . . 121
12.3.5 Summary . . . 121
12.4 Practical Example: Robust Error Handling in Async Workflows . . . 121
12.4.1 Summary . . . 123

13 Parallelism, Concurrency, and Workers 125
13.1 Differences Between Parallelism and Concurrency . . . 125
13.2 Web Workers in Browser JavaScript . . . 126
13.3 Worker Threads in Node.js . . . 129
13.3.1 Summary . . . 131
13.4 Practical Example: Offloading CPU-Intensive Tasks . . . 131
13.4.1 Summary . . . 133

14 Advanced Topics in Asynchronous Control Flow 135
14.1 Async Queues and Rate Limiting . . . 135
14.1.1 Summary . . . 137
14.2 Cancellation with AbortController . . . 137
14.2.1 Summary . . . 138
14.3 Retrying and Timeout Strategies . . . 139
14.3.1 Summary . . . 140
14.4 Practical Example: Robust API Request with Retries and Timeout . . . 140
14.4.1 Explanation . . . 142
14.4.2 Summary . . . 142

Chapter 1.

Introduction to Asynchronous Programming

1. What Is Asynchronous Programming?
2. Synchronous vs Asynchronous Execution
3. Why Asynchronous Programming Matters in JavaScript
4. Callback Functions: The Basics
5. Practical Example: Simple Callback Usage

1 Introduction to Asynchronous Programming

1.1 What Is Asynchronous Programming?

Imagine you're cooking dinner while also answering your phone and checking the oven. You don't stop everything just to wait for the oven timer to ring; instead, you keep doing other things and only react when the timer goes off. This real-world example is a great way to understand **asynchronous programming**.

In programming, **asynchronous programming** means writing code that can start a task and then move on to other things without waiting for that task to finish. It allows your program to handle multiple operations "at the same time" or, more accurately, in a way that doesn't block the main flow of the program.

In traditional, **synchronous programming**, the code runs step-by-step: each operation must complete before the next one begins. This is like waiting by the oven until the timer rings — nothing else can happen while you wait.

JavaScript, especially in web environments, often deals with tasks that can take time to complete — like fetching data from a server, reading files, waiting for user input, or setting timers. If JavaScript waited for these tasks to finish before moving on, the whole page would freeze and become unresponsive.

Instead, asynchronous programming lets JavaScript start these time-consuming tasks and continue running other code right away. When the task is done, JavaScript "comes back" to handle the result. This way, programs stay fast and responsive.

Think of it like sending a letter and then going about your day instead of standing at the mailbox waiting for the reply. When the reply arrives, you get notified and can react accordingly.

1.2 Synchronous vs Asynchronous Execution

To understand the power of asynchronous programming, it's important to first see how it differs from **synchronous** execution — the more traditional way that code runs step-by-step, waiting for each task to finish before moving on.

Synchronous Execution (Blocking)

In synchronous programming, every line of code runs one after another, in sequence. Each operation *blocks* the next until it completes. This means the program can't do anything else while waiting for a time-consuming task.

Here's a simple example in JavaScript:

Full runnable code:

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

function waitThreeSeconds() {
  const start = Date.now();
  while (Date.now() - start < 3000) {
    // busy wait for 3 seconds
  }
}

waitThreeSeconds();

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

Output:

```
Start
(3 second pause)
End
```

In this example, the program prints "Start," then blocks for 3 seconds before printing "End." During that wait, nothing else can happen — the program is stuck.

Asynchronous Execution (Non-blocking)

Now, let's look at an asynchronous version that doesn't block the program while waiting:

Full runnable code:

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

setTimeout(() => {
  console.log("Waited 3 seconds");
}, 3000);

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

Output:

```
Start
End
Waited 3 seconds
```

Here, the program prints "Start," then immediately prints "End" without waiting. After 3 seconds, the message "Waited 3 seconds" appears. The `setTimeout` function schedules a task to run later but lets the rest of the code continue running right away.

Why Does This Matter?

In web applications, blocking the main thread (like in the synchronous example) can freeze the interface, making the app feel slow or unresponsive. Imagine a website that locks up every time it loads data or waits for a timer — users would get frustrated.

Asynchronous code allows the browser to keep responding to user actions — clicks, typing,

scrolling — even while waiting for other tasks to complete. This improves the overall user experience by keeping the app smooth and interactive.

Summary:

- **Synchronous code** runs in order, blocking the program until each task finishes.
- **Asynchronous code** runs tasks "in the background," letting the program continue without waiting.
- Using asynchronous programming in JavaScript helps build faster, more responsive web apps by preventing the interface from freezing during long operations.

1.3 Why Asynchronous Programming Matters in JavaScript

JavaScript is a **single-threaded** language, which means it can only execute one piece of code at a time. Unlike some other programming languages that can run multiple threads simultaneously, JavaScript runs on a single main thread that handles all the code, user interactions, rendering, and events.

This single-threaded nature creates a challenge: if one task takes a long time to complete — like fetching data from a server or processing a large file — the entire program waits, causing the webpage or app to freeze or become unresponsive. This is where **asynchronous programming** becomes crucial.

By using asynchronous programming, JavaScript can start a time-consuming task and then immediately move on to other work — like responding to user clicks, updating animations, or handling other events. When the task finishes, JavaScript returns to process its result. This non-blocking approach keeps the app smooth and responsive.

Here are some common scenarios where asynchronous programming is essential in JavaScript:

- **API Calls:** When your app requests data from a remote server (e.g., fetching user profiles or product information), it can take time for the server to respond. Asynchronous code lets JavaScript continue running while waiting for the data, so the user interface doesn't freeze.
- **Animations and User Interaction:** Smooth animations and immediate responses to user inputs rely on the main thread being free to run continuously. If the main thread is blocked, animations stutter and inputs lag, ruining the user experience.
- **File Operations:** Reading or writing files (especially large ones) can take noticeable time. Async operations let JavaScript initiate these tasks and handle the results later without pausing the rest of the program.

1.4 Callback Functions: The Basics

In JavaScript, **callback functions** are one of the most fundamental ways to handle asynchronous tasks. Simply put, a callback is a function that you **pass as an argument** to another function, with the expectation that it will be **called (or "called back") later**, usually when an asynchronous operation finishes.

This pattern allows JavaScript to start an operation and continue running other code, then come back and execute the callback function when the operation is complete. Callbacks enable non-blocking, asynchronous behavior that keeps your programs responsive.

How Callbacks Work

Let's break down the idea of a callback with a simple example.

Full runnable code:

```
function greet(name) {
  console.log("Hello, " + name + "!");
}

function processUserInput(callback) {
  const userName = "Alice";
  callback(userName);
}

processUserInput(greet);
```

What happens here?

- We declare a function called `greet` that takes a `name` and logs a greeting.
- Another function, `processUserInput`, accepts a `callback` function as a parameter.
- Inside `processUserInput`, we simulate getting a username (in this case, `"Alice"`) and then call the callback, passing that username as an argument.
- Finally, we call `processUserInput(greet)` — passing the `greet` function as the callback.
- When `processUserInput` runs, it invokes the `greet` function with `"Alice"` as the parameter, printing:

  ```
  Hello, Alice!
  ```

Key Points:

- Functions in JavaScript are **first-class citizens** — meaning they can be treated like any other data. You can store them in variables, pass them as arguments, and return them from other functions.
- Passing a function as an argument doesn't **execute** it immediately; it only sends the reference. The function is called later inside the receiving function when appropriate.

- Callbacks often receive data as arguments, enabling them to work with the results of the async operation.

Why Are Callbacks Important?

Callbacks form the foundation of many asynchronous operations in JavaScript, such as responding to user events, timers, or network requests. Before newer patterns like Promises and `async/await` came along, callbacks were the primary way to manage asynchronous code.

Even today, understanding callbacks is essential, as they provide insight into how JavaScript handles tasks behind the scenes.

1.5 Practical Example: Simple Callback Usage

To better understand how callbacks work in asynchronous JavaScript, let's look at a simple, self-contained example that simulates fetching data from a server using `setTimeout()`. This built-in function lets us delay code execution, making it perfect for mimicking a time-consuming task like a network request.

Here's the full example, with clear comments to guide you:

Full runnable code:

```
// A function that simulates fetching data from a server
function fetchData(callback) {
  console.log("Starting data fetch...");

  // Simulate a 2-second delay (like a real server request)
  setTimeout(() => {
    const data = { id: 1, name: "Alice", age: 25 };
    console.log("Data fetched successfully.");

    // Call the callback function and pass the data
    callback(data);
  }, 2000);
}

// A callback function to handle the fetched data
function handleData(data) {
  console.log("Handling data:");
  console.log(JSON.stringify(data,null,2));
}
// Start the process
fetchData(handleData);
```

Expected Output:

```
Starting data fetch...
Data fetched successfully.
Handling data:
```

```
{ id: 1, name: 'Alice', age: 25 }
```

1.5.1 Explanation:

1. `fetchData(callback)` starts by printing a message and then uses `setTimeout()` to delay the next step by 2 seconds — simulating a slow operation like an API call.
2. After 2 seconds, some mock data is created and logged.
3. The `callback(data)` line calls the function passed in (in this case, `handleData`) and sends the mock data to it.
4. `handleData(data)` then prints the received data to the console.

This example demonstrates the power of callbacks: you can define what should happen *after* a task finishes, without blocking the rest of your program. It's the foundation of asynchronous control in JavaScript and an important stepping stone toward more advanced techniques like Promises and `async/await`.

Chapter 2.

Understanding the Event Loop

1. JavaScript's Single Threaded Model
2. The Call Stack, Callback Queue, and Event Loop Explained
3. Macro Tasks vs Micro Tasks
4. Practical Example: Visualizing Event Loop Behavior

2 Understanding the Event Loop

2.1 JavaScript's Single Threaded Model

JavaScript is built on a **single-threaded** execution model, which means it can only do **one thing at a time**. Unlike some languages that support multi-threading — where different parts of a program can run in parallel — JavaScript runs all code on a single thread: the **main thread**.

Think of the main thread like a **one-lane road**. Only one car (task) can drive on it at any given time. If a car stops, all others must wait. This design makes code execution predictable and avoids complex issues like race conditions, but it also creates a challenge: how do we handle tasks that take time — like loading data or waiting for user input — without blocking the road?

That's where JavaScript's **event-driven, asynchronous architecture** comes in.

Why Async Needs Special Handling

Because JavaScript can't run multiple pieces of code simultaneously, it uses a **non-blocking** approach to manage long-running tasks. It doesn't wait for the task to finish; instead, it hands off the work (like a server request or timer) to the browser or Node.js runtime and keeps going. Once the task is done, a **callback** is scheduled to run later, when the main thread is free.

Without this mechanism, a single slow task would freeze the entire application — users wouldn't be able to click buttons, scroll pages, or interact with the app until the task finished.

A Simple Metaphor

Imagine a chef in a kitchen (JavaScript engine) with one burner (the thread). The chef can only cook one dish at a time. If a pot needs to simmer for 10 minutes, the chef doesn't just stand there — instead, they start another dish while the first one cooks. When the simmering is done, a timer (event system) alerts the chef to return and finish the original dish.

This is how JavaScript handles concurrency: it delegates waiting tasks to the environment (like the kitchen timer), and uses the **event loop** to decide when to come back and finish them — all while using just one thread.

2.2 The Call Stack, Callback Queue, and Event Loop Explained

To understand how JavaScript handles asynchronous operations, we need to look at three key components of its runtime architecture:

1. **The Call Stack**
2. **The Callback Queue**

3. **The Event Loop**

Together, these form the core of how JavaScript manages function execution and async behavior — even though it's single-threaded.

The Call Stack

The **call stack** is where JavaScript keeps track of function calls. It works like a stack of plates: when a function is called, it's placed ("pushed") on top of the stack. When it finishes, it's removed ("popped") off.

Here's a simple example:

```
function greet() {
  console.log("Hello!");
}

function start() {
  greet();
}

start();
```

Call Stack Flow:

1. `start()` is pushed to the stack.
2. `start()` calls `greet()`, which is pushed next.
3. `greet()` executes `console.log("Hello!")`.
4. Once `greet()` finishes, it's popped off. Then `start()` is popped off.

All of this happens **synchronously**, one after the other.

The Callback Queue

The **callback queue** (also called the task queue) holds functions that are **waiting to be run** after an async task completes — for example, a `setTimeout`, a click event, or an AJAX request.

But these callbacks don't run immediately when they're ready. Instead, they wait their turn to be pushed onto the call stack.

The Event Loop

The **event loop** is the mechanism that coordinates everything. It constantly checks:

- Is the call stack empty?
- If yes, is there anything in the callback queue?

If both conditions are met, the event loop **moves the first function in the callback queue onto the call stack**, and execution begins.

Step-by-Step Async Example

Let's walk through a simple asynchronous example:

Full runnable code:

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

setTimeout(() => {
  console.log("Callback");
}, 2000);

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

Execution Flow:

1. `"Start"` is logged (pushed and popped from the call stack).
2. `setTimeout()` is called.
   - JavaScript hands the timer task off to the browser environment.
   - The callback (`() => console.log("Callback")`) is **registered** to be queued in 2000ms.
3. `"End"` is logged.
4. After ~2 seconds, the callback is added to the **callback queue**.
5. The event loop sees the call stack is empty and **moves** the callback onto the stack.
6. `"Callback"` is logged.

Output:

```
Start
End
Callback
```

2.2.1 Summary

- The **call stack** handles function execution in a strict, synchronous order.
- The **callback queue** stores async callbacks once they're ready.
- The **event loop** watches the call stack and, when it's empty, pushes callbacks from the queue to be executed.

This elegant system lets JavaScript perform non-blocking operations — like handling user input or making network requests — while still being single-threaded.

2.3 Macro Tasks vs Micro Tasks

In JavaScript's asynchronous execution model, not all tasks are treated equally. Once you understand the **event loop**, it's important to know how **macro tasks** and **micro tasks** fit into the picture — because they determine **when** your async code actually runs.

2.3.1 Macro Tasks

Macro tasks (also called *tasks*) are scheduled operations that are executed in the **callback queue**. Each macro task is processed one at a time, and only after the current call stack is empty.

Common macro tasks include:

- `setTimeout()`
- `setInterval()`
- `setImmediate()` (Node.js)
- I/O operations (like file system reads)
- UI rendering events (in browsers)

When the event loop runs, it picks the first macro task from the queue, executes it completely (including all synchronous and micro task code), and then moves on to the next macro task.

2.3.2 Micro Tasks

Micro tasks (also known as *jobs*) are smaller, higher-priority tasks that run **immediately after the current execution context completes**, but **before** the next macro task starts.

Common micro task sources:

- Promises (`.then()`, `.catch()`, `.finally()`)
- `queueMicrotask()`
- `process.nextTick()` (Node.js)

These are used for fast, fine-grained operations that should run as soon as possible, before handling new events or timers.

2.3.3 Execution Order: Micro Macro

After a function finishes running and the call stack is empty, the **event loop**:

1. Executes **all micro tasks**, in order.

2. Then picks the next **macro task** and starts the process again.

This execution order is critical to understand, because it affects the timing and order of your asynchronous code.

2.3.4 Example:

Full runnable code:

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

setTimeout(() => {
  console.log("Macro Task");
}, 0);

Promise.resolve().then(() => {
  console.log("Micro Task");
});

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

Output:

```
Start
End
Micro Task
Macro Task
```

Here's what happens:

1. `"Start"` and `"End"` run immediately.
2. The Promise resolves and adds its `.then()` to the **micro task queue**.
3. `setTimeout()` schedules a **macro task**.
4. After the current stack clears, the micro task runs first (`"Micro Task"`).
5. Then the macro task runs (`"Macro Task"`).

2.3.5 Why It Matters

Understanding this priority helps prevent subtle bugs and ensures you write efficient, responsive code. For example, if you rely on `setTimeout` to defer a task, but a Promise runs first, you could see unexpected execution order unless you account for the micro/macro difference.

Mastering these queues is essential for working with modern async patterns like Promises, `async/await`, and concurrent UI updates.

2.4 Practical Example: Visualizing Event Loop Behavior

To truly understand how the **event loop**, **call stack**, **callback queue**, and **microtask queue** work together, it helps to observe them in action. Below is a self-contained, runnable example that demonstrates how JavaScript schedules and executes different types of asynchronous tasks — including **macrotasks** like `setTimeout` and **microtasks** like `Promise.then`.

Full runnable code:

```
console.log("Script start");

setTimeout(() => {
  console.log("setTimeout");
}, 0);

Promise.resolve().then(() => {
  console.log("Promise 1");
});

Promise.resolve().then(() => {
  console.log("Promise 2");
});

queueMicrotask(() => {
  console.log("queueMicrotask");
});

console.log("Script end");
```

2.4.1 Expected Output:

```
Script start
Script end
Promise 1
Promise 2
queueMicrotask
setTimeout
```

2.4.2 What's Happening?

Let's break it down step by step:

1. **Synchronous code** is executed first, line by line:
   - `"Script start"` is logged.
   - `setTimeout()` schedules a **macrotask** (timer callback) for later.

- Two `.then()` handlers and a `queueMicrotask()` are scheduled as **microtasks**.
- `"Script end"` is logged.

2. **Microtasks are next**, after the synchronous stack clears:
   - `Promise 1` logs.
   - `Promise 2` logs.
   - `queueMicrotask` logs.
3. **Macrotasks run last**, once all microtasks are done:
   - `setTimeout` logs.

2.4.3 Try It Yourself!

This example is fully runnable in any browser console or Node.js environment. To deepen your understanding:

- Try changing the `setTimeout` delay to a larger number.
- Add a `setImmediate()` (in Node.js) or `process.nextTick()` to see how they behave.
- Nest `Promise.resolve().then()` inside `setTimeout` to observe how task queues interact.

2.4.4 Why This Matters

This example clearly shows how **microtasks have higher priority than macrotasks** and always run before any timers or I/O callbacks. Understanding this ordering helps prevent unexpected behaviors in real-world applications — especially when combining `async/await`, Promises, and event listeners.

By experimenting with small examples like this, you'll develop an intuitive feel for how the event loop keeps JavaScript both responsive and efficient — even with only one thread.

Chapter 3.

Callbacks and Callback Hell

1. Writing and Using Callbacks Effectively
2. Problems with Nested Callbacks (Callback Hell)
3. Error-First Callbacks and Handling Errors
4. Practical Example: Callback Pyramid of Doom and Refactoring

3 Callbacks and Callback Hell

3.1 Writing and Using Callbacks Effectively

Callbacks are a core part of asynchronous programming in JavaScript. When used well, they make your code flexible and modular. But if misused, they can lead to messy, unreadable, and error-prone programs. In this section, we'll explore **best practices** for writing clean and manageable callback functions.

