ScopeTreasure Courses

JAVASCRIPT COURSES · LESSON 3.21 · CODE SCOPE SERIES

async / await

Promises that read like normal code: async functions, await, side-by-side with .then, try/catch for failures, and the Promise.all lesson.

Introduction

Promises fixed the pyramid, but a long `.then` chain still reads like a machine. `async` and `await` are the same promises wearing readable clothes — no new power, just code that reads top to bottom, like normal code, like a story.

In this lesson you write the same job both ways, catch failures with `try` / `catch`, and learn the one teamwork trick — `Promise.all` — that stops independent jobs from wasting each other's time. *(Roman Urdu: async/await wohi promise hai, bas parhne mein aam code jaisa seedha aur saaf lagta hai.)*

Key points

- **`async` makes a function return a promise — always** — write `async function load() { return "done"; }` and `load()` gives a promise fulfilled with `"done"`, even though you returned a plain value. A throw inside it becomes a rejection.
- **`await` pauses inside the function only** — `const data = await getData();` waits right there until the promise settles, then continues with the real value — no `.then` callback needed.

  (Roman Urdu: await sirf usi function ko rokta hai — baaki program aur page bilkul nahi rukte.)
- **The rest of the program does NOT freeze** — while an `async` function waits at an `await`, other code outside it keeps running, timers keep ticking, the page stays alive. Only that one function is paused.
- **The same job reads like a story** — a `.then` chain nests the steps in callbacks; `async` / `await` writes them as plain lines in order: get the user, get the orders, print the result. Same promises underneath.
- **Failures go in `try` / `catch`** — wrap your `await` lines in `try { ... } catch (error) { ... }`. When the awaited promise rejects, it **throws** into the `catch` block, exactly like a normal error.
- **`await` one-by-one adds the times up** — three independent jobs of 1 second each, awaited in a row, take **3 seconds**. They never needed to wait for each other.
- **`Promise.all` runs them together** — `await Promise.all([jobA(), jobB(), jobC()])` starts all three at once and finishes in the time of the **longest single job** — 1 second in that example, not 3. It fulfills with an array of all values, in order; if any one rejects, the whole `Promise.all` rejects.

  (Roman Urdu: Alag alag await matlab saara waqt jama — Promise.all sab ko saath chalata hai, waqt sirf sab se lambe kaam ka lagta hai.)
- **`await` lives inside `async` functions** — using `await` in a plain function is a syntax error. (One honest exception: at the top level of a JavaScript module file, `await` is allowed.)

Code examples

Example 1 — The same job with a .then chain

```
function getMessage() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Hello from the server");
    }, 1000);
  });
}

getMessage()
  .then((message) => {
    console.log("Message received: " + message);
  })
  .catch((error) => {
    console.log("Failed: " + error.message);
  });

console.log("Request sent, waiting...");
```

What it does: `getMessage` pretends to be a slow server job: it returns a promise that fulfills after 1 second with a message. The `.then` callback receives the message when it arrives; `.catch` stands guard below it. This is the promise style from Lesson 3.20, working perfectly.

What you will see: at once: `Request sent, waiting...`. After 1 second: `Message received: Hello from the server`. Correct — but the real work hides inside a callback, and with four or five steps the chain grows hard to follow.

Example 2 — The same job with async/await, plus try/catch

```
function getMessage() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Hello from the server");
    }, 1000);
  });
}

function getBrokenMessage() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject(new Error("Server is down"));
    }, 1000);
  });
}

async function showMessages() {
  try {
    const message = await getMessage();
    console.log("Message received: " + message);

    const broken = await getBrokenMessage();
    console.log("This line never runs: " + broken);
  } catch (error) {
    console.log("Failed: " + error.message);
  }
  console.log("Function finished");
}

showMessages();
console.log("Request sent, waiting...");
```

What it does: The exact same first job, written with `await` inside an `async` function. The code reads in order, like a story: wait for the message, print it. Then a second job that **rejects** after 1 second — the `await` throws into `catch`, the line after it is skipped, and the function still finishes cleanly.

