Async/await syntax and error handling

🔄 Async/Await Syntax and Error Handling in JavaScript — In Detail

---

🧠 What is `async/await`?

`async/await` is **syntactic sugar** over Promises, introduced in ES2017, that allows you to write asynchronous code that looks and behaves like synchronous code — making it more readable and easier to manage.

✅ `async` Keyword

- Declares a function as asynchronous.
- Always returns a **Promise** — even if you return a plain value.

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

greet().then(console.log); // "Hello"
```

✅ `await` Keyword

- Can **only be used inside an `async` function.**
- Waits for the Promise to **resolve or reject**, and returns the result.

```
async function fetchData() {
  const response = await fetch("https://api.example.com/data");
  const data = await response.json();
  console.log(data);
}
```

> 📝 If the Promise is rejected, the exception is thrown at the point of await.

✅ Full Example

```
async function getUserData() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/users/1");
    const data = await response.json();
    console.log("User:", data.name);
  } catch (error) {
    console.error("Error fetching user:", error);
  }
}
```

🛠️ Error Handling with `try...catch`

When a promise **rejects**, the `await` throws an error just like a `throw` statement. You handle it using `try...catch`.

```
async function fetchUser() {
  try {
    const response = await fetch("https://wrong-url.com");
    const data = await response.json();
    console.log(data);
  } catch (err) {
    console.error("Failed to fetch:", err.message);
  }
}
```

📊 Comparing Promises vs Async/Await

Aspect	Promise Syntax	Async/Await Syntax
Syntax	`.then().catch()`	`try { await } catch {}`
Readability	Harder in deeply nested chains	Much easier to follow
Error Handling	`catch()`	`try...catch`
Return Value	Wrapped in Promise	Wrapped in Promise

🧪 Returning Multiple Awaits

```
async function multipleRequests() {
  const user = await fetch("/user");
  const posts = await fetch("/posts");
  // Do something with both
}
```

⚠️ **Note:** These will run **sequentially** — to run them in parallel:

```
async function parallelRequests() {
  const [user, posts] = await Promise.all([
    fetch("/user"),
    fetch("/posts"),
  ]);
}
```

🚨 Pitfalls

1. `await` **blocks** within its async function.
2. Use `Promise.all` if you need **parallelism**.
3. Don't use `await` at the top level (except in supported environments like modern browsers or Node with top-level await).

🧠 Best Practices

- Always wrap `await` calls in `try...catch`.
- Use `async/await` for sequential flow logic.
- Use `Promise.all` with `await` for concurrent execution.
- Never `await` inside a loop if not necessary — instead, gather promises and use `Promise.all`.

Let me know if you'd like code examples using real APIs or want to compare performance with chained promises.