3.1.1 Use Descriptive Function Names

Instead of passing anonymous functions everywhere, define named functions that clearly describe what they do. This improves readability and makes debugging easier.

Avoid:

```
getData(function(data) {
  console.log(data);
});
```

Better:

```
function logData(data) {
  console.log(data);
}

getData(logData);
```

Naming your callbacks makes the flow of logic easier to follow and separates responsibilities clearly.

3.1.2 Keep Callbacks Pure and Focused

A good callback should do **one thing well**. Avoid callbacks that:

- Mutate global state unnecessarily
- Perform multiple unrelated operations
- Contain side effects that make testing difficult

Try to make your callbacks **pure functions**: functions that produce the same output for the same input and don't alter external state.

3.1.3 Separate Callback Logic into Reusable Functions

If you find yourself writing the same anonymous callback multiple times, consider pulling it out into a named, reusable function.

Example:

```
function handleError(err) {
  if (err) {
    console.error("Something went wrong:", err);
    return;
  }
  console.log("Success!");
}

someAsyncTask(handleError);
anotherAsyncTask(handleError);
```

This promotes **modularity** and makes your code easier to test, reuse, and refactor.

3.1.4 Avoid Deep Nesting Early

Nesting callbacks inside callbacks quickly leads to **callback hell**. Even in simple cases, try to flatten your code by extracting logic into separate functions or chaining operations when appropriate.

Avoid:

```
readFile("file.txt", function(err, data) {
  if (!err) {
    parseData(data, function(err, parsed) {
      if (!err) {
        saveData(parsed, function(err) {
          if (!err) {
            console.log("Done!");
          }
        });
      }
    });
  }
});
```

Better:

```
function onFileRead(err, data) {
  if (err) return handleError(err);
  parseData(data, onDataParsed);
}

function onDataParsed(err, parsed) {
  if (err) return handleError(err);
  saveData(parsed, onSaveComplete);
}
```

```
function onSaveComplete(err) {
  if (err) return handleError(err);
  console.log("Done!");
}

readFile("file.txt", onFileRead);
```

3.1.5 Summary

To write effective callbacks:

- Use **clear, descriptive function names**
- **Keep functions focused** and avoid side effects
- Extract **reusable logic** into named functions
- Flatten nesting when possible to improve **readability**

Following these practices ensures your callback-based code remains clean, manageable, and maintainable — especially as your application grows.

3.2 Problems with Nested Callbacks (Callback Hell)

Asynchronous programming with callbacks starts off simple — but as your logic becomes more complex, you may encounter a problem known as **callback hell**.

Callback hell refers to a situation where multiple nested callbacks result in code that is deeply indented, difficult to read, hard to maintain, and prone to bugs. This pattern is sometimes called the **"pyramid of doom"** because of how the code visually forms a triangular shape as it nests deeper and deeper.

3.2.1 Why Does Callback Hell Happen?

In JavaScript, many async operations — like reading files, making API calls, or processing data — are performed using callbacks. When tasks must be performed in sequence, developers often nest one callback inside another to ensure correct order of execution.

While this works, it quickly leads to:

- **Poor readability** due to deep indentation
- **Hard-to-follow logic flow**
- **Difficulty in handling errors** (error handling has to be repeated at each level)
- **Code duplication and tight coupling**

3.2.2 A Contrived Example

```
loginUser("alice", "password123", function(err, user) {
  if (err) return console.error("Login failed:", err);

  getUserSettings(user.id, function(err, settings) {
    if (err) return console.error("Settings error:", err);

    fetchDashboardData(settings, function(err, data) {
      if (err) return console.error("Dashboard error:", err);

      renderDashboard(data, function(err) {
        if (err) return console.error("Render failed:", err);

        console.log("Dashboard loaded successfully!");
      });
    });
  });
});
```

At first glance, this code isn't doing anything complicated: it logs in a user, fetches settings, loads dashboard data, and renders the result. But the **nested structure** makes it difficult to read and scale.

If you need to add another step (e.g., send analytics data), you have to nest even deeper. If any of these functions fail, handling errors consistently also becomes tricky.

3.2.3 The Bottom Line

Callback hell is not about using callbacks — it's about overusing **nested** callbacks without structure. Fortunately, there are solutions: modularizing code, using Promises, and adopting `async/await` can help flatten this pyramid and make your asynchronous code much more manageable. We'll explore those strategies in later chapters.

3.3 Error-First Callbacks and Handling Errors

In asynchronous JavaScript — especially in **Node.js** — a widely adopted pattern for handling errors in callbacks is the **error-first callback** convention. This pattern helps developers manage errors in a consistent and predictable way, particularly in deeply asynchronous code.

3.3.1 What Is an Error-First Callback?

An **error-first callback** is a function where the **first argument is always reserved for an error**, and the **second (and subsequent) arguments** are used for successful results. If there is **no error**, the first argument is `null` or `undefined`.

Here's a typical example:

Full runnable code:

```
function fetchData(callback) {
  // Simulating async behavior
  setTimeout(() => {
    const error = false; // Set to true to simulate an error
    const data = { id: 1, name: "Alice" };

    if (error) {
      callback(new Error("Failed to fetch data"));
    } else {
      callback(null, data);
    }
  }, 1000);
}

fetchData((err, result) => {
  if (err) {
    console.error("Error:", err.message);
    return;
  }
  console.log("Data received:", JSON.stringify(result,null,2));
});
```

3.3.2 Why This Pattern Matters

This pattern is:

- **Standardized**: Most Node.js APIs and third-party libraries use it.
- **Predictable**: Developers always know where to check for errors.
- **Compositional**: It makes it easier to chain or nest async operations while handling errors consistently.

3.3.3 Propagating Errors

When writing your own async functions that use callbacks, always check for errors and **pass them forward** if necessary.

Full runnable code:

```
function stepOne(input, callback) {
  if (!input) {
    return callback(new Error("Invalid input"));
  }
  callback(null, input + 1);
}

function stepTwo(data, callback) {
  if (data > 10) {
    return callback(new Error("Value too large"));
  }
  callback(null, data * 2);
}

stepOne(5, (err, result1) => {
  if (err) return console.error("Step One Failed:", err.message);

  stepTwo(result1, (err, result2) => {
    if (err) return console.error("Step Two Failed:", err.message);

    console.log("Final result:", result2);
  });
});
```

3.3.4 Summary

The **error-first callback** pattern helps manage errors in asynchronous JavaScript by using a consistent structure:

- The first argument is the error (if any).
- Subsequent arguments contain the result.

Following this convention makes your code **clearer**, **easier to debug**, and **compatible** with the broader Node.js ecosystem. It also prepares your code for smoother transition to Promises and `async/await` later on.

3.4 Practical Example: Callback Pyramid of Doom and Refactoring

Let's bring the concept of **callback hell** to life with a practical example. We'll simulate a series of asynchronous tasks — such as fetching user data, retrieving settings, and loading dashboard content — using deeply nested callbacks. Then we'll refactor it to make it cleaner and more maintainable.

3.4.1 Original: Callback Pyramid of Doom

Full runnable code:

```
function fetchUser(id, callback) {
  setTimeout(() => {
    console.log("Fetched user");
    callback(null, { id, name: "Alice" });
  }, 500);
}

function fetchSettings(user, callback) {
  setTimeout(() => {
    console.log("Fetched settings");
    callback(null, { theme: "dark", language: "en" });
  }, 500);
}

function loadDashboard(settings, callback) {
  setTimeout(() => {
    console.log("Loaded dashboard with settings:", settings);
    callback(null, "Dashboard loaded");
  }, 500);
}

// Nested callbacks (Pyramid of Doom)
fetchUser(1, (err, user) => {
  if (err) return console.error("User error:", err);

  fetchSettings(user, (err, settings) => {
    if (err) return console.error("Settings error:", err);

    loadDashboard(settings, (err, result) => {
      if (err) return console.error("Dashboard error:", err);

      console.log(JSON.stringify(result,null,2));
    });
  });
});
```

While this works, it's hard to read, test, and extend. This kind of nesting grows quickly with added complexity.

3.4.2 Refactored: Named Callback Functions

We can flatten the structure using **named functions** for each step:

```
function handleUser(err, user) {
  if (err) return console.error("User error:", err);
  fetchSettings(user, handleSettings);
}

function handleSettings(err, settings) {
```

```
  if (err) return console.error("Settings error:", err);
  loadDashboard(settings, handleDashboard);
}

function handleDashboard(err, result) {
  if (err) return console.error("Dashboard error:", err);
  console.log(JSON.stringify(result,null,2));
}

// Start the chain
fetchUser(1, handleUser);
```

Benefits:

- No deep nesting
- Clear, modular function responsibilities
- Easier to test and debug each step individually

3.4.3 Optional Teaser: Promise-Based Refactor (For Future Chapters)

If Promises are allowed later in the book, this is a preview of how modern JavaScript can simplify the same flow:

Full runnable code:

```
function fetchUser(id) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Fetched user");
      resolve({ id, name: "Alice" });
    }, 500);
  });
}

function fetchSettings(user) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Fetched settings");
      resolve({ theme: "dark", language: "en" });
    }, 500);
  });
}

function loadDashboard(settings) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Loaded dashboard with settings:", JSON.stringify(settings,null,2));
      resolve("Dashboard loaded");
    }, 500);
  });
}
```

```
// Chained Promise version
fetchUser(1)
  .then(fetchSettings)
  .then(loadDashboard)
  .then(console.log)
  .catch(console.error);
```

3.4.4 Summary

The callback pyramid (or "doom") is a common hurdle in async JavaScript. By modularizing callbacks into named functions, you can improve readability and maintainability. Later chapters will show how Promises and `async/await` can take this even further.

Chapter 4.

Promises Fundamentals

1. Introduction to Promises
2. Creating and Consuming Promises
3. Promise States and Chaining
4. Error Handling with Promises
5. Practical Example: Wrapping Async Operations in Promises

4 Promises Fundamentals

4.1 Introduction to Promises

As JavaScript applications grew more complex, developers needed better tools to manage asynchronous code. Callback functions worked, but they often led to **callback hell** — tangled, hard-to-read code with deeply nested structures and scattered error handling.

To solve these problems, JavaScript introduced **Promises** in ES6. A **Promise** provides a cleaner, more structured way to handle asynchronous operations. It represents a **value that may not be available yet**, but will be resolved (or rejected) at some point in the future.

4.1.1 What Is a Promise?

A Promise is an object that acts as a placeholder for the eventual result of an asynchronous operation. Think of it like ordering food at a restaurant:

- You place your order (start the async task).
- You receive a receipt (the Promise).
- Later, the food arrives (the Promise is resolved), or the restaurant says they're out of what you ordered (the Promise is rejected).

This abstraction allows you to write code that waits for the result of an async task in a more readable and maintainable way.

4.1.2 Promise Lifecycle: The Three States

A Promise has **three possible states**:

1. **Pending** – The initial state; the operation is still ongoing.
2. **Fulfilled** – The operation completed successfully, and a result is available.
3. **Rejected** – The operation failed, and an error is available.

Once a Promise is either fulfilled or rejected, it becomes **settled** and cannot change state again.

4.1.3 Basic Example:

Full runnable code:

```
const promise = new Promise((resolve, reject) => {
  const success = true;

  setTimeout(() => {
    if (success) {
      resolve("Data fetched successfully");
    } else {
      reject("Error fetching data");
    }
  }, 1000);
});

promise.then(result => {
  console.log(result); // Runs if resolved
}).catch(error => {
  console.error(error); // Runs if rejected
});
```

4.1.4 Why Use Promises?

- **Improved readability**: Promises avoid deeply nested callbacks.
- **Chainable**: You can sequence async tasks using `.then()`.
- **Centralized error handling**: You can catch errors in one place using `.catch()`.

In short, Promises give JavaScript developers a powerful, elegant way to write non-blocking, asynchronous code that's both **easier to read** and **easier to maintain**.

4.2 Creating and Consuming Promises

A **Promise** in JavaScript is created using the `new Promise()` constructor, which takes a single argument called the **executor function**. This function contains the asynchronous operation and receives two functions as parameters: `resolve` and `reject`.

4.2.1 Creating a Promise

The **executor function** defines the async work. When the operation succeeds, you call `resolve(value)` to fulfill the Promise with a value. If it fails, you call `reject(error)` to reject the Promise with an error.

Here's a simple example simulating a network request:

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
```

```
    const success = true; // Change to false to simulate failure

    if (success) {
      resolve("Data loaded successfully!");
    } else {
      reject("Failed to load data.");
    }
  }, 1000);
});
```

In this code:

- After a 1-second delay, the Promise either resolves with a success message or rejects with an error.
- Notice that `resolve` and `reject` control the Promise's final state.

4.2.2 Consuming a Promise

Once a Promise is created, you **consume** it using these methods:

- **`.then()`**: Runs when the Promise is fulfilled. Receives the resolved value as an argument.
- **`.catch()`**: Runs when the Promise is rejected. Receives the error as an argument.
- **`.finally()`**: Runs regardless of success or failure, useful for cleanup tasks.

Example of consuming the `fetchData` Promise:

```
fetchData
  .then(result => {
    console.log("Success:", result);
  })
  .catch(error => {
    console.error("Error:", error);
  })
  .finally(() => {
    console.log("Operation completed.");
  });
```

4.2.3 What Happens Here?

- If `fetchData` resolves, the `.then()` callback runs, logging the success message.
- If it rejects, the `.catch()` callback runs, logging the error.
- The `.finally()` callback runs regardless of outcome, signaling the operation has finished.

4.2.4 Creating Promises with Functions

You can also wrap async operations inside functions that **return** a Promise:

Full runnable code:

```
function getUserData(userId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (userId === 1) {
        resolve({ id: 1, name: "Alice" });
      } else {
        reject("User not found");
      }
    }, 1000);
  });
}

getUserData(1)
  .then(user => console.log("User:", JSON.stringify(user,null,2)))
  .catch(err => console.error("Error:", err));
```

This pattern is very common for async APIs, allowing you to chain calls and handle results cleanly.

4.2.5 Summary

- Create Promises with `new Promise((resolve, reject) => { ... })`.
- Use `resolve(value)` to fulfill and `reject(error)` to reject.
- Consume Promises with `.then()`, `.catch()`, and `.finally()` for success, error, and cleanup handling.
- Wrapping async tasks in functions returning Promises is a powerful, reusable pattern that improves readability and flow control in your code.

4.3 Promise States and Chaining

To master Promises, it's important to understand their **states** and how **chaining** allows you to perform sequential asynchronous operations without falling back into callback hell.

4.3.1 Promise States Recap

A Promise can be in one of three states:

1. **Pending** — The Promise is still waiting for the async operation to complete.
2. **Fulfilled** — The async operation completed successfully, and the Promise has a value.
3. **Rejected** — The async operation failed, and the Promise has a reason for the failure (an error).

Once a Promise settles into **fulfilled** or **rejected**, its state is final and will not change.

4.3.2 How Chaining Works

One of the key benefits of Promises is the ability to **chain** multiple asynchronous tasks in a clean, readable way. This is done by returning a new Promise (or any value) inside a `.then()` handler.

Each `.then()` call returns a **new Promise**, allowing the next `.then()` in the chain to wait for the previous async operation to complete.

4.3.3 Example: Sequential Async Tasks Using Chaining

Full runnable code:

```
function fetchUser(userId) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Fetched user");
      resolve({ id: userId, name: "Alice" });
    }, 1000);
  });
}

function fetchOrders(user) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Fetched orders for", user.name);
      resolve(["order1", "order2", "order3"]);
    }, 1000);
  });
}

function fetchOrderDetails(orderIds) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Fetched order details");
      resolve(orderIds.map(id => ({ id, detail: "Details for " + id })));
    }, 1000);
  });
}

fetchUser(1)
```

```
  .then(user => fetchOrders(user))          // Return a new Promise
  .then(orders => fetchOrderDetails(orders))// Return another Promise
  .then(details => {
    console.log("Order details:", JSON.stringify(details,null,2));
  })
  .catch(error => {
    console.error("Error:", error);
  });
```

4.3.4 Whats Happening?

- `fetchUser(1)` returns a Promise that resolves with user data.
- The first `.then()` receives the user object and **returns a new Promise** from `fetchOrders(user)`.
- The next `.then()` waits for the orders Promise to resolve, then returns another Promise from `fetchOrderDetails(orders)`.
- The final `.then()` receives all order details and logs them.
- If any Promise in the chain rejects, the `.catch()` at the end handles the error.

4.3.5 Visual Flowchart

```
fetchUser()
   |
   v (resolves user)
.then(fetchOrders)
   |
   v (resolves orders)
.then(fetchOrderDetails)
   |
   v (resolves order details)
.then(log details)
   |
   v
.catch(handle error)
```

4.3.6 Why Chaining Solves Callback Hell

By returning Promises from `.then()` handlers, you avoid nesting callbacks inside callbacks. Instead, the code flows **linearly**, making it easier to read, write, and maintain. This powerful pattern simplifies complex async workflows without sacrificing control or error handling.

Mastering Promise chaining lays the foundation for even cleaner asynchronous code with

`async/await`, which we'll explore in later chapters.

4.4 Error Handling with Promises

Handling errors effectively is crucial in asynchronous programming. Promises provide a clean, consistent way to catch and propagate errors through chains, improving on the often messy error handling in nested callbacks.

4.4.1 How Errors Propagate in Promise Chains

When a Promise is **rejected** or an error is thrown inside a `.then()` handler, the error propagates **down the chain** until it is caught by a `.catch()` handler. This means you can handle multiple async operations' errors in **one centralized place**, rather than handling errors at every step.

4.4.2 Correct Error Handling Example

```
fetchUser(1)
  .then(user => {
    if (!user) {
      throw new Error("User not found");
    }
    return fetchOrders(user);
  })
  .then(orders => fetchOrderDetails(orders))
  .then(details => {
    console.log("Order details:", details);
  })
  .catch(error => {
    // This handles any error in the entire chain above
    console.error("Error caught:", error.message);
  });
```

Here, whether the error comes from:

- A rejected Promise (e.g., `fetchUser` or `fetchOrders` rejects),
- Or a thrown error inside any `.then()`,

the `.catch()` at the end **catches it all** and prevents the program from crashing or leaving errors unhandled.

4.4.3 What Happens Without Proper `.catch()`

If you don't attach a `.catch()` handler or place it too early, you might end up with **unhandled promise rejections**, which can:

- Cause your app to crash (in Node.js)
- Be silently ignored (in some browsers)
- Make debugging difficult

4.4.4 Incorrect Error Handling Example