What you will see: at once: `Request sent, waiting...` — proof the program did not freeze at the `await`. After 1 second: `Message received: Hello from the server`. After 1 more second (2 seconds from the start): `Failed: Server is down` then `Function finished`. The line `This line never runs` never appears.

Example 3 — One-by-one vs Promise.all: the timing lesson

```
function job(name) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(name + " done");
    }, 1000);
  });
}

async function runOneByOne() {
  console.log("One by one: start");
  const a = await job("A");
  console.log(a);
  const b = await job("B");
  console.log(b);
  const c = await job("C");
  console.log(c);
  console.log("One by one: finished");
}

async function runTogether() {
  console.log("Together: start");
  const results = await Promise.all([job("A"), job("B"), job("C")]);
  console.log(results[0]);
  console.log(results[1]);
  console.log(results[2]);
  console.log("Together: finished");
}

async function main() {
  await runOneByOne();
  await runTogether();
}

main();
```

What it does: Three independent jobs, each taking 1 second. `runOneByOne` awaits them in a row, so each waits for the last — the times add up. `runTogether` starts all three at the same moment with `Promise.all`, so the wait is only as long as the slowest single job.

What you will see: at once: `One by one: start`. Then `A done` after 1 second, `B done` after 2 seconds, `C done` after 3 seconds, then `One by one: finished` — 3 seconds total. Next, `Together: start`, and just 1 second later (4 seconds from the very beginning) all three lines arrive together: `A done`, `B done`, `C done`, then `Together: finished`. Same three jobs: 3 seconds in a queue, 1 second as a team.

Quick recap

- `async` makes a function always return a promise; `await` pauses only inside that function until the promise settles — the program never freezes.
- Wrap `await` lines in `try` / `catch`: a rejection throws into `catch`, like a normal error.
- Independent jobs awaited in a row take the **sum** of their times; `Promise.all` starts them together and takes only the **longest** single time.

Common mistakes

1. **Forgetting `await`** — `const data = getData();` without `await` stores the **promise object**, not the value, and printing it shows `Promise { <pending> }`. The fix: `const data = await getData();` inside an `async` function.
2. **Using `await` in a plain function** — the file stops with a syntax error before anything runs. The fix: mark the function `async` — `await` only lives inside `async` functions (or at the top level of a module).
3. **Awaiting independent jobs one by one** — three 1-second jobs that do not depend on each other take 3 slow seconds in a row. The fix: start them together with `await Promise.all([jobA(), jobB(), jobC()])` and pay only the longest single time.

Practice

1. Write an `async function greet()` that awaits a promise which resolves with your name after 1 second, then logs `Welcome,` plus the name. Add a `console.log` after the `greet()` call and confirm it prints **before** the welcome line.
2. Make three jobs that each resolve after 1 second with the numbers 10, 20 and 30. First add their results with three separate `await` lines and time it (about 3 seconds). Then rewrite it with `Promise.all` and confirm the same total — 60 — arrives in about 1 second.

Practice quiz

1. What does an async function always return?

- A plain value, never a promise
- A promise, even when you return a plain value
- undefined, always
- A string

Answer: A promise, even when you return a plain value

2. While an async function waits at an await, what happens to the rest of the program?

- Everything freezes until the await finishes
- Only that one function is paused, other code keeps running and the page stays alive

- The browser tab closes
- All other functions are cancelled

Answer: Only that one function is paused, other code keeps running and the page stays alive

3. Three independent jobs take 1 second each. Awaited one by one they take 3 seconds. How long with await Promise.all([jobA(), jobB(), jobC()])?

- 3 seconds, the same
- 1 second, the time of the longest single job, because they start together
- 0 seconds
- 9 seconds

Answer: 1 second, the time of the longest single job, because they start together

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/js-async-await · More free tools and notes at https://scopetreasure.com