```
fetchUser(1)
  .then(user => {
    // Missing catch here means errors thrown inside this block won't be caught properly
    if (!user) {
      throw new Error("User missing");
    }
    return fetchOrders(user);
  })
  .then(orders => fetchOrderDetails(orders));

// No .catch() at the end: errors here will go unhandled!
```

In this case, if `fetchUser` or any `.then()` throws or rejects, the error is **not caught** and will generate an unhandled rejection warning.

4.4.5 Multiple `.catch()` Handlers

You can place `.catch()` after specific steps to handle errors locally:

```
fetchUser(1)
  .catch(err => {
    console.error("Failed to fetch user:", err);
    throw err; // Re-throw to propagate to next catch if needed
  })
  .then(user => fetchOrders(user))
  .catch(err => {
    console.error("Failed to fetch orders:", err);
  });
```

But this can complicate flow. Often, a single `.catch()` at the end of the chain is simpler and recommended for most cases.

4.4.6 Summary

- Errors thrown or rejected in Promises propagate down the chain until caught.
- Place a `.catch()` **at the end of your chain** to handle errors globally and avoid unhandled rejections.
- You can use multiple `.catch()` handlers, but be careful to avoid swallowing errors silently.
- Proper error handling with Promises improves your app's reliability and makes debugging much easier.

4.5 Practical Example: Wrapping Async Operations in Promises

Often, you'll encounter asynchronous functions that rely on **callbacks**, such as `setTimeout` or older API functions. Wrapping these in Promises allows you to use the modern Promise-based syntax, improving readability and control flow.

4.5.1 Example: Wrapping `setTimeout` in a Promise

This example demonstrates how to wrap the classic `setTimeout` function in a Promise, simulating a delay-based async operation.

Full runnable code:

```
// wrapTimeout returns a Promise that resolves after 'ms' milliseconds
function wrapTimeout(ms) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(`Waited for ${ms} milliseconds`);
    }, ms);
  });
}

// Using the wrapped function
wrapTimeout(1500)
  .then(message => {
    console.log(message);
    // You can chain more Promises here if needed
    return wrapTimeout(1000);
  })
  .then(nextMessage => {
    console.log(nextMessage);
  })
  .catch(err => {
    console.error("Error:", err);
  });
```

4.5.2 Whats Happening Here?

- The `wrapTimeout` function creates and returns a new Promise.
- Inside the Promise executor, `setTimeout` delays for the specified time.
- After the delay, `resolve` is called, fulfilling the Promise with a message.
- This Promise can be consumed with `.then()`, allowing chaining of async tasks.
- `.catch()` handles any errors that might occur (though `setTimeout` itself does not error).

4.5.3 Why Wrap Async Callbacks?

- **Cleaner syntax:** Using Promises avoids nested callbacks.
- **Better control flow:** Promises can be chained sequentially or run in parallel.
- **Error handling:** Promises simplify catching errors from async operations.
- **Compatibility:** Wrapping legacy callback APIs lets you integrate them smoothly into modern Promise-based code.

4.5.4 How to Run This Example

- Copy the code into any modern browser console or Node.js environment.
- Run it and observe the logged messages appear after the delays.
- Try changing the delay times or chaining more calls to see how the flow stays readable.

4.5.5 Summary

Wrapping callback-based async functions into Promises is a powerful technique. It enhances code readability, allows sequential async workflows without callback nesting, and standardizes error handling — making your JavaScript asynchronous code easier to write and maintain.

Chapter 5.

Advanced Promise Patterns

1. Promise.all, Promise.race, Promise.any, and Promise.allSettled
2. Sequential vs Parallel Execution with Promises
3. Practical Example: Fetching Multiple APIs Concurrently

5 Advanced Promise Patterns

5.1 Promise.all, Promise.race, Promise.any, and Promise.allSettled

JavaScript provides several powerful static methods on the `Promise` object to work with **multiple Promises** simultaneously. These methods let you coordinate async operations efficiently and handle their outcomes in different ways. Let's explore the four main ones: `Promise.all`, `Promise.race`, `Promise.any`, and `Promise.allSettled`.

5.1.1 `Promise.all`

- **Purpose:** Waits for **all** Promises in an iterable to **fulfill**.
- **Result:** Resolves with an array of all resolved values, in order.
- **Rejects:** Immediately rejects if **any** Promise rejects, with that rejection reason.

Example:

Full runnable code:

```
const p1 = Promise.resolve(1);
const p2 = Promise.resolve(2);
const p3 = Promise.resolve(3);

Promise.all([p1, p2, p3])
  .then(results => {
    console.log("All results:", results); // [1, 2, 3]
  })
  .catch(err => {
    console.error("Error:", err);
  });
```

If any promise rejects, `Promise.all` rejects immediately, and the remaining promises are ignored.

5.1.2 `Promise.race`

- **Purpose:** Resolves or rejects as soon as **any one** Promise settles (fulfills or rejects).
- **Result:** The result or error of the **first settled** Promise.
- **Use case:** When you want the fastest result, regardless of success or failure.

Example:

Full runnable code:

```
const fast = new Promise(resolve => setTimeout(() => resolve("fast"), 100));
const slow = new Promise(resolve => setTimeout(() => resolve("slow"), 500));

Promise.race([fast, slow])
  .then(result => {
    console.log("Race winner:", result); // "fast"
  });
```

If the first settled Promise rejects, `Promise.race` rejects immediately.

5.1.3 `Promise.any`

- **Purpose:** Resolves as soon as **any one** Promise **fulfills**.
- **Result:** The value of the first fulfilled Promise.
- **Rejects:** Only if **all** Promises reject, with an `AggregateError`.

Example:

Full runnable code:

```
const p1 = Promise.reject("Error 1");
const p2 = Promise.reject("Error 2");
const p3 = Promise.resolve("Success!");

Promise.any([p1, p2, p3])
  .then(result => {
    console.log("First fulfilled:", result); // "Success!"
  })
  .catch(err => {
    console.error("All failed:", err);
  });
```

`Promise.any` is useful when you want the **first successful** result and can ignore failures unless all fail.

5.1.4 `Promise.allSettled`

- **Purpose:** Waits for **all** Promises to **settle** (either fulfill or reject).
- **Result:** Resolves with an array of objects describing each Promise's outcome (`status` and `value` or `reason`).
- **Never rejects:** Always resolves once all Promises settle.

Example:

Full runnable code:

```
const p1 = Promise.resolve("Success");
const p2 = Promise.reject("Failure");

Promise.allSettled([p1, p2])
  .then(results => {
    console.log(JSON.stringify(results,null,2));
    /*
    [
      { status: "fulfilled", value: "Success" },
      { status: "rejected", reason: "Failure" }
    ]
    */
  });
```

Use `allSettled` when you want to know the outcome of every Promise, regardless of success or failure.

5.1.5 Summary Table

Method	Waits for…	Resolves when…	Rejects when…	Result
`Promise.all`	All promises fulfill	All fulfill	Any promise rejects	Array of values
`Promise.race`	First promise settles	First settles (fulfill or reject)	N/A (rejects if first rejects)	Value or error of first settled
`Promise.any`	First promise fulfills	First fulfills	All promises reject	Value of first fulfilled
`Promise.allSettled`	All promises settle	All settle (fulfill or reject)	Never rejects	Array of status objects

Understanding these methods empowers you to coordinate multiple async tasks according to your app's needs — whether you want all results, the fastest one, the first success, or a complete report of every outcome.

5.2 Sequential vs Parallel Execution with Promises

When working with multiple asynchronous tasks, understanding **how** and **when** they run is key to writing efficient JavaScript. Promises give you flexibility to execute tasks **sequentially** or **in parallel**, each with trade-offs related to performance and task dependencies.

5.2.1 Sequential Execution: One After Another

Sequential execution means running async tasks **one at a time**, waiting for each to complete before starting the next. This is often necessary when a task depends on the result of a previous one.

You achieve this by **chaining Promises** using `.then()`:

Full runnable code:

```
function task(name, delay) {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log(`${name} completed`);
      resolve(name);
    }, delay);
  });
}

// Run tasks sequentially
task("Task 1", 1000)
  .then(() => task("Task 2", 1000))
  .then(() => task("Task 3", 1000))
  .then(() => {
    console.log("All tasks done sequentially");
  });
```

Output (approximate timing):

- Task 1 completed (after 1s)
- Task 2 completed (after 2s)
- Task 3 completed (after 3s)
- All tasks done sequentially

Each task waits for the previous one to finish, resulting in a **total duration equal to the sum** of all task times.

5.2.2 Parallel Execution: Running at the Same Time

Parallel execution means starting multiple async tasks **at once** and waiting for all to finish. This improves performance when tasks are independent.

Use `Promise.all()` to run tasks in parallel:

```
Promise.all([
  task("Task 1", 1000),
  task("Task 2", 1000),
  task("Task 3", 1000)
]).then(() => {
  console.log("All tasks done in parallel");
});
```

Output (approximate timing):

- Task 1 completed (after 1s)
- Task 2 completed (after 1s)
- Task 3 completed (after 1s)
- All tasks done in parallel

All tasks start together, so the total time is roughly the **longest individual task duration** (about 1 second here).

5.2.3 Trade-offs Between Sequential and Parallel

Aspect	Sequential	Parallel
Performance	Slower (sum of task durations)	Faster (duration of longest task)
Order / Dependency	Necessary if tasks depend on each other	Suitable for independent tasks
Resource usage	Uses fewer resources simultaneously	May consume more resources

5.2.4 When to Use Which?

- Use **sequential** execution if tasks **depend on previous results**, such as fetching user data, then orders for that user.
- Use **parallel** execution when tasks are **independent** and can run simultaneously, like loading multiple images or API calls unrelated to each other.

5.2.5 Summary

Promises let you control the flow of asynchronous tasks:

- **Sequential chaining** enforces order but takes longer.
- **Parallel execution** with `Promise.all` speeds up overall runtime when tasks can run simultaneously.

Understanding these patterns helps you write performant, clear, and effective async JavaScript code tailored to your app's needs.

5.3 Practical Example: Fetching Multiple APIs Concurrently

In real-world applications, you often need to fetch data from multiple APIs at the same time — for example, loading user info, recent posts, and notifications simultaneously. Using `Promise.all`, you can run these requests **concurrently** to improve performance and handle all results together.

5.3.1 Example: Fetching Multiple Mock APIs

This example simulates fetching data from three mock APIs using `fetch` wrapped with `Promise.resolve` and `setTimeout` to imitate network delays.

Full runnable code:

```
// Simulated API fetch function returning a Promise
function fetchMockAPI(name, delay, shouldFail = false) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (shouldFail) {
        reject(`${name} API failed`);
      } else {
        resolve({ api: name, data: `${name} data` });
      }
    }, delay);
  });
}

// Fetch all APIs concurrently using Promise.all
Promise.all([
  fetchMockAPI("User", 1000),
  fetchMockAPI("Posts", 1500),
  fetchMockAPI("Notifications", 1200)
])
  .then(results => {
    console.log("All API data fetched successfully:");
    results.forEach(result => console.log(JSON.stringify(result,null,2)));
  })
  .catch(error => {
    console.error("One or more API calls failed:", error);
  });
```

5.3.2 How This Works

- We define `fetchMockAPI` to simulate an API call that resolves after a delay or rejects to simulate failure.
- `Promise.all` receives an array of these simulated fetches and runs them **in parallel**.
- If **all** Promises resolve, `.then()` receives an array of results in the order of the Promises.

- If **any** Promise rejects, `.catch()` catches the error immediately.

5.3.3 Testing and Experimenting

- Copy and paste the code into any modern browser console or Node.js environment.
- Run it and observe how all three API results print together after the longest delay.
- To test error handling, add `true` as the third argument to any `fetchMockAPI` call to simulate a failure, e.g.:

```
fetchMockAPI("Posts", 1500, true)
```

- Notice that the `.catch()` block runs immediately with the error message from the failed API.

5.3.4 Why Use `Promise.all` Here?

- **Performance:** All API calls start simultaneously, reducing total wait time.
- **Consistency:** You get all results in one place and maintain order.
- **Error Handling:** Any failure triggers the catch block, letting you handle partial failures gracefully.

5.3.5 Summary

`Promise.all` is an ideal tool for running multiple async operations concurrently when all results are needed together. This example shows how easy it is to coordinate multiple API calls, handle success, and catch errors without complicated callback structures. Try modifying delays or failure flags to see how `Promise.all` manages concurrency and errors in practice.

Chapter 6.

Async/Await Syntax

1. Introduction to Async/Await
2. Converting Promises to Async/Await
3. Error Handling with Try/Catch in Async Functions
4. Practical Example: Refactoring Promises to Async/Await

6 Async/Await Syntax

6.1 Introduction to Async/Await

JavaScript's **async/await** syntax, introduced in ES2017, is a powerful way to write asynchronous code that looks and behaves like synchronous code. It is often called **syntactic sugar** over Promises because it builds on top of Promises but makes the code easier to read, write, and maintain.

6.1.1 What Are Async Functions?

An **async function** is a special kind of function that always **returns a Promise**, even if you explicitly return a non-Promise value. Marking a function with the `async` keyword means it can contain the `await` keyword inside it.

Example:

```
async function greet() {
  return "Hello, World!";
}

greet().then(message => console.log(message));  // Logs: Hello, World!
```

Although `greet()` returns a simple string inside, calling it returns a Promise that resolves with that string. This allows you to use `.then()` or `await` when calling the function.

6.1.2 What Does `await` Do?

The `await` keyword pauses the execution of an async function until a Promise settles (either resolves or rejects). It **unwraps** the Promise value, so you get the resolved data directly without needing `.then()` callbacks.

Using `await` makes asynchronous code **look synchronous**:

```
async function fetchData() {
  const data = await fetchSomeData();  // Wait for the Promise to resolve
  console.log(data);                   // Use the resolved data
}
```

Here, the code waits for `fetchSomeData()` to finish, then proceeds — just like regular blocking code, but without freezing the main thread.

6.1.3 Simple Async/Await Example

Full runnable code:

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function sayHello() {
  console.log("Waiting 1 second...");
  await delay(1000);  // Pause here for 1 second
  console.log("Hello after 1 second!");
}

sayHello();
```

- The function `delay` returns a Promise that resolves after a timeout.
- Inside the async function `sayHello`, we `await` the delay, pausing the function execution for 1 second without blocking the whole program.
- After the delay, the message logs to the console.

6.1.4 Why Use Async/Await?

- **Cleaner, linear code:** No more nested `.then()` chains or callback pyramids.
- **Better readability:** Async operations look like regular synchronous code.
- **Easier error handling:** Try/catch blocks work naturally with async/await (covered in the next section).
- **Full Promise compatibility:** Async functions still return Promises and work well with existing Promise-based code.

In summary, async/await simplifies working with asynchronous tasks by making your code more straightforward and easier to follow, while keeping the power and flexibility of Promises underneath. It's a modern tool every JavaScript developer should master.

6.2 Converting Promises to Async/Await

One of the greatest benefits of async/await is how it transforms complex Promise chains into simple, easy-to-read code that looks synchronous. In this section, we'll explore how to refactor existing Promise-based code to use async/await and understand the key changes involved.

6.2.1 Promise-Based Code Example

Consider the following Promise chain that fetches user data, then fetches their orders, and finally logs the result:

Full runnable code:

```
function fetchUser(userId) {
  return new Promise((resolve) => {
    setTimeout(() => resolve({ id: userId, name: "Alice" }), 1000);
  });
}

function fetchOrders(user) {
  return new Promise((resolve) => {
    setTimeout(() => resolve([`Order1 for ${user.name}`, `Order2 for ${user.name}`]), 1000);
  });
}

fetchUser(1)
  .then(user => fetchOrders(user))
  .then(orders => {
    console.log("Orders:", orders);
  })
  .catch(error => {
    console.error("Error:", error);
  });
```

Although this works perfectly, the nested `.then()` calls can quickly become unwieldy with more steps or complex logic.

6.2.2 Refactoring Using Async/Await

To refactor this to async/await, follow these steps:

1. **Mark the function as `async`** where you want to perform the async tasks.
2. **Replace `.then()` calls with `await` expressions** to pause execution until the Promise resolves.
3. Use **try/catch** blocks (covered in the next section) to handle errors gracefully.

Here's the refactored code:

```
async function getUserOrders(userId) {
  try {
    const user = await fetchUser(userId);
    const orders = await fetchOrders(user);
    console.log("Orders:", orders);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

```
getUserOrders(1);
```

6.2.3 Key Changes Explained

- The **`getUserOrders`** function is declared as `async`. This means it implicitly returns a Promise.
- Instead of chaining `.then()`, we `await` the result of each Promise: `fetchUser` and then `fetchOrders`.
- The code reads top-to-bottom, making the logic easier to follow.
- The **error handling** is centralized in a `try/catch` block rather than a `.catch()` at the end of the chain.
- The function is called normally — calling an async function returns a Promise, so you can still chain `.then()` or use `await` when calling it.

6.2.4 Why Refactor?

- **Readability:** Code looks like synchronous code, making it more approachable.
- **Maintainability:** Less nesting means fewer bugs and easier debugging.
- **Control flow:** Use familiar control structures like `if`, `for`, and `try/catch` around async code.
- **Consistency:** Async/await works seamlessly with existing Promise APIs.

6.2.5 Summary

Converting Promise chains to async/await mainly involves replacing `.then()` calls with `await` inside async functions, resulting in cleaner, more straightforward code. This modern approach improves readability without sacrificing any of the powerful capabilities of Promises. In the next section, we'll explore how to handle errors effectively using try/catch in async functions.

6.3 Error Handling with Try/Catch in Async Functions

When working with Promises, error handling is typically done using `.catch()` at the end of a Promise chain. With async/await, you can handle errors more naturally using **`try/catch` blocks** inside your async functions. This approach feels more like synchronous code and

offers clearer, more maintainable error handling.

6.3.1 Why Use Try/Catch in Async Functions?

Inside an `async` function, any Promise that rejects will cause the `await` expression to throw an error — similar to how a regular `throw` statement works in synchronous code. Wrapping the `await` calls in a `try` block lets you **catch these errors immediately** and respond appropriately.

This contrasts with Promise chains, where you typically attach a `.catch()` method at the end, which handles errors from any preceding Promise in the chain.

6.3.2 Example: Handling Errors in Promise Chains

```
fetchUser(1)
  .then(user => fetchOrders(user))
  .then(orders => {
    console.log("Orders:", orders);
  })
  .catch(error => {
    console.error("Error caught:", error);
  });
```

Here, `.catch()` handles any rejection or thrown error from the entire chain.

6.3.3 Equivalent Code Using Async/Await and Try/Catch

```
async function getUserOrders(userId) {
  try {
    const user = await fetchUser(userId);
    const orders = await fetchOrders(user);
    console.log("Orders:", orders);
  } catch (error) {
    console.error("Error caught:", error);
  }
}

getUserOrders(1);
```

In this example:

- The `try` block contains all awaited calls.
- If **any** `await`ed Promise rejects or throws, execution immediately jumps to the `catch`

block.
- This structure allows for clear, linear error handling without needing to chain `.catch()` calls.

6.3.4 Best Practices for Try/Catch with Async/Await

- Use **one try/catch per logical operation** to handle errors meaningfully.
- You can also have **nested try/catch blocks** if different errors require different handling.
- Always handle errors inside async functions, or make sure the caller handles the rejected Promise.

6.3.5 Example: Nested Try/Catch

```
async function process() {
  try {
    const user = await fetchUser(1);

    try {
      const orders = await fetchOrders(user);
      console.log("Orders:", orders);
    } catch (ordersError) {
      console.error("Failed to fetch orders:", ordersError);
    }

  } catch (userError) {
    console.error("Failed to fetch user:", userError);
  }
}

process();
```

6.3.6 Summary

- `try/catch` in async functions lets you catch errors thrown by rejected Promises **at the exact point** of the `await` expression.
- This style improves readability and error scope control compared to `.catch()` on Promise chains.
- Proper error handling ensures your application behaves predictably and errors are easier to debug.

Using `try/catch` with async/await is the modern, clean way to manage asynchronous errors in JavaScript.

6.4 Practical Example: Refactoring Promises to Async/Await

Let's walk through a practical example where we start with a multi-step asynchronous operation implemented using Promise chains, and then refactor it using async/await. This comparison will highlight how async/await improves clarity and error handling.

6.4.1 Original Promise Chain Example

Imagine we want to fetch a user by ID, then fetch their orders, and finally log the orders:

Full runnable code:

```
function fetchUser(userId) {
  return new Promise((resolve) => {
    setTimeout(() => resolve({ id: userId, name: "Alice" }), 1000);
  });
}

function fetchOrders(user) {
  return new Promise((resolve) => {
    setTimeout(() => resolve([`Order1 for ${user.name}`, `Order2 for ${user.name}`]), 1000);
  });
}

fetchUser(1)
  .then(user => {
    return fetchOrders(user);
  })
  .then(orders => {
    console.log("Orders:", orders);
  })
  .catch(error => {
    console.error("Error:", error);
  });
```

What's happening here?

- Each `.then()` waits for the previous Promise to resolve.
- The nested `.then()` calls can become confusing as logic grows.
- The `.catch()` at the end handles errors from any step.

6.4.2 Refactored Version Using Async/Await

Now, let's rewrite the same logic inside an `async` function:

```
function fetchUser(userId) {
  return new Promise((resolve) => {
    setTimeout(() => resolve({ id: userId, name: "Alice" }), 1000);
```

```
  });
}

function fetchOrders(user) {
  return new Promise((resolve) => {
    setTimeout(() => resolve([`Order1 for ${user.name}`, `Order2 for ${user.name}`]), 1000);
  });
}

async function getUserOrders(userId) {
  try {
    const user = await fetchUser(userId);      // Wait for user data
    const orders = await fetchOrders(user);    // Wait for orders
    console.log("Orders:", orders);             // Use the results
  } catch (error) {
    console.error("Error:", error);             // Handle any errors here
  }
}

getUserOrders(1);
```

6.4.3 Side-by-Side Comparison and Benefits

Aspect	Promise Chain	Async/Await
Readability	Nested `.then()` calls can get deep and harder to follow	Reads top-to-bottom like synchronous code
Error Handling	Single `.catch()` at the end handles all errors	`try/catch` lets you handle errors inline and with familiar syntax
Flow Control	Returning Promises inside `.then()` can be confusing	`await` pauses execution until Promise resolves, making logic straightforward
Maintainability	Adding more steps increases indentation and complexity	Easy to add steps sequentially without extra nesting
Debugging	Stack traces may be less clear due to chained callbacks	Errors thrown inside async function behave like synchronous errors

6.4.4 Key Takeaways

- Async/await transforms complex Promise chains into clear, linear code.
- Error handling with `try/catch` inside async functions feels natural and scoped.
- Async/await preserves all Promise functionality while improving developer experience.

6.4.5 Running the Example

- Paste either example in a modern browser console or Node.js.
- Observe the output and error handling.
- Try introducing errors (e.g., reject a Promise) to see how each method responds.

6.4.6 Summary

Refactoring Promise chains into async/await code results in cleaner, easier-to-understand asynchronous logic. This makes your code more maintainable and less error-prone — essential for mastering modern JavaScript asynchronous programming.

Chapter 7.

Generators and Async Iteration

1. Understanding Generators (`function*`)
2. Yielding Values and Pausing Execution
3. Using Generators for Async Control Flow
4. Async Iterators and `for await...of` Loops
5. Practical Example: Fetching Paginated Data with Async Iterators

7 Generators and Async Iteration

7.1 Understanding Generators (`function*`)

Generators are a special type of function in JavaScript that allow you to **pause and resume execution** at will. Unlike regular functions that run from start to finish, a generator function can yield multiple values over time, giving you fine control over the flow of your code.

7.1.1 What Is a Generator?

A generator is defined using the `function*` syntax instead of the regular `function`. The asterisk `*` indicates that this function is a generator:

```
function* simpleGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
```

When you call a generator function, it doesn't execute its code immediately. Instead, it returns a special **generator object** which conforms to the **iterator protocol**.

7.1.2 The Generator Object and `.next()` Method

The generator object has a `.next()` method, which is how you **step through** the generator's execution. Each call to `.next()` resumes the function until it hits the next `yield` statement or finishes.

Full runnable code:

```
function* simpleGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
const gen = simpleGenerator();

console.log(JSON.stringify(gen.next(),null,2)); // { value: 1, done: false }
console.log(JSON.stringify(gen.next(),null,2)); // { value: 2, done: false }
console.log(JSON.stringify(gen.next(),null,2)); // { value: 3, done: false }
console.log(JSON.stringify(gen.next(),null,2)); // { value: undefined, done: true }
```

- Each `.next()` call returns an object with two properties:
  - **`value`**: The yielded value at that step.
  - **`done`**: A boolean indicating if the generator has completed execution (`true` means

finished).

The generator pauses at each `yield` and only continues when `.next()` is called again.

7.1.3 How Generators Pause and Resume Execution

Think of a generator as a **book with bookmarks**. Each `yield` is like a bookmark inside the function. When you call `.next()`, the generator reads from the last bookmark until it reaches the next one, then pauses again.

For example:

Full runnable code:

```
function* countUpToThree() {
  console.log("Start counting");
  yield 1;
  console.log("Reached 1");
  yield 2;
  console.log("Reached 2");
  yield 3;
  console.log("Done counting");
}

const counter = countUpToThree();

counter.next(); // Logs "Start counting", returns {value: 1, done: false}
counter.next(); // Logs "Reached 1", returns {value: 2, done: false}
counter.next(); // Logs "Reached 2", returns {value: 3, done: false}
counter.next(); // Logs "Done counting", returns {value: undefined, done: true}
```

7.1.4 The Iterator Protocol

Generators implement the **iterator protocol**, meaning they provide a `.next()` method that returns `{ value, done }` objects. This lets you use generators with language features like `for...of` loops:

Full runnable code:

```
function* simpleGenerator() {
  yield 1;
  yield 2;
  yield 3;
}
for (const value of simpleGenerator()) {
  console.log(value);
}
// Logs:
```

```
// 1
// 2
// 3
```

7.1.5 Summary

Generators are powerful tools that let you write functions capable of pausing and resuming execution, producing multiple values over time. By using the `function*` syntax, generators return an iterator object with a `.next()` method. This method controls the flow by moving execution from one `yield` statement to the next, providing a simple yet flexible mechanism to handle sequences and control flow synchronously.

Understanding these basics prepares you to use generators for more advanced patterns, including asynchronous control flow and data streaming, which we'll explore later in this chapter.

7.2 Yielding Values and Pausing Execution

The heart of a generator function lies in the `yield` keyword. Unlike regular functions that run straight through, a generator uses `yield` to **produce values one at a time** and **pause execution** between those values. This powerful mechanism gives you fine-grained control over function execution and data flow.

7.2.1 How `yield` Works

When a generator function hits a `yield` statement, it:

1. **Outputs (yields) a value** to the caller.
2. **Pauses its execution**, saving its current state.
3. Waits until the caller resumes execution by calling `.next()` again.

This behavior allows the generator to produce a sequence of values lazily, meaning values are generated only as needed, rather than all at once.

7.2.2 Basic Example: Yielding Values

Full runnable code:

```
function* simpleGenerator() {
  yield 'First';
  yield 'Second';
  yield 'Third';
}

const gen = simpleGenerator();

console.log(gen.next()); // { value: 'First', done: false }
console.log(gen.next()); // { value: 'Second', done: false }
console.log(gen.next()); // { value: 'Third', done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Each call to `gen.next()` resumes the function until the next `yield`, returning the yielded value and pausing again.

7.2.3 Sending Values Back In

An often overlooked feature of generators is that you can **send values back into the generator** when resuming execution by passing an argument to `.next(value)`. This value becomes the result of the current `yield` expression inside the generator.

Example:

Full runnable code:

```
function* echoGenerator() {
  const input1 = yield 'Say something:';
  console.log('Received:', input1);

  const input2 = yield 'Say something else:';
  console.log('Received:', input2);
}

const gen = echoGenerator();

console.log(gen.next());          // Start generator, outputs: { value: 'Say something:', done: false }
console.log(gen.next('Hello'));  // Sends 'Hello' back, logs "Received: Hello"
console.log(gen.next('World'));  // Sends 'World' back, logs "Received: World"
console.log(gen.next());          // Ends generator
```

This two-way communication enables complex control flows where the generator can react dynamically to external input.

7.2.4 Control Flow Advantages

Using `yield` and `.next(value)` gives generators several advantages:

- **Pause and resume:** You can pause execution exactly where you want and pick up later.
- **Lazy evaluation:** Instead of calculating an entire sequence upfront, you generate values on demand.
- **Two-way communication:** Generators can receive data during execution, enabling interactive or stateful flows.

7.2.5 Practical Use: Representing Sequences Lazily

Imagine generating an infinite sequence without calculating all values at once:

Full runnable code:

```
function* infiniteNumbers() {
  let num = 1;
  while (true) {
    yield num++;
  }
}

const gen = infiniteNumbers();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // 3
// And so on...
```

The generator produces numbers only when `.next()` is called, saving memory and processing time.

7.2.6 Summary

- The `yield` keyword pauses a generator, outputting a value.
- Passing values back with `.next(value)` enables two-way interaction.
- Generators can produce sequences lazily, improving performance and control.
- This flexible flow control is the foundation for advanced asynchronous patterns, which we'll explore in upcoming sections.

7.3 Using Generators for Async Control Flow

Before `async/await` became part of JavaScript, generators were used as a clever way to manage asynchronous code and make it look synchronous. By leveraging the ability of

generators to pause and resume execution with `yield`, developers could write asynchronous workflows that were easier to read and maintain than deeply nested callbacks or complex Promise chains.

7.3.1 The Idea Behind Generators for Async Control

Generators pause execution at each `yield`. If you yield a **Promise**, you can wait for that Promise to resolve before continuing the generator's execution. This lets you write code that appears to run in a straight line, even though it's handling asynchronous tasks.

However, JavaScript itself doesn't automatically handle resuming the generator once the Promise resolves. To make this work, you need a **runner function** or a helper library (like `co`) that takes care of:

- Calling `.next()` on the generator,
- Waiting for the yielded Promise to resolve,
- Resuming the generator with the resolved value.

This pattern was a major step toward modern async flow control before native `async/await` was available.

7.3.2 Manual Runner Example

Here's a simple manual runner that drives a generator which yields Promises:

```
function run(generatorFunc) {
  const iterator = generatorFunc();

  function iterate(iteration) {
    if (iteration.done) return Promise.resolve(iteration.value);

    return Promise.resolve(iteration.value)
      .then(res => iterate(iterator.next(res)))
      .catch(err => iterator.throw(err));
  }

  try {
    return iterate(iterator.next());
  } catch (err) {
    return Promise.reject(err);
  }
}
```

7.3.3 Using the Runner with a Generator

Now, let's write an async workflow using a generator:

Full runnable code:

```
function run(generatorFunc) {
  const iterator = generatorFunc();

  function iterate(iteration) {
    if (iteration.done) return Promise.resolve(iteration.value);

    return Promise.resolve(iteration.value)
      .then(res => iterate(iterator.next(res)))
      .catch(err => iterator.throw(err));
  }

  try {
    return iterate(iterator.next());
  } catch (err) {
    return Promise.reject(err);
  }
}

function fetchData(url) {
  return new Promise(resolve => {
    setTimeout(() => resolve(`Data from ${url}`), 1000);
  });
}

function* asyncWorkflow() {
  try {
    const user = yield fetchData('/user');
    console.log('User:', user);

    const orders = yield fetchData('/orders');
    console.log('Orders:', orders);

    return 'Done';
  } catch (error) {
    console.error('Error:', error);
  }
}

run(asyncWorkflow).then(result => {
  console.log(result); // Logs: Done
});
```

7.3.4 How This Works

- The generator yields a Promise each time it wants to wait for async data.
- The `run` function waits for the Promise to resolve, then calls `.next(resolvedValue)` to resume the generator.

- Errors from Promises are caught and passed back to the generator with `.throw()`, allowing error handling inside the generator's `try/catch`.
- The result is asynchronous code written in a **linear, readable style** without nested callbacks.

7.3.5 Why This Was Important

Before `async/await`, this pattern helped solve callback hell and made asynchronous code more manageable. Libraries like `co` automated this runner functionality.

With native `async/await`, this pattern is largely obsolete, but understanding it provides insight into how async flow control evolved in JavaScript and how generators can be harnessed creatively beyond their original use.

7.3.6 Summary

- Generators can pause at `yield` while waiting for Promises, enabling asynchronous control flow that looks synchronous.
- A runner function resumes the generator once Promises settle.
- This pattern improved async code readability before `async/await`.
- Knowing this pattern helps understand the evolution of JavaScript async programming and the power of generators.

7.4 Async Iterators and `for await...of` Loops

In modern JavaScript, **async iterators** provide a powerful way to work with streams of asynchronous data. Just like regular iterators let you consume sequences one item at a time, async iterators let you do the same—but for data that arrives asynchronously, such as reading from a file, receiving network data, or querying paginated APIs.

7.4.1 What Are Async Iterators?

An async iterator is an object that implements the `[Symbol.asyncIterator]` method. Unlike regular iterators that return `{ value, done }` immediately, async iterators return a **Promise** resolving to `{ value, done }`. This means each step can wait for asynchronous operations to complete before producing the next value.

7.4.2 The `for await...of` Loop

To consume async iterators, JavaScript provides the `for await...of` loop. It looks similar to the regular `for...of` loop but is designed to handle Promises automatically.

```
for await (const item of asyncIterable) {
  console.log(item);
}
```

- It **waits** for each Promise to resolve before moving to the next iteration.
- The loop pauses until the async iterator provides the next value.
- It ends when the iterator signals it's done.

This syntax makes reading asynchronous streams straightforward and elegant.

7.4.3 How It Works Behind the Scenes

When the loop begins, it calls the async iterator's `[Symbol.asyncIterator]()` method, which returns an async iterator object with a `.next()` method.

Each `.next()` call returns a Promise that resolves to `{ value, done }`. The loop waits (`await`s) for this Promise, then processes the value if `done` is `false`. When `done` is `true`, the loop terminates.

7.4.4 Example: Async Generator Producing Data Chunks

Full runnable code:

```
async function* fetchDataChunks() {
  const chunks = ['chunk1', 'chunk2', 'chunk3'];
  for (const chunk of chunks) {
    // Simulate asynchronous delay
    await new Promise(resolve => setTimeout(resolve, 500));
    yield chunk;
  }
}

(async () => {
  for await (const chunk of fetchDataChunks()) {
    console.log('Received:', chunk);
  }
  console.log('All chunks received');
})();
```

This example simulates receiving data chunks asynchronously. The `for await...of` loop automatically waits for each chunk before printing it.

7.4.5 Key Differences from Regular Iteration

Feature	Regular Iterator (`for...of`)	Async Iterator (`for await...of`)
Method	`[Symbol.iterator]()`	`[Symbol.asyncIterator]()`
`.next()` return	`{ value, done }`	`Promise` resolving to `{ value, done }`
Use case	Synchronous sequences	Asynchronous data streams
Loop waits for value?	No — values are immediately available	Yes — waits for Promise to resolve

7.4.6 Summary

Async iterators and `for await...of` loops provide a clean, readable way to consume asynchronous sequences. They combine the convenience of iteration with the power of Promises, making asynchronous data handling simpler and more intuitive. This pattern is essential for working with streams, paginated APIs, and other async data sources in modern JavaScript.

7.5 Practical Example: Fetching Paginated Data with Async Iterators

Fetching data from APIs that return results in **pages** is a common scenario. Using async iterators, you can build a clean, readable way to fetch and process each page one at a time as the data arrives — without loading everything upfront.

7.5.1 Async Iterator for Paginated API

Here's a simple example that simulates fetching paginated data from an API using an async generator function. It yields each page's results as they become available:

```
// Simulated paginated API function
async function fetchPage(pageNumber) {
  // Simulate network delay
  await new Promise(resolve => setTimeout(resolve, 500));

  // Simulated data pages (3 pages total)
  const pages = {
    1: ['item1', 'item2', 'item3'],
    2: ['item4', 'item5', 'item6'],
    3: ['item7', 'item8']
  };
```

```
  // Simulate no more pages
  if (!pages[pageNumber]) {
    return null;
  }

  return pages[pageNumber];
}

// Async generator to fetch pages one by one
async function* paginateAPI() {
  let page = 1;
  while (true) {
    try {
      const data = await fetchPage(page);
      if (!data) break; // No more pages, exit loop

      yield data;        // Yield current page data
      page++;
    } catch (error) {
      console.error('Error fetching page:', error);
      break;            // Stop iteration on error
    }
  }
}
```

7.5.2 Consuming Paginated Data with `for await...of`

Now, use the async iterator with `for await...of` to process the pages as they arrive:

```
(async () => {
  try {
    for await (const pageData of paginateAPI()) {
      console.log('Received page:', pageData);
      // Process pageData here (e.g., render UI, aggregate results)
    }
    console.log('All pages fetched.');
  } catch (error) {
    console.error('Error during pagination:', error);
  }
})();
```

7.5.3 How This Works

- The `paginateAPI` async generator fetches one page at a time.
- Each `yield` pauses the generator and hands control back to the consumer.
- The `for await...of` loop waits for each page's data asynchronously before continuing.
- Errors in fetching are caught inside the generator and stop iteration gracefully.
- The consumer can handle unexpected errors with an outer `try/catch`.

7.5.4 Benefits of This Approach

- **Memory efficient:** Only one page is fetched and processed at a time.
- **Readable and linear:** The code looks synchronous but handles async fetching seamlessly.
- **Easy error handling:** Errors can be managed locally within the generator or globally in the consumer.
- **Flexible:** This pattern works well for real paginated APIs, streams, or large datasets.

7.5.5 Try It Yourself

- Copy and paste this code into a modern browser console or Node.js environment.
- Modify `fetchPage` to simulate delays, errors, or different data.
- Experiment with processing logic inside the `for await...of` loop.

This example shows how async iterators empower you to handle complex async sequences, like paginated API data, with clean, manageable code.

Chapter 8.

Event Emitters and Callbacks

1. Introduction to EventEmitter in Node.js
2. Emitting and Listening to Events
3. Practical Example: Custom Event-Driven Module

8 Event Emitters and Callbacks

8.1 Introduction to EventEmitter in Node.js

In Node.js, the **EventEmitter** class is a core part of how asynchronous events are handled. It provides a powerful way to create and manage events, enabling an event-driven programming model that helps build scalable, efficient applications.

8.1.1 What Is EventEmitter?

At its core, **EventEmitter** is a class that allows objects to **emit named events** and **register listeners** (handlers) that react when those events occur. This means instead of calling a callback directly, your code can emit an event, and any number of listeners can respond independently.

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
```

8.1.2 The Event-Driven Programming Model

Event-driven programming is a paradigm where the flow of the program is determined by events — user actions, messages from other programs, or asynchronous operations like network requests or timers.

With EventEmitter, you **subscribe** to specific events by adding listeners, and you **emit** those events when something interesting happens:

Full runnable code:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

emitter.on('dataReceived', (data) => {
  console.log('Data received:', data);
});

emitter.emit('dataReceived', 'Hello, world!');
```

This model decouples the event source from event handlers, making code more modular and easier to maintain.

8.1.3 How EventEmitter Differs from Traditional Callbacks

Callbacks are functions passed directly to asynchronous operations and invoked when those operations complete. While simple, callbacks tightly couple the async operation with a single handler, making complex coordination and multiple listeners cumbersome.

In contrast, EventEmitter allows multiple listeners to react to the same event independently, and listeners can be added or removed dynamically at runtime. This flexibility supports more scalable and loosely coupled applications, especially when multiple components need to react to the same event.

8.1.4 Why EventEmitter Is Important for Scalable Apps

- **Decoupling:** EventEmitters promote loose coupling between modules.
- **Multiple listeners:** Many parts of an app can listen and react to the same event.
- **Asynchronous event handling:** Events fit naturally with Node.js's non-blocking, single-threaded model.
- **Flexibility:** Listeners can be added, removed, or modified on the fly.

EventEmitter is the foundation for many built-in Node.js modules (like streams, HTTP servers, and timers) and is essential for designing robust, responsive applications that handle multiple async tasks elegantly.

8.1.5 Summary

The EventEmitter class is a cornerstone of Node.js's asynchronous event handling. It enables an event-driven programming style where events trigger multiple independent listeners, offering a flexible alternative to callbacks and helping you build scalable, maintainable applications.

8.2 Emitting and Listening to Events

The **EventEmitter** class in Node.js lets you create event-driven systems by emitting events and registering listeners that respond when those events occur. This section explains how to use EventEmitter to register listeners, emit events, and manage listeners effectively.

8.2.1 Creating an EventEmitter Instance

To start using events, you first create an instance of the EventEmitter class:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
```

This `emitter` object can now emit events and register listeners.

8.2.2 Registering Event Listeners with `.on()`

To react to an event, you register a listener function using the `.on()` method:

```
emitter.on('greet', (name) => {
  console.log(`Hello, ${name}!`);
});
```

This listener will be called every time the `greet` event is emitted.

8.2.3 Emitting Events with `.emit()`

You trigger (or emit) an event using `.emit()`, optionally passing data to listeners:

```
emitter.emit('greet', 'Alice'); // Output: Hello, Alice!
```

When `greet` is emitted, all registered listeners for that event are invoked synchronously, in the order they were registered.

8.2.4 Multiple Listeners for the Same Event

EventEmitter supports multiple listeners per event:

```
emitter.on('greet', (name) => {
  console.log(`How are you, ${name}?`);
});

emitter.emit('greet', 'Bob');
// Output:
// Hello, Bob!
// How are you, Bob?
```

Each listener receives the event data and runs independently.

8.2.5 One-Time Listeners with `.once()`

Sometimes you want a listener to run only **once**, then automatically remove itself. Use `.once()` for this:

```
emitter.once('welcome', () => {
  console.log('Welcome event fired!');
});

emitter.emit('welcome'); // Logs once
emitter.emit('welcome'); // No output this time
```

This is useful for events that should trigger a single action, like initial setup or connection events.

8.2.6 Removing Listeners

You can remove listeners with `.off()` or `.removeListener()`:

```
function onGreet(name) {
  console.log(`Hi again, ${name}!`);
}

emitter.on('greet', onGreet);
emitter.emit('greet', 'Carol'); // Calls both listeners

emitter.off('greet', onGreet); // Remove the specific listener
emitter.emit('greet', 'Carol'); // Calls only remaining listeners
```

Removing listeners helps avoid memory leaks and unwanted behavior.

8.2.7 Summary

- Use `.on(event, listener)` to register listeners called every time an event is emitted.
- Use `.emit(event, ...args)` to trigger events and pass data to listeners.
- Multiple listeners can respond to the same event.
- Use `.once(event, listener)` for listeners that run only once.
- Use `.off(event, listener)` to remove listeners when no longer needed.

These methods give you full control over event-driven communication in your Node.js apps.

8.3 Practical Example: Custom Event-Driven Module

To see EventEmitter in action, let's build a simple custom module that simulates a **task runner**. This module emits events to signal status updates and data results, showing how event-driven design can organize asynchronous operations cleanly.

8.3.1 Creating the Custom Module

Create a file named `taskRunner.js`:

```
const EventEmitter = require('events');

class TaskRunner extends EventEmitter {
  constructor() {
    super();
  }

  runTasks(tasks) {
    this.emit('start', `Starting ${tasks.length} tasks...`);

    tasks.forEach((task, index) => {
      setTimeout(() => {
        // Simulate task completion with a result
        const result = `Result of task ${index + 1}: ${task()}`;
        this.emit('taskComplete', result);

        // When last task finishes, emit 'done'
        if (index === tasks.length - 1) {
          this.emit('done', 'All tasks completed!');
        }
      }, 1000 * (index + 1)); // staggered completion
    });
  }
}

module.exports = TaskRunner;
```

8.3.2 Using the Module and Subscribing to Events

Now, create a separate file (e.g., `app.js`) to use the `TaskRunner` module:

```
const TaskRunner = require('./taskRunner');

const runner = new TaskRunner();

// Register listeners for different events
runner.on('start', (message) => {
  console.log('Status:', message);
});
```

```
runner.on('taskComplete', (result) => {
  console.log('Task finished:', result);
});

runner.once('done', (message) => {
  console.log('Status:', message);
});

// Define some sample tasks
const tasks = [
  () => 'Clean database',
  () => 'Send emails',
  () => 'Generate reports'
];

// Start running tasks
runner.runTasks(tasks);
```

8.3.3 What Happens Here?

- When `runTasks()` is called, the `start` event notifies listeners that tasks are beginning.
- Each task simulates asynchronous completion with `setTimeout`. As each completes, it emits a `taskComplete` event with its result.
- When all tasks finish, the `done` event fires once.
- The event listeners in `app.js` respond accordingly, printing messages as the tasks progress.

8.3.4 Benefits of This Event-Driven Design

- **Decoupling:** The `TaskRunner` class doesn't need to know who listens or how results are handled.
- **Scalability:** Adding or removing listeners is easy without changing the core module.
- **Flexibility:** Multiple parts of your app can react differently to the same events.
- **Clean async flow:** Events signal progress without complex callbacks or Promise chains.

8.3.5 Try It Yourself

- Save both files and run `node app.js` in your terminal.
- Modify the tasks or add new event listeners to customize behavior.
- Experiment with emitting additional events like `error` for robust handling.

This example illustrates how EventEmitter empowers you to write modular, maintainable, and reactive code—making asynchronous JavaScript apps easier to build and understand.

Chapter 9.

Timers and Scheduling

1. Using `setTimeout()`, `setInterval()`, and `clearTimeout()`
2. Throttling and Debouncing Techniques
3. Practical Example: Debounced Input Validation

9 Timers and Scheduling

9.1 Using `setTimeout()`, `setInterval()`, and `clearTimeout()`

JavaScript provides powerful timer functions—`setTimeout()`, `setInterval()`, `clearTimeout()`, and `clearInterval()`—to schedule code execution after delays or at regular intervals. These are essential tools for controlling asynchronous timing in your programs.

9.1.1 `setTimeout()`: Delayed Execution

`setTimeout()` lets you run a function once after a specified delay (in milliseconds):

```
const timerId = setTimeout(() => {
  console.log('This runs after 2 seconds');
}, 2000);
```

Here, the callback function executes approximately 2 seconds later. The `setTimeout` returns a timer ID that you can use to cancel the timer if needed.

9.1.2 `setInterval()`: Repeated Execution

`setInterval()` runs a function repeatedly, with a fixed delay between each call:

```
const intervalId = setInterval(() => {
  console.log('This runs every 1 second');
}, 1000);
```

This will print the message every second until stopped.

9.1.3 Canceling Timers with `clearTimeout()` and `clearInterval()`

If you need to stop a scheduled timeout or interval, use `clearTimeout()` or `clearInterval()` respectively:

```
clearTimeout(timerId);    // Cancels the delayed execution
clearInterval(intervalId); // Stops the repeated execution
```

This is important for managing resources, preventing unwanted behavior, or cleaning up timers when no longer needed.

9.1.4 Common Pitfalls: Timer Drift and the Event Loop

- **Timer Drift:** Neither `setTimeout` nor `setInterval` guarantees exact timing. Delays may be longer than requested due to CPU load, other code executing, or browser throttling.
- **Event Loop Impact:** JavaScript's single-threaded event loop processes timers only when the call stack is empty. If your program is busy running other tasks, timers will be delayed.
- **Example:** If you set `setInterval` for every 1000ms but your code is busy, actual intervals might be longer, causing "drift" over time.

9.1.5 Practical Tips

- Use `setTimeout` over `setInterval` when possible for repeated tasks by recursively scheduling the next call. This approach helps avoid overlapping executions:

Full runnable code:

```
function repeatTask() {
  console.log('Task running...');
  setTimeout(repeatTask, 1000);
}
repeatTask();
```

- Always clear timers when they're no longer needed to avoid memory leaks or unexpected behavior.

9.1.6 Summary

- `setTimeout(fn, delay)` schedules a one-time delayed execution.
- `setInterval(fn, interval)` schedules repeated execution every interval milliseconds.
- Use `clearTimeout(id)` and `clearInterval(id)` to cancel scheduled timers.
- Timers are influenced by JavaScript's single-threaded event loop and system load, causing delays or drift.
- Careful timer management is essential for reliable and efficient async code.

9.2 Throttling and Debouncing Techniques

When working with events like `scroll`, `resize`, or `input`, your event handler might get called **many times per second**, which can hurt performance and cause janky user experiences. To manage this, two popular techniques—**throttling** and **debouncing**—help optimize how often your event handlers run.

9.2.1 What Is Throttling?

Throttling ensures a function runs **at most once** in a specified time interval, no matter how many times the event fires. It spreads out execution, limiting how often your callback runs.

When to use:

- Useful when you want to respond at a regular pace while the event is firing continuously.
- Example: Updating the position of an animation during scrolling or window resizing.

9.2.2 Throttling Example

Here's a simple throttling implementation using `setTimeout`:

```
function throttle(fn, delay) {
  let lastCall = 0;
  return function(...args) {
    const now = Date.now();
    if (now - lastCall >= delay) {
      lastCall = now;
      fn.apply(this, args);
    }
  };
}

// Usage: Log scroll position at most every 200ms
window.addEventListener('scroll', throttle(() => {
  console.log('Scrolled at:', window.scrollY);
}, 200));
```

9.2.3 What Is Debouncing?

Debouncing delays the execution of a function until after the event has stopped firing for a specified amount of time. This means the function only runs **once after the user stops triggering the event**.

When to use:

- Ideal for input validation, search autocomplete, or resize events where you only want to react once after the user finishes typing or resizing.

9.2.4 Debouncing Example

Here's a basic debounce function using `setTimeout` and `clearTimeout`:

```
function debounce(fn, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {
      fn.apply(this, args);
    }, delay);
  };
}

// Usage: Validate input only after 300ms pause in typing
const input = document.querySelector('input');
input.addEventListener('input', debounce(() => {
  console.log('Validating input:', input.value);
}, 300));
```

9.2.5 Key Differences

Feature	Throttling	Debouncing
Executes	At regular intervals while event fires	Only once, after event stops firing
Use case example	Scrolling, resizing, animations	Text input validation, search
Goal	Limit execution frequency	Delay execution until inactivity

9.2.6 Why Use These Techniques?

- **Performance:** Avoid expensive computations running too often.
- **User Experience:** Smooth out UI updates and reduce jitter.
- **Resource Management:** Prevent overwhelming APIs or systems.

By applying throttling or debouncing thoughtfully, your app stays responsive and efficient even during rapid event firing.

9.2.7 Summary

- **Throttle** limits how often a function runs during rapid events.
- **Debounce** delays execution until the event stops firing.
- Both improve performance and user experience for event-heavy interactions.
- Use throttling for steady interval updates; use debouncing for waiting until "quiet" periods.

Mastering these patterns helps build smoother, more performant asynchronous applications in JavaScript.

9.3 Practical Example: Debounced Input Validation

Validating user input in real time can improve user experience by providing instant feedback. However, validating on every keystroke can be costly and inefficient. This is where **debouncing** helps — it delays the validation until the user stops typing for a set amount of time.

9.3.1 Example: Debounced Input Validation

Below is a runnable example that validates an email input field. The validation runs only after the user stops typing for 500 milliseconds, reducing unnecessary validations.

HTML:

```
<form>
  <label for="email">Email:</label>
  <input type="text" id="email" />
  <p id="message"></p>
</form>
```

Javascript:

```
// Debounce function: delays the execution of 'fn' until 'delay' ms after last call
function debounce(fn, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {
      fn.apply(this, args);
    }, delay);
  };
}

// Simple email validation function
function validateEmail(email) {
  const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return regex.test(email);
```

```
    }

    // Validation logic to run after debounced delay
    function handleValidation(event) {
      const email = event.target.value;
      const message = document.getElementById('message');

      if (email === '') {
        message.textContent = '';
        return;
      }

      if (validateEmail(email)) {
        message.textContent = 'Valid email YES';
        message.style.color = 'green';
      } else {
        message.textContent = 'Invalid email NO';
        message.style.color = 'red';
      }
    }

    // Attach debounced validation to input event
    const emailInput = document.getElementById('email');
    emailInput.addEventListener('input', debounce(handleValidation, 500));
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Debounced Input Validation</title>
</head>
<body>
  <form>
    <label for="email">Email:</label>
    <input type="text" id="email" />
    <p id="message"></p>
  </form>

  <script>
    // Debounce function: delays the execution of 'fn' until 'delay' ms after last call
    function debounce(fn, delay) {
      let timeoutId;
      return function(...args) {
        clearTimeout(timeoutId);
        timeoutId = setTimeout(() => {
          fn.apply(this, args);
        }, delay);
      };
    }

    // Simple email validation function
    function validateEmail(email) {
      const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
      return regex.test(email);
    }
```

```
    // Validation logic to run after debounced delay
    function handleValidation(event) {
      const email = event.target.value;
      const message = document.getElementById('message');

      if (email === '') {
        message.textContent = '';
        return;
      }

      if (validateEmail(email)) {
        message.textContent = 'Valid email YES';
        message.style.color = 'green';
      } else {
        message.textContent = 'Invalid email NO';
        message.style.color = 'red';
      }
    }

    // Attach debounced validation to input event
    const emailInput = document.getElementById('email');
    emailInput.addEventListener('input', debounce(handleValidation, 500));
  </script>
</body>
</html>
```

9.3.2 How It Works

- The `debounce` function wraps `handleValidation`, ensuring validation runs only after 500ms of inactivity.
- On each keystroke, the previous timer is cleared, and a new timer starts.
- When the user stops typing for 500ms, the email validation runs.
- Feedback is displayed below the input, turning green if valid, red if invalid.
- Empty input clears the message immediately.

9.3.3 Suggestions for Further Enhancements

- **More complex validation:** Add checks for domain-specific rules or MX records.
- **Async validation:** Extend to validate with server-side calls (e.g., checking if email is already registered), wrapping those calls in promises and debouncing accordingly.
- **UI improvements:** Disable submit button until input is valid.
- **Accessibility:** Add ARIA attributes to improve feedback for screen readers.

9.3.4 Summary

Debouncing input validation reduces unnecessary work and improves user experience by validating only when the user pauses typing. This pattern is especially helpful for form-heavy applications where responsiveness matters. Experiment with the delay time and validation rules to best fit your app's needs!

Chapter 10.

Web APIs and Asynchronous Browser Features

1. Fetch API Basics and Streaming Responses
2. XMLHttpRequest vs Fetch
3. Using Service Workers and Background Sync
4. Practical Example: Building a Progressive Web App Offline Cache

10 Web APIs and Asynchronous Browser Features

10.1 Fetch API Basics and Streaming Responses

The **Fetch API** is a modern, promise-based interface for making asynchronous HTTP requests in the browser. It simplifies the process of fetching resources, such as JSON data, images, or files, compared to older approaches like `XMLHttpRequest` (XHR).

10.1.1 Why Use Fetch?

Before Fetch, XHR was the primary method for HTTP requests, but its complex API and callback-heavy nature made code harder to write and maintain. Fetch improves on this by:

- Returning **Promises** that enable cleaner async handling with `.then()` or `async/await`.
- Providing a simpler, more intuitive API.
- Supporting streaming responses, allowing handling of large data efficiently.

10.1.2 Basic Syntax

Fetch accepts a URL (and optionally a configuration object) and returns a Promise that resolves to a `Response` object:

```
fetch(url, options)
  .then(response => {
    // handle response here
  })
  .catch(error => {
    // handle network errors here
  });
```

10.1.3 GET and POST Requests

- **GET request:**

```
fetch('https://api.example.com/data')
  .then(response => response.json())  // Parse JSON body
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

- **POST request:** Add a `method` and `body` in the options object:

```
fetch('https://api.example.com/data', {
  method: 'POST',
  headers: {
```

```
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({ username: 'user1', password: 'pass123' }),
})
  .then(response => response.json())
  .then(data => console.log('Success:', data))
  .catch(error => console.error('Error:', error));
```

10.1.4 Handling Streaming Responses

For large resources, waiting for the entire response before processing can be inefficient or cause delays. The Fetch API supports **streaming responses** by providing a readable stream that can be processed chunk-by-chunk as data arrives.

This lets you:

- Show progress to users.
- Process data on-the-fly (e.g., parsing logs or large JSON arrays incrementally).
- Reduce memory consumption by not loading the entire response at once.

10.1.5 Example: Fetching and Processing Streamed Data

Here's an example fetching a large text file and logging it in chunks:

Full runnable code:

```
fetch('https://example.com/large-text-file.txt')
  .then(response => {
    const reader = response.body.getReader();
    const decoder = new TextDecoder('utf-8');

    function read() {
      return reader.read().then(({ done, value }) => {
        if (done) {
          console.log('Stream complete');
          return;
        }
        console.log('Received chunk:', decoder.decode(value));
        return read(); // Read next chunk
      });
    }

    return read();
  })
  .catch(error => console.error('Fetch error:', error));
```

10.1.6 Summary

- The **Fetch API** is a clean, promise-based way to make HTTP requests in modern browsers.
- It handles GET and POST requests simply, with convenient JSON parsing.
- Streaming responses allow efficient processing of large data in chunks as it arrives.
- Fetch's readable streams open doors for building responsive apps that handle big data smoothly.

By mastering Fetch and streaming, you'll write faster, cleaner, and more modern asynchronous web code.

10.2 XMLHttpRequest vs Fetch

Before the Fetch API became widely adopted, the **XMLHttpRequest (XHR)** object was the primary method for making HTTP requests in JavaScript. Understanding the differences between these two approaches helps appreciate why Fetch has largely replaced XHR in modern web development.

10.2.1 XMLHttpRequest: The Old Standard

Introduced in the late 1990s, XMLHttpRequest allowed web pages to send and receive data asynchronously without refreshing. While revolutionary, XHR has some notable drawbacks:

- **Callback-based:** XHR uses event handlers like `onreadystatechange` or `onload` to process responses, which can lead to nested callbacks and harder-to-maintain code.
- **Verbose and less intuitive:** The API requires manually checking `readyState` and status codes, and handling data parsing (e.g., JSON) yourself.
- **Limited streaming:** XHR supports partial response handling via `responseText` in certain ready states, but it is not as powerful or flexible as Fetch streams.
- **Inconsistent behavior:** Some browsers historically had quirks or inconsistencies with XHR, making cross-browser compatibility challenging.

Here's a simple example of a GET request using XHR:

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com/data');
xhr.onreadystatechange = () => {
  if (xhr.readyState === 4) {
    if (xhr.status === 200) {
      const data = JSON.parse(xhr.responseText);
      console.log(data);
    } else {
```

```
      console.error('Request failed');
    }
  }
};
xhr.send();
```

10.2.2 Fetch: The Modern Alternative

The Fetch API modernizes HTTP requests with a **promise-based** approach:

- **Cleaner, more readable code:** Promises and `async/await` syntax reduce callback nesting and improve flow control.
- **Built-in JSON parsing:** The `Response` object provides convenient methods like `.json()`, `.text()`, and `.blob()`.
- **Streaming support:** Fetch can process large responses chunk-by-chunk using readable streams.
- **Better composability:** Promises make it easier to chain requests, handle errors, and coordinate multiple async operations.

The equivalent GET request using Fetch is much simpler:

Full runnable code:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Fetch error:', error));
```

10.2.3 Browser Compatibility and Use Cases

- **Browser support:** Fetch is widely supported in modern browsers, but some older browsers (notably Internet Explorer) do not support it natively. Polyfills are available for broader compatibility.
- **When to use XHR:**
  - Legacy projects that rely on XHR or have compatibility constraints.
  - Certain features like tracking upload progress via `xhr.upload.onprogress` which Fetch currently lacks.
- **When to use Fetch:**
  - New projects prioritizing clean, modern syntax and promise-based control flow.

– Applications needing streaming or advanced response handling.

10.2.4 Summary

Feature	XMLHttpRequest (XHR)	Fetch API
Async model	Callback-based	Promise-based
Syntax complexity	Verbose, nested callbacks	Clean, chainable promises
Response parsing	Manual	Built-in `.json()`, `.text()`, etc.
Streaming support	Limited	Powerful streaming with streams
Browser compatibility	Older browsers, IE support	Modern browsers; polyfills for old ones
Special use cases	Upload progress tracking	Better for modern async workflows

While XHR paved the way for asynchronous HTTP in JavaScript, Fetch provides a simpler, more powerful model that's now the preferred choice for most web developers.

10.3 Using Service Workers and Background Sync

Service workers are a powerful feature in modern browsers that allow JavaScript to run in the background, independent of the main web page. They enable critical asynchronous capabilities such as intercepting network requests, caching responses, and synchronizing data in the background—even when the user is offline.

10.3.1 What Is a Service Worker?

A service worker is a special type of web worker that acts as a **proxy between your web app and the network**. Once registered, it runs separately from the main thread and can intercept fetch requests, allowing you to cache assets or serve fallback content when offline.

Key characteristics:

- Runs independently of web pages.
- Is asynchronous and event-driven.
- Has no direct access to the DOM.
- Supports background features like push notifications and sync.

10.3.2 Lifecycle of a Service Worker

1. **Registration** – Your site registers the service worker using JavaScript.
2. **Installation** – The browser downloads and installs the service worker script.
3. **Activation** – After installation, the worker activates and takes control of pages.
4. **Fetch/Sync Events** – The worker listens for fetch events or background sync triggers.

```
// Registering a service worker
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js')
    .then(() => console.log('Service Worker registered'))
    .catch(console.error);
}
```

10.3.3 Caching Strategies

Service workers enable **programmatic caching** of resources using the Cache API. Common strategies include:

- **Cache First**: Serve cached content if available, fetch from network otherwise.
- **Network First**: Try network first, fall back to cache.
- **Stale While Revalidate**: Serve from cache, update in background.

```
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request).then(cachedResponse => {
      return cachedResponse || fetch(event.request);
    })
  );
});
```

This allows web apps to work offline, load faster, and reduce server load.

10.3.4 Background Sync

Background Sync enables your app to defer actions until the user has a stable internet connection. For example, you can queue form submissions or message sends while offline and replay them once online.

```
self.addEventListener('sync', event => {
  if (event.tag === 'sync-posts') {
    event.waitUntil(sendQueuedPosts());
  }
});
```

To request a sync event:

```
navigator.serviceWorker.ready.then(reg => {
  reg.sync.register('sync-posts');
});
```

This ensures reliability, especially for apps like chat, forms, or uploads.

10.3.5 Browser Support

- Most modern browsers (Chrome, Edge, Firefox, Safari) support service workers.
- Background Sync has more limited support (Chrome, Edge). Always feature-detect before using.

10.3.6 Conceptual Overview Diagram

```
[ Web Page ]
     |
     |   fetch()
     V
[ Service Worker ]
     |
     |  (fetch event → custom logic)
     V
[ Network ]       [ Cache Storage ]
```

10.3.7 Summary

Service workers are essential for building resilient, offline-capable web apps. By handling network requests and background tasks asynchronously, they bring native-like performance and reliability to web experiences. Combined with background sync, they allow apps to gracefully recover from lost connections and deliver a seamless user experience.

10.4 Practical Example: Building a Progressive Web App Offline Cache

In this section, we'll walk through creating a simple **Progressive Web App (PWA)** that uses a **service worker** to cache essential files and API responses. This allows the app to continue functioning even when offline—a core feature of PWAs.

10.4.1 Step 1: Register the Service Worker

In your main HTML file or JavaScript entry point (e.g., `main.js`), add the following code to register your service worker:

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js')
    .then(() => console.log('Service Worker registered successfully.'))
    .catch(error => console.error('Service Worker registration failed:', error));
}
```

> YES Tip: Make sure `sw.js` is located at the root or correctly scoped to control the app pages.

10.4.2 Step 2: Create the Service Worker (`sw.js`)

This service worker script handles three key events: `install`, `activate`, and `fetch`.

```
const CACHE_NAME = 'pwa-cache-v1';
const ASSETS_TO_CACHE = [
  '/',
  '/index.html',
  '/styles.css',
  '/app.js',
  '/offline.html'
];

// Install event - cache core assets
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(cache => {
        console.log('[Service Worker] Caching assets...');
        return cache.addAll(ASSETS_TO_CACHE);
      })
  );
});

// Activate event - clean up old caches
self.addEventListener('activate', event => {
  event.waitUntil(
    caches.keys().then(keys =>
      Promise.all(
        keys.filter(key => key !== CACHE_NAME)
            .map(key => caches.delete(key))
      )
    )
  );
});

// Fetch event - serve cached content or fallback
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request)
```

```
      .then(cached => cached || fetch(event.request)
        .catch(() => caches.match('/offline.html'))
      )
  );
});
```

10.4.3 Step 3: Test Offline Functionality

1. **Open DevTools** in your browser (F12), go to the **Application** tab.
2. Under **Service Workers**, check "Offline."
3. Refresh the page — if properly cached, your app should still load.
4. Try navigating to a cached or uncached page to test fallback behavior.

10.4.4 Step 4: Cache API Responses (Optional)

You can extend the fetch event handler to cache dynamic data, like API responses:

```
if (event.request.url.includes('/api/data')) {
  event.respondWith(
    fetch(event.request)
      .then(response => {
        const cloned = response.clone();
        caches.open(CACHE_NAME).then(cache => cache.put(event.request, cloned));
        return response;
      })
      .catch(() => caches.match(event.request))
  );
}
```

10.4.5 Debugging Tips

- Use **DevTools > Application > Service Workers** to inspect status and update workers.
- Use **Console logs** in `sw.js` to trace install/activate events.
- Clear caches with `caches.delete()` or via the **Clear Site Data** button.

10.4.6 Summary

This example demonstrates how to turn a basic web app into a reliable offline-capable PWA using service workers. By caching assets and optionally API data, your app can load and function seamlessly—even without a network connection. This improves user experience, performance, and resilience, especially on mobile or unreliable networks.

Chapter 11.

Handling Streams

1. Introduction to Streams: Readable, Writable, Duplex
2. Stream Events and Backpressure
3. Using Streams with Promises and Async Iterators
4. Practical Example: Reading and Writing Files with Streams

11 Handling Streams

11.1 Introduction to Streams: Readable, Writable, Duplex

Streams are powerful abstractions in JavaScript (especially in Node.js) that handle data flow **efficiently and incrementally**. Rather than waiting for an entire file or response to load into memory, streams allow you to **process data piece-by-piece** as it arrives—saving time and memory, especially with large data sources.

At a high level, streams are like **pipelines**: they enable chunks of data to move from one place to another in a continuous flow.

11.1.1 Why Streams Matter

Streams are essential for:

- **Reading large files** without loading them fully into memory.
- **Writing logs or responses** incrementally.
- **Handling network communication** in chunks.
- **Transforming data on the fly**, like compressing or encrypting a file during transfer.

They are **event-driven** and can be paused, resumed, or piped to other streams for modular, high-performance I/O.

11.1.2 Types of Streams

JavaScript (especially Node.js) supports four main types of streams:

1. **Readable Streams**
   - *Direction:* Data flows **into your program** (you consume it).
   - *Examples:* Reading from a file, receiving HTTP responses.
   - *Interface:* Emits `data`, `end`, and `error` events.

```
const fs = require('fs');
const readStream = fs.createReadStream('input.txt');
readStream.on('data', chunk => console.log('Chunk:', chunk.toString()));
```

2. **Writable Streams**
   - *Direction:* Data flows **out from your program** (you produce it).
   - *Examples:* Writing to a file, sending HTTP responses.
   - *Interface:* Accepts `.write()` and `.end()` methods.

```
const writeStream = fs.createWriteStream('output.txt');
writeStream.write('Hello, stream!');
writeStream.end();
```

3. **Duplex Streams**

   - *Direction:* Both readable and writable.
   - *Examples:* Sockets, transform streams.
   - A duplex stream implements both interfaces and can read and write independently.

```
const { Duplex } = require('stream');

const duplex = new Duplex({
  read(size) { this.push('Hello'); this.push(null); },
  write(chunk, encoding, callback) {
    console.log('Written:', chunk.toString());
    callback();
  }
});

duplex.on('data', data => console.log('Read:', data.toString()));
duplex.write('World');
```

11.1.3 Conceptual Diagram

```
[ Readable Stream ] ---> [ Writable Stream ]
     (Input)                (Output)

[ Duplex Stream ]
     <--- Read
     ---> Write
```

11.1.4 Summary

Streams allow for efficient, non-blocking processing of data in chunks, which is crucial for working with large files, APIs, and network protocols. Understanding readable, writable, and duplex streams sets the foundation for building scalable applications that work well with asynchronous data flow.

In the following sections, you'll learn how streams emit events, how to handle backpressure, and how to use them with Promises and async iteration for even cleaner code.

11.2 Stream Events and Backpressure

Streams are event-driven objects that emit signals as data flows through them. Understanding these events and how **backpressure** works is key to writing efficient, non-blocking applications in Node.js.

11.2.1 Common Stream Events

Streams emit various events to signal changes in state or data availability. Here are the most important ones:

1. **`data`** (Readable streams) Fired when a chunk of data is available for consumption.
```
readStream.on('data', chunk => {
  console.log('Received:', chunk.toString());
});
```

2. **`end`** (Readable streams) Emitted when no more data will be provided.
```
readStream.on('end', () => {
  console.log('No more data.');
});
```

3. **`error`** (Readable or Writable) Triggered when an error occurs during reading or writing.
```
stream.on('error', err => {
  console.error('Stream error:', err);
});
```

4. **`drain`** (Writable streams) Fired when a write buffer is emptied, signaling it's safe to write again.
```
writeStream.write(largeDataChunk, () => {
  if (!writeStream.write(moreData)) {
    writeStream.once('drain', () => {
      console.log('Buffer drained, writing resumed.');
    });
  }
});
```

11.2.2 What Is Backpressure?

Backpressure occurs when the writable side of a stream **can't keep up with the speed of the readable side**. If you write too much data too quickly, the writable stream's internal buffer fills up, and `write()` returns `false`.

To prevent overwhelming the system and running out of memory, you must **pause** the data source and wait until it's safe to resume.

11.2.3 Managing Backpressure: Pause and Resume

Node.js streams offer built-in methods to control flow:

- **`pause()`**: Temporarily stops the `data` event.
- **`resume()`**: Restarts the flow of data.

Example:

```
const fs = require('fs');
const readStream = fs.createReadStream('bigfile.txt');
const writeStream = fs.createWriteStream('copy.txt');

readStream.on('data', chunk => {
  const canContinue = writeStream.write(chunk);
  if (!canContinue) {
    readStream.pause(); // Stop reading if write buffer is full
    writeStream.once('drain', () => {
      readStream.resume(); // Resume when buffer is cleared
    });
  }
});

readStream.on('end', () => {
  writeStream.end();
});
```

This approach ensures your program doesn't crash or slow down due to excessive memory usage.

11.2.4 Summary

Stream events give you fine-grained control over how data is handled, especially in I/O-heavy applications. Properly managing **backpressure** is essential for creating **efficient**, **resilient**, and **scalable** data pipelines. By listening for `drain`, `pause()`, and `resume()` events, your application can balance input and output speed, avoiding bottlenecks and resource exhaustion.

11.3 Using Streams with Promises and Async Iterators

Modern JavaScript allows you to work with streams in a **promise-friendly** and **readable** way by combining them with **async iterators**. This makes it easier to process streamed data using `for await...of` loops, improving the clarity of asynchronous I/O operations.

11.3.1 Streams and Async Iteration

Node.js readable streams (from version 10+) implement the async iterator protocol. This means you can use `for await...of` to read chunks of data from a stream **asynchronously** without needing to manually handle events like `data` and `end`.

Here's a basic example of reading a file using an async iterator:

```
const fs = require('fs');

async function readFileLineByLine(filePath) {
  const stream = fs.createReadStream(filePath, { encoding: 'utf-8' });

  for await (const chunk of stream) {
    console.log('Chunk:', chunk);
  }

  console.log('File reading completed.');
}

readFileLineByLine('./sample.txt');
```

This syntax avoids `on('data')` and `on('end')` listeners, making the flow cleaner and easier to follow.

11.3.2 Converting Legacy Streams to Async Iterators

If you're working with a custom or older stream that doesn't support async iteration, you can manually convert it using a helper:

```
async function* streamToAsyncIterator(stream) {
  const reader = new Promise((resolve, reject) => {
    stream.on('data', chunk => resolve({ value: chunk, done: false }));
    stream.on('end', () => resolve({ done: true }));
    stream.on('error', err => reject(err));
  });

  while (true) {
    const { value, done } = await reader;
    if (done) break;
    yield value;
  }
}
```

Use it like this:

```
for await (const chunk of streamToAsyncIterator(myStream)) {
  console.log(chunk.toString());
}
```

11.3.3 Example: Streaming and Writing with Promises

Let's say you want to read a large file and write it to another file using async iteration and Promises:

Full runnable code:

```
const fs = require('fs');

async function copyFile(source, target) {
  const reader = fs.createReadStream(source);
  const writer = fs.createWriteStream(target);

  for await (const chunk of reader) {
    const ok = writer.write(chunk);
    if (!ok) {
      await new Promise(resolve => writer.once('drain', resolve));
    }
  }

  writer.end();
}

copyFile('input.txt', 'output.txt');
```

This method handles **backpressure** gracefully using Promises and avoids nesting callbacks or handling multiple events manually.

11.3.4 Summary

Using **async iterators with streams** provides a powerful and elegant way to manage asynchronous I/O in Node.js. It simplifies control flow, enhances readability, and integrates well with modern async/await patterns, making it easier to build scalable and maintainable applications.

11.4 Practical Example: Reading and Writing Files with Streams

When working with large files, loading the entire file into memory using traditional methods like `fs.readFile()` can lead to memory exhaustion and performance issues. Streams offer a powerful alternative by **processing data in chunks**, which allows efficient and scalable file I/O operations.

11.4.1 Goal

We'll build a simple Node.js program that reads a large file using a **readable stream** and writes its contents to another file using a **writable stream**. We'll also include proper **error handling** and explain why this method is more efficient.

11.4.2 Complete Example

```
const fs = require('fs');
const path = require('path');

// Define input and output file paths
const inputPath = path.join(__dirname, 'large-input.txt');
const outputPath = path.join(__dirname, 'large-output.txt');

// Create read and write streams
const readStream = fs.createReadStream(inputPath, { encoding: 'utf-8' });
const writeStream = fs.createWriteStream(outputPath, { encoding: 'utf-8' });

// Handle streaming data
readStream.on('data', chunk => {
  const canContinue = writeStream.write(chunk);
  if (!canContinue) {
    // Pause reading if write buffer is full
    readStream.pause();
    writeStream.once('drain', () => readStream.resume());
  }
});

// Handle errors
readStream.on('error', err => {
  console.error('Read error:', err.message);
});
writeStream.on('error', err => {
  console.error('Write error:', err.message);
});

// End the write stream once reading is done
readStream.on('end', () => {
  writeStream.end();
  console.log('File copy completed.');
});
```

11.4.3 How It Works

- **`fs.createReadStream`** and **`fs.createWriteStream`** allow you to process the file incrementally, chunk by chunk.
- The `data` event gives access to each chunk read from the file.

- The `write()` method returns `false` when the internal buffer is full, and `drain` signals when it's safe to resume writing.
- `pause()` and `resume()` manage the flow to avoid overwhelming the writable stream.

11.4.4 Error Handling

Errors may occur due to permission issues, missing files, or disk problems. Listening to `error` events on both streams ensures that such issues are caught and reported properly, preventing unhandled exceptions.

11.4.5 Performance Benefits

Using streams:

- Reduces **memory usage** by avoiding full-file buffering.
- Supports **non-blocking I/O**, which means other operations can continue while the file is being processed.
- Scales better for **large files**, such as logs, videos, or large datasets.

Compared to `fs.readFile()`, which loads everything into RAM before acting, streams enable your application to stay responsive and efficient, even when working with gigabytes of data.

11.4.6 Conclusion

Streams are essential for efficient file I/O in Node.js. This example showcases how to use them effectively to read and write large files with minimal memory usage, smooth error handling, and controlled data flow — critical for building high-performance Node.js applications.

Chapter 12.

Error Handling and Debugging Asynchronous Code

1. Common Pitfalls and Anti-Patterns
2. Debugging Async Code with Breakpoints and Logs
3. Handling Promise Rejections Gracefully
4. Practical Example: Robust Error Handling in Async Workflows

12 Error Handling and Debugging Asynchronous Code

12.1 Common Pitfalls and Anti-Patterns

Asynchronous programming in JavaScript is powerful but can be error-prone if not used carefully. Several common mistakes can lead to hard-to-debug issues, memory leaks, or unexpected behavior. This section highlights some of the most frequent **anti-patterns** and explains how to avoid them.

Unhandled Promise Rejections

A common mistake is to create Promises without handling potential rejections. Since Promises are asynchronous, unhandled errors might not be visible until runtime and can crash Node.js in future versions.

```
// Anti-pattern: No rejection handler
fetchData(); // Returns a Promise
```

If `fetchData()` fails, the rejection is not caught.

YES Solution: Always handle rejections with `.catch()` or `try/catch` in async functions.

```
fetchData().catch(error => {
  console.error('Failed to fetch data:', error);
});
```

Or using `async/await`:

```
try {
  const data = await fetchData();
} catch (error) {
  console.error('Error:', error);
}
```

Mixing Callbacks and Promises

Combining both styles in the same code leads to confusion and error-prone logic, especially if callbacks are executed alongside Promises.

```
// Anti-pattern: Mixing styles
doSomething((err, result) => {
  if (err) return;
  somePromise().then(data => {
    // Callback + Promise confusion
  });
});
```

YES Solution: Stick to one async pattern per module.

If the API uses callbacks, consider **promisifying** it using `util.promisify`:

```
const util = require('util');
const doSomethingAsync = util.promisify(doSomething);

doSomethingAsync().then(...).catch(...);
```

Swallowing Errors Silently

Sometimes developers use empty `catch()` blocks or don't log errors, making debugging nearly impossible.

```
// Anti-pattern: Silent catch
someAsyncTask().catch(() => {}); // No logging or recovery
```

YES Solution: Always log or respond to errors.

```
someAsyncTask().catch(err => {
  console.error('Unhandled error:', err);
});
```

Forgetting to Return Promises

When chaining Promises, forgetting to return them can break control flow or lead to premature termination.

```
// Anti-pattern: No return
function saveData(data) {
  validate(data)
    .then(() => writeToDB(data)); // This returns nothing
}
```

YES Solution: Return the Promise chain.

```
function saveData(data) {
  return validate(data)
    .then(() => writeToDB(data));
}
```

12.1.1 Summary

By avoiding these common pitfalls—**unhandled rejections, mixed async models, silent failures, and broken chains**—you'll write more robust, maintainable asynchronous code. Following consistent patterns and leveraging proper error handling ensures your application is resilient and easier to debug.

12.2 Debugging Async Code with Breakpoints and Logs

Debugging asynchronous JavaScript can be tricky because code doesn't always execute in the order it's written. Promises, async/await, timers, and events all introduce non-linear

execution, making it harder to trace issues. Fortunately, modern tools and techniques make debugging async code much easier.

12.2.1 Use Console Logs Effectively

The most basic and widely used debugging method is `console.log()`. Strategic logging helps you:

- Track the flow of async functions.
- Monitor variable values before and after async calls.
- Identify where a failure or unexpected result occurs.

Tip: Label logs clearly and use timestamps when needed.

```
console.log('Start fetching...');
fetchData()
  .then(data => {
    console.log('Data received:', data);
  })
  .catch(err => {
    console.error('Fetch failed:', err);
  });
```

12.2.2 Set Breakpoints in Async Code

Both browser developer tools (like Chrome DevTools) and Node.js have built-in debuggers that allow setting breakpoints in async code.

In the Browser:

- Open DevTools (F12 or right-click → Inspect).
- Go to the **Sources** tab.
- Find your JavaScript file and click the line number to set a breakpoint.
- Run the async code and DevTools will pause at the breakpoint, allowing step-by-step inspection.

Async/Await: You can step *into* and *over* `await` statements to watch how control returns after the Promise resolves.

In Node.js:

Use the `inspect` flag:

```
node --inspect-brk script.js
```

Then open `chrome://inspect` in Chrome to attach to the debugger. You can now pause execution, set breakpoints, and step through asynchronous flows.

12.2.3 Trace Async Call Stacks

Async code doesn't always preserve the full call stack. However, many modern environments show **async stack traces**, especially for Promises and async/await.

Tip: Add labels to Promises to clarify what part of the code is running.

```
someAsyncTask()
  .then(result => {
    console.log('[step1] result:', result);
    return process(result);
  })
  .then(output => {
    console.log('[step2] processed:', output);
  })
  .catch(err => {
    console.error('[error]', err);
  });
```

12.2.4 Advanced Tools

- `console.trace()`: Logs the current stack trace.
- **Chrome DevTools async call stacks**: Enable "Async Stack Traces" under DevTools settings for better context.
- **VS Code Debugger**: Use breakpoints, watch expressions, and async call tracing with `.vscode/launch.json` setup.

12.2.5 Summary

Debugging async code requires awareness of non-linear execution and good use of tools. Combine **breakpoints** for interactive step-through, **logs** for visibility, and **stack traces** for error context. Mastering these strategies will save hours of frustration and help you confidently trace bugs in complex async logic.

12.3 Handling Promise Rejections Gracefully

In asynchronous JavaScript, **unhandled Promise rejections** are a common source of bugs and crashes. Whether you're working with `.then()` chains or `async/await`, it's essential to anticipate and handle errors in a structured and user-friendly way.

12.3.1 Local Error Handling with `.catch()`

When using Promises directly, you should always end your `.then()` chain with a `.catch()` block to handle any errors that occur in the chain.

```
fetchData()
  .then(processData)
  .then(displayResult)
  .catch(error => {
    console.error('Something went wrong:', error.message);
    showErrorToUser('Unable to load data. Please try again later.');
  });
```

This pattern ensures that if any part of the chain throws, the error will be caught and can be reported or logged.

12.3.2 Error Handling in `async/await` with `try/catch`

When using `async/await`, use `try/catch` blocks to handle exceptions just like synchronous code.

```
async function loadAndDisplayData() {
  try {
    const data = await fetchData();
    const result = await processData(data);
    displayResult(result);
  } catch (error) {
    console.error('Error loading data:', error);
    showErrorToUser('Data could not be loaded.');
  }
}
```

This approach keeps error handling localized, clear, and easy to manage.

12.3.3 Global Error Handling in Node.js

Sometimes, a Promise may be rejected without a catch handler. In Node.js, unhandled rejections can terminate the process unless handled globally.

You can catch these using the `unhandledRejection` event:

```
process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled Rejection:', reason);
  // Optionally log or clean up
});
```

This provides a last-resort catch-all to prevent your app from silently crashing. However, you should **always prefer local handling** whenever possible.

12.3.4 Recovery and User Experience

Good error handling goes beyond logging. Always aim to:

- Show **user-friendly messages** (avoid raw error stacks).
- Provide **fallbacks** (e.g., retry, offline mode, cached data).
- Avoid exposing sensitive information in logs or UI.

For example:

```
function showErrorToUser(message) {
  document.querySelector('#status').textContent = message;
}
```

12.3.5 Summary

Graceful error handling in Promises involves catching errors at the right level—either via `.catch()` or `try/catch`—and managing unhandled rejections globally when needed. More importantly, errors should be reported meaningfully to users, and your code should aim to fail safely and recover when possible. This leads to more resilient, user-friendly applications.

12.4 Practical Example: Robust Error Handling in Async Workflows

Robust error handling in asynchronous workflows involves anticipating failures, catching exceptions at the right places, and responding with meaningful actions or fallbacks. Below is a runnable example that simulates fetching user data from an API, processing it, and displaying it to the user—with full error handling and recovery mechanisms.

Example: Fetching and Displaying User Data with Fallback

Full runnable code:

```
// Simulated async functions
function fetchUserData(userId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulate a 50% chance of failure
      if (Math.random() > 0.5) {
        resolve({ id: userId, name: "Alice" });
      } else {
        reject(new Error("Failed to fetch user data."));
      }
    }, 1000);
  });
}
```

```
function fetchUserSettings(userId) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve({ theme: "dark", notifications: true });
    }, 500);
  });
}

// Fallback function in case of data fetch failure
function getDefaultUserData() {
  return { id: 0, name: "Guest" };
}

// Display function (simulated UI)
function displayUserInfo(user, settings) {
  console.log("User Info:");
  console.log(`Name: ${user.name}`);
  console.log(`Theme: ${settings.theme}`);
  console.log(`Notifications: ${settings.notifications ? "On" : "Off"}`);
}

// Error display handler
function showErrorToUser(message) {
  console.error("UI Error:", message);
}

// Main async function with robust error handling
async function loadUserProfile(userId) {
  let user;
  try {
    // Try to fetch user data
    user = await fetchUserData(userId);
  } catch (error) {
    console.warn("Warning:", error.message);
    showErrorToUser("Could not load user profile. Using guest data.");
    // Use fallback user data
    user = getDefaultUserData();
  }

  try {
    // Even if user data fails, still try to load settings
    const settings = await fetchUserSettings(user.id);
    displayUserInfo(user, settings);
  } catch (error) {
    showErrorToUser("Failed to load user settings.");
    console.error("Settings error:", error.message);
  }
}

// Run the example
loadUserProfile(123);
```

What This Demonstrates:

- **Local try/catch blocks** handle failures close to their source.
- **Fallback logic** ensures the application remains usable even after errors.
- **Separation of concerns**: fetching, displaying, and error messaging are modular.

- **User feedback** is shown without revealing technical errors.

12.4.1 Summary

This pattern is crucial for real-world apps where network or processing failures are common. By isolating failure points, providing default behavior, and clearly communicating with users, you can build resilient asynchronous applications that degrade gracefully and maintain a good user experience.

Chapter 13.

Parallelism, Concurrency, and Workers

1. Differences Between Parallelism and Concurrency
2. Web Workers in Browser JavaScript
3. Worker Threads in Node.js
4. Practical Example: Offloading CPU-Intensive Tasks

13 Parallelism, Concurrency, and Workers

13.1 Differences Between Parallelism and Concurrency

Understanding the difference between **parallelism** and **concurrency** is fundamental when working with JavaScript's asynchronous model, especially as applications grow in complexity and need better performance and scalability.

What is Parallelism?

Parallelism means executing multiple tasks **at the exact same time**, truly simultaneously. Imagine a kitchen where multiple chefs are cooking different dishes at once, each on their own stove. The tasks run independently without waiting for others to finish, making full use of available resources like multiple CPU cores.

In computing, parallelism requires hardware support like multi-core processors. Each core can execute a separate task simultaneously. This can drastically reduce the total time for CPU-intensive or heavy workloads.

What is Concurrency?

Concurrency, on the other hand, is about **managing multiple tasks that can overlap in time**, but not necessarily running all at once. Picture a single chef juggling several dishes: chopping vegetables, stirring a pot, and preparing sauce — switching attention between tasks so everything progresses without delay. The chef handles tasks by quickly switching between them, giving the illusion of multitasking, but only truly doing one thing at a time.

In software, concurrency means organizing tasks so they efficiently share the CPU and resources by interleaving their execution. It involves managing asynchronous operations, handling I/O, timers, and events without blocking the main thread.

JavaScript and Its Single-Threaded Nature

JavaScript famously runs on a **single-threaded event loop**. This means only one piece of JavaScript code executes at a time. Because of this, JavaScript cannot achieve true parallelism on its own—everything runs on a single core.

Instead, JavaScript **manages concurrency**. The event loop handles many asynchronous operations—like network requests, timers, or user interactions—by scheduling their callbacks without blocking the main thread. This concurrency model lets JavaScript stay responsive and efficient, even while waiting for slower operations.

Why This Distinction Matters

- **Performance:** Understanding concurrency helps you write non-blocking code that keeps your UI smooth and responsive.
- **Scalability:** Knowing when parallelism is needed (e.g., for CPU-heavy tasks) leads to using Web Workers or Node.js Worker Threads, which run code on separate threads or cores.

- **Design:** Awareness of concurrency guides proper async patterns and avoids pitfalls like blocking or race conditions.

Summary

- **Parallelism** = multiple tasks running at the same time (requires multiple CPU cores).
- **Concurrency** = multiple tasks making progress overlapping in time but not necessarily simultaneously.
- JavaScript uses concurrency through its event loop but requires workers to achieve parallelism.

Grasping these concepts equips you to design asynchronous code that maximizes performance and scalability, leveraging JavaScript's strengths and overcoming its single-thread limitations.

13.2 Web Workers in Browser JavaScript

Web Workers provide a powerful way to run JavaScript code in the background, separate from the main UI thread. This allows web applications to perform heavy or time-consuming tasks—like complex calculations or data processing—without freezing or slowing down the user interface.

What Are Web Workers?

A **Web Worker** is a background thread that runs independently from the main JavaScript execution environment. Since browsers are mostly single-threaded, running heavy code on the main thread can block the UI, causing janky animations or unresponsive pages. Web Workers solve this by offloading such tasks, keeping the main thread free to handle user interactions smoothly.

Creating a Web Worker

To create a Web Worker, you provide the URL of a JavaScript file that the worker will run:

```
const worker = new Worker('worker.js');
```

The `worker.js` file contains the code that runs in the worker thread.

Communication with Message Passing

Web Workers cannot directly access the DOM or main thread variables. Instead, communication happens via **message passing** using `postMessage` and the `message` event.

- **Main thread sends a message to the worker:**

```
worker.postMessage({ type: 'start', data: 42 });
```

- **Worker listens for messages:**

```
self.addEventListener('message', event => {
  const input = event.data;
  // Process input, then send back a result
  self.postMessage({ result: input.data * 2 });
});
```

- **Main thread receives messages:**

```
worker.addEventListener('message', event => {
  console.log('Result from worker:', event.data.result);
});
```

This structured communication ensures the main thread and worker remain isolated but synchronized.

Terminating a Worker

When the worker is no longer needed, it can be terminated to free up resources:

```
worker.terminate();
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Web Worker Example</title>
  <style>
    body {
      font-family: sans-serif;
      text-align: center;
      padding: 40px;
    }
    button {
      padding: 10px 20px;
      font-size: 16px;
      margin-top: 20px;
    }
    #output {
      margin-top: 30px;
      font-size: 18px;
    }
  </style>
</head>
<body>

  <h1>Web Worker Demo</h1>
  <button id="startBtn">Send Data to Worker</button>
  <div id="output">Result will appear here.</div>

  <script>
    // Define worker logic as a string
    const workerCode = `
      self.addEventListener('message', event => {
        const input = event.data;
        const doubled = input.data * 2;
```

```
      self.postMessage({ result: doubled });
    });
  `;

  // Create a Blob URL from the code
  const blob = new Blob([workerCode], { type: 'application/javascript' });
  const workerUrl = URL.createObjectURL(blob);

  // Create a new worker
  const worker = new Worker(workerUrl);

  // DOM elements
  const startBtn = document.getElementById("startBtn");
  const output = document.getElementById("output");

  // Receive message from worker
  worker.addEventListener('message', event => {
    output.textContent = "Result from worker: " + event.data.result;
  });

  // Send data when button is clicked
  startBtn.addEventListener("click", () => {
    const value = Math.floor(Math.random() * 100); // random input
    output.textContent = "Sending " + value + " to worker...";
    worker.postMessage({ type: 'start', data: value });
  });
</script>

</body>
</html>
```

Limitations of Web Workers

- **No access to the DOM:** Workers can't manipulate the web page directly.
- **Limited APIs:** Some browser APIs are unavailable inside workers.
- **Message serialization:** Data sent between the main thread and workers is copied, not shared, so large data sets may impact performance.

Common Use Cases

- **Heavy Computations:** Running mathematical calculations or image processing without freezing the UI.
- **Data Processing:** Parsing or transforming large datasets asynchronously.
- **Background Tasks:** Handling tasks like real-time data updates or cryptographic operations in the background.

Summary Example

```
// main.js
const worker = new Worker('worker.js');
worker.postMessage(10);
worker.onmessage = event => {
  console.log('Result from worker:', event.data);
};
```

```
// worker.js
self.onmessage = event => {
  const number = event.data;
  const result = number * number; // Heavy calculation
  self.postMessage(result);
};
```

By using Web Workers, your web applications become more responsive, providing smoother experiences even during intensive tasks.

13.3 Worker Threads in Node.js

Node.js traditionally runs JavaScript code in a **single thread**, relying on its event loop for concurrency. However, this model is not ideal for **CPU-intensive tasks** like complex computations or data processing, because such tasks can block the event loop, degrading performance and responsiveness.

To address this, Node.js introduced **Worker Threads**—a powerful way to run JavaScript in parallel threads, enabling true parallelism for CPU-heavy operations.

What Are Worker Threads?

Worker Threads are separate threads within the same Node.js process that can run JavaScript code independently. Unlike child processes, which spawn entirely new Node.js instances with separate memory, worker threads share the same memory space but operate on different threads. This reduces overhead and enables faster communication via **message passing** and **SharedArrayBuffer**.

How Do Worker Threads Differ from Child Processes?

Feature	Worker Threads	Child Processes
Runs within same process	Yes	No, separate OS process
Shared memory	Yes (via SharedArrayBuffer)	No
Communication	Message passing with `postMessage`	IPC via stdio or message events
Overhead	Lower (lighter-weight threads)	Higher (full separate process)
Suitable for	CPU-intensive parallel tasks	Heavy tasks requiring process isolation

Creating and Using Worker Threads

You create a worker thread by importing the `Worker` class from the `worker_threads` module and specifying the JavaScript file it should run.

```
// main.js
const { Worker } = require('worker_threads');

function runWorker(data) {
  return new Promise((resolve, reject) => {
    const worker = new Worker('./worker.js', { workerData: data });

    worker.on('message', resolve);  // Receive result from worker
    worker.on('error', reject);     // Handle errors from worker
    worker.on('exit', code => {
      if (code !== 0)
        reject(new Error(`Worker stopped with exit code ${code}`));
    });
  });
}

runWorker(42)
  .then(result => console.log('Result:', result))
  .catch(err => console.error('Worker error:', err));
```

Worker Thread Script

The worker script receives initial data via `workerData` and can send results back using `parentPort.postMessage()`:

```
// worker.js
const { parentPort, workerData } = require('worker_threads');

function heavyComputation(input) {
  // Simulate CPU-intensive work
  let result = 0;
  for (let i = 0; i < 1e9; i++) {
    result += input * Math.random();
  }
  return result;
}

const result = heavyComputation(workerData);
parentPort.postMessage(result);
```

When to Use Worker Threads

- Offloading **CPU-bound tasks** that block the event loop.
- Performing **parallel processing** while sharing memory efficiently.
- Handling tasks requiring **fine-grained control** over thread lifecycle and communication.

For I/O-bound tasks, Node's async APIs and event loop typically suffice, but worker threads shine when heavy computation threatens to block responsiveness.

13.3.1 Summary

Worker Threads extend Node.js's asynchronous model by introducing **true parallelism**. They allow running multiple JavaScript executions simultaneously within the same process, sharing memory but running on different threads. This makes Node.js suitable for CPU-intensive workloads that were difficult to handle efficiently before.

In the next section, we'll look at a practical example demonstrating how to offload CPU-intensive tasks using Worker Threads to keep your Node.js applications fast and responsive.

13.4 Practical Example: Offloading CPU-Intensive Tasks

Handling CPU-heavy tasks like complex calculations or image processing directly on the main thread can block JavaScript's event loop, making the UI unresponsive in browsers or delaying other operations in Node.js. Offloading such work to **worker threads** (Node.js) or **Web Workers** (browsers) allows the main thread to stay responsive while processing happens in the background.

Scenario: Calculating Large Prime Numbers in Node.js Using Worker Threads

Suppose we want to find prime numbers up to a large limit—a CPU-intensive operation. Running this on the main thread would freeze the app until it finishes.

Step 1: Create the Worker Script

This script runs in the worker thread and performs the calculation:

```
// primeWorker.js
const { parentPort, workerData } = require('worker_threads');

function isPrime(num) {
  if (num < 2) return false;
  for (let i = 2; i <= Math.sqrt(num); i++) {
    if (num % i === 0) return false;
  }
  return true;
}

function findPrimes(limit) {
  const primes = [];
  for (let i = 2; i <= limit; i++) {
    if (isPrime(i)) primes.push(i);
  }
  return primes;
}

try {
  const primes = findPrimes(workerData.limit);
  parentPort.postMessage({ primes });
} catch (error) {
  parentPort.postMessage({ error: error.message });
```

```
}
```

- The worker calculates primes up to `workerData.limit`.
- Results or errors are sent back via `postMessage`.

Step 2: Main Thread Code Using the Worker

This code creates the worker, handles messaging, and demonstrates that the main thread remains free to process other events:

```
// main.js
const { Worker } = require('worker_threads');

function runPrimeWorker(limit) {
  return new Promise((resolve, reject) => {
    const worker = new Worker('./primeWorker.js', { workerData: { limit } });

    worker.on('message', (message) => {
      if (message.error) {
        reject(new Error(message.error));
      } else {
        resolve(message.primes);
      }
    });

    worker.on('error', reject);

    worker.on('exit', (code) => {
      if (code !== 0)
        reject(new Error(`Worker stopped with exit code ${code}`));
    });
  });
}

console.log('Starting prime calculation...');

// Start worker, offloading the heavy task
runPrimeWorker(100000)
  .then((primes) => {
    console.log(`Found ${primes.length} primes.`);
  })
  .catch((err) => {
    console.error('Worker error:', err);
  });

// Main thread remains responsive here
console.log('Main thread is free to do other work while worker runs.');
```

What Happens Here?

- The **main thread** spawns a worker to calculate primes asynchronously.
- The **main thread** logs immediately, showing it is **not blocked**.
- When the worker finishes, it sends the prime list back.
- Errors are properly handled and reported.
- Worker cleanup happens automatically on exit.

Benefits of Offloading Work

- **Improved responsiveness:** UI or main event loop stays fluid.
- **Better scalability:** Heavy computations don't stall other operations.
- **Cleaner architecture:** Background tasks encapsulated in workers.

Browser Alternative: Web Worker

The pattern is similar in browsers with Web Workers. You create a worker script, communicate via `postMessage()`, and listen for messages to keep the UI thread responsive.

13.4.1 Summary

Offloading CPU-intensive tasks to worker threads or Web Workers is a best practice to maintain app responsiveness and scalability. By isolating heavy work in separate threads and communicating via message passing, JavaScript applications stay performant and user-friendly—even under demanding workloads.

Chapter 14.

Advanced Topics in Asynchronous Control Flow

1. Async Queues and Rate Limiting
2. Cancellation with AbortController
3. Retrying and Timeout Strategies
4. Practical Example: Robust API Request with Retries and Timeout

14 Advanced Topics in Asynchronous Control Flow

14.1 Async Queues and Rate Limiting

When working with asynchronous operations—especially network requests, file processing, or database calls—controlling how many tasks run concurrently is crucial. Without limits, running too many async tasks at once can overwhelm resources such as CPU, memory, or network bandwidth. It can also trigger API rate limiting, where external services restrict how frequently you can make requests.

Why Control Concurrency?

- **Resource exhaustion:** Unlimited parallel tasks can consume excessive memory or CPU, causing your app or server to slow down or crash.
- **API throttling:** Many APIs enforce limits on requests per second or minute. Ignoring these can lead to blocked requests or bans.
- **Predictable performance:** Managing concurrency helps maintain smooth, reliable app behavior under load.

Async Queues: Managing Task Execution Order and Concurrency

An **async queue** is a structure that holds tasks waiting to be executed, releasing them in a controlled manner—often in a **First-In-First-Out (FIFO)** order. Queues allow you to specify a **concurrency limit**, meaning how many tasks can run simultaneously.

Common Strategies:

- **FIFO queue with concurrency limit:** Tasks are processed in the order they arrive, but only a fixed number run at the same time.
- **Priority queues:** Tasks with higher priority run before others.
- **Rate limiting:** Limits how many tasks start within a time window (e.g., 5 requests per second).

Example: Simple FIFO Async Queue

Here's a minimal queue implementation managing concurrency:

Full runnable code:

```
class AsyncQueue {
  constructor(concurrency) {
    this.concurrency = concurrency;
    this.running = 0;
    this.queue = [];
  }

  enqueue(task) {
    return new Promise((resolve, reject) => {
      this.queue.push(() => task().then(resolve, reject));
      this.process();
```

```
    });
  }

  process() {
    while (this.running < this.concurrency && this.queue.length) {
      const task = this.queue.shift();
      this.running++;
      task().finally(() => {
        this.running--;
        this.process();
      });
    }
  }
}
//Usage:

const queue = new AsyncQueue(2); // Limit concurrency to 2

function fetchData(id) {
  return new Promise((res) =>
    setTimeout(() => res(`Data for ${id}`), 1000)
  );
}

for (let i = 1; i <= 5; i++) {
  queue.enqueue(() => fetchData(i)).then(console.log);
}
```

This runs only two `fetchData` calls at a time, queuing the rest until slots free up.

Using Libraries: `async.js`

For more robust needs, libraries like `async.js` provide well-tested queue implementations with features like:

- Concurrency control
- Task prioritization
- Rate limiting
- Event callbacks (`drain`, `error`, etc.)

Example with async.js queue:

```
const async = require('async');

const queue = async.queue(async (task) => {
  const result = await fetchData(task.id);
  console.log(result);
}, 2); // concurrency limit 2

queue.push({ id: 1 });
queue.push({ id: 2 });
// Add more tasks as needed
```

14.1.1 Summary

Async queues and rate limiting are essential tools to control concurrency in asynchronous JavaScript programs. They protect your app from overload, comply with external API limits, and ensure smooth performance. Whether implementing your own FIFO queue or using established libraries, mastering these patterns is key for building reliable, scalable asynchronous applications.

14.2 Cancellation with AbortController

In modern JavaScript, handling cancellation of asynchronous operations—especially network requests—is essential to build responsive applications. The **AbortController** API provides a standardized way to signal cancellation, allowing you to abort ongoing operations like `fetch` requests gracefully.

What is AbortController?

`AbortController` is a built-in browser API that creates a controller object with an associated **AbortSignal**. This signal can be passed to APIs that support cancellation (like `fetch`). Calling the controller's `abort()` method triggers the signal, which notifies the operation to cancel.

How to Use AbortController

1. **Create an AbortController instance:**

```
const controller = new AbortController();
const signal = controller.signal;
```

2. **Pass the signal to an async operation (e.g., fetch):**

```
fetch('https://api.example.com/data', { signal })
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => {
    if (error.name === 'AbortError') {
      console.log('Fetch aborted');
    } else {
      console.error('Fetch error:', error);
    }
  });
```

3. **Trigger cancellation when needed:**

```
// Abort the fetch request after 2 seconds
setTimeout(() => controller.abort(), 2000);
```

Handling Aborted Promises

When a fetch is aborted, it rejects with an error of type `AbortError`. You should always handle this specific case to distinguish intentional cancellation from other errors.

Integrating Cancellation in Async Workflows

Using AbortController integrates cleanly into async/await code as well:

```
async function fetchData(url, controller) {
  try {
    const response = await fetch(url, { signal: controller.signal });
    const data = await response.json();
    return data;
  } catch (error) {
    if (error.name === 'AbortError') {
      console.log('Request was aborted');
    } else {
      throw error;
    }
  }
}

const controller = new AbortController();
fetchData('https://api.example.com/data', controller);

// Cancel after 3 seconds if still pending
setTimeout(() => controller.abort(), 3000);
```

Browser Support and Polyfills

Most modern browsers support AbortController natively. However, if you need to support older browsers (like Internet Explorer), polyfills are available, such as abortcontroller-polyfill.

Note that while `fetch` supports AbortController natively, other async APIs may require manual integration or custom wrappers.

14.2.1 Summary

The AbortController API is a powerful tool for cleanly canceling asynchronous tasks like fetch requests. By creating an AbortController and passing its signal, you gain control to abort operations on demand, improve responsiveness, and avoid unnecessary work. Proper error handling for aborted promises ensures smooth cancellation without confusing it for failures. Understanding and applying this API is key for building robust asynchronous JavaScript applications.

14.3 Retrying and Timeout Strategies

In real-world applications, network requests and other asynchronous operations often face unreliable conditions such as temporary outages, slow responses, or server errors. Implementing **retry** and **timeout** strategies is crucial to improve robustness, user experience, and system reliability.

Why Retries and Timeouts Matter

- **Retries** give transient failures a chance to recover automatically without immediate user intervention.
- **Timeouts** prevent your app from waiting indefinitely on slow or unresponsive operations, allowing fallback or user notifications.

Together, they help maintain responsiveness and resilience in asynchronous workflows.

Retry Strategies: Exponential Backoff and Jitter

A naive retry approach might retry immediately or after fixed intervals, but this can overload servers or networks, especially under heavy load or outages. To mitigate this, advanced retry strategies include:

- **Exponential Backoff:** Increase the delay between retries exponentially (e.g., 1s, 2s, 4s, 8s…). This reduces request frequency over time, allowing systems to recover.
- **Jitter:** Add randomness to the delay to prevent many clients retrying simultaneously ("thundering herd" problem).

Example of Exponential Backoff with Jitter:

```
function getBackoffDelay(attempt) {
  const baseDelay = 1000; // 1 second
  const maxDelay = 30000; // 30 seconds max
  let delay = Math.min(maxDelay, baseDelay * 2 ** attempt);
  // Add random jitter between 0 and 500 ms
  delay += Math.floor(Math.random() * 500);
  return delay;
}
```

Implementing Timeouts with Promises

JavaScript doesn't have built-in timeout support for promises, but you can create a **timeout wrapper** that rejects if the operation takes too long:

```
function withTimeout(promise, ms) {
  const timeout = new Promise((_, reject) =>
    setTimeout(() => reject(new Error('Operation timed out')), ms)
  );
  return Promise.race([promise, timeout]);
}
```

You can now run a fetch or other async call with a timeout:

```
withTimeout(fetch(url), 5000)
  .then(response => /* handle response */)
```

```
  .catch(error => console.error(error.message));
```

Combining Retry and Timeout Logic

Retries can be combined with timeouts to create a robust request function:

```
async function fetchWithRetry(url, options = {}, retries = 3) {
  for (let attempt = 0; attempt <= retries; attempt++) {
    try {
      const response = await withTimeout(fetch(url, options), 5000);
      if (!response.ok) throw new Error(`HTTP ${response.status}`);
      return response.json();
    } catch (error) {
      if (attempt === retries) throw error; // Max retries reached
      const delay = getBackoffDelay(attempt);
      console.log(`Retrying in ${delay}ms...`);
      await new Promise(res => setTimeout(res, delay));
    }
  }
}
```

14.3.1 Summary

Retries with exponential backoff and jitter reduce load during failures and improve reliability, while timeouts avoid hanging operations. Combining these patterns results in resilient async workflows that handle unreliable conditions gracefully, giving users better feedback and smoother experiences. Mastering these strategies is key for building robust JavaScript applications.

14.4 Practical Example: Robust API Request with Retries and Timeout

In this section, we'll build a reusable function, `fetchWithRetries`, that performs an API request with built-in support for **retries**, **timeouts**, and **cancellation** using `AbortController`. This example demonstrates clean `async/await` syntax, proper error handling, and flexible configuration.

The Implementation

```
async function fetchWithRetries(url, options = {}) {
  const {
    retries = 3,            // Number of retry attempts
    timeout = 5000,         // Timeout per request in milliseconds
    signal: externalSignal, // Optional external AbortSignal for cancellation
  } = options;
```

```
  // Helper: Wrap fetch with timeout and abort support
  function fetchWithTimeout(signal) {
    return new Promise((resolve, reject) => {
      const timer = setTimeout(() => {
        reject(new Error('Request timed out'));
        controller.abort(); // Abort fetch if timeout triggers
      }, timeout);

      const controller = new AbortController();

      // If external signal exists, listen for abort and forward it
      if (externalSignal) {
        externalSignal.addEventListener('abort', () => {
          clearTimeout(timer);
          controller.abort();
          reject(new Error('Request cancelled externally'));
        });
      }

      fetch(url, { ...options, signal: controller.signal })
        .then(response => {
          clearTimeout(timer);
          if (!response.ok) {
            reject(new Error(`HTTP Error: ${response.status}`));
          } else {
            resolve(response);
          }
        })
        .catch(err => {
          clearTimeout(timer);
          reject(err);
        });
    });
  }

  // Retry loop with exponential backoff
  for (let attempt = 0; attempt <= retries; attempt++) {
    try {
      const response = await fetchWithTimeout(externalSignal);
      const data = await response.json();
      return data; // Success
    } catch (error) {
      if (error.name === 'AbortError') {
        // Propagate abort immediately
        throw new Error('Fetch aborted');
      }
      if (attempt === retries) {
        // All retries exhausted, rethrow error
        throw error;
      }
      // Calculate backoff delay with jitter
      const delay = Math.min(1000 * 2 ** attempt + Math.random() * 500, 10000);
      console.warn(`Attempt ${attempt + 1} failed: ${error.message}. Retrying in ${delay.toFixed(0)} ms
      await new Promise(res => setTimeout(res, delay));
    }
  }
}
```

How to Use It

```
(async () => {
  const controller = new AbortController();

  // Cancel the request after 8 seconds (optional)
  setTimeout(() => controller.abort(), 8000);

  try {
    const data = await fetchWithRetries('https://jsonplaceholder.typicode.com/posts/1', {
      retries: 4,
      timeout: 3000,
      signal: controller.signal,
    });
    console.log('Fetched data:', data);
  } catch (err) {
    console.error('Fetch failed:', err.message);
  }
})();
```

14.4.1 Explanation

- The **`fetchWithRetries`** function wraps a fetch call, adding a per-request timeout and automatic retries with exponential backoff plus jitter.
- It accepts an optional **`AbortSignal`** to allow cancellation from outside the function.
- The fetch is wrapped in a promise that rejects if the timeout expires or if the external signal is aborted.
- Retries only occur for network or HTTP errors; abort errors immediately stop retries.
- The backoff delay doubles on each retry attempt with some randomness to prevent retry storms.
- The usage example demonstrates cancellation after 8 seconds and logs the final success or failure.

14.4.2 Summary

This pattern offers a clean, modular approach to performing **robust API requests** that handle common real-world issues like slow responses, failures, and cancellations. Using `async/await` makes the code readable, and integrating `AbortController` ensures you can cancel requests if needed — an essential feature for modern web applications and responsive user interfaces.