ScopeTreasure Courses

JAVASCRIPT COURSES · LESSON 3.20 · CODE SCOPE SERIES

Callbacks → Promises

Waiting without freezing: callbacks and their pyramid of doom, the Promise and its three states, .then/.catch and chaining that reads clean.

Introduction

Some jobs take time — a server answering, a file loading, a timer finishing. Your program cannot freeze and wait; the page would die. So JavaScript uses a simple deal: "when the job is done, run **this** function." That function is a **callback**, and its modern, cleaner form is the **Promise**.

This lesson goes in order: callbacks first, the mess they create when they nest, then promises — the receipt that fixes the mess. *(Roman Urdu: Callback matlab kaam khatam ho to ye function chalana — Promise usi khayal ki saaf aur nayi shakal hai.)*

Key points

- **A callback is a function passed as an argument** — you hand it over now, and the other code calls it **later**, when its job is done. Think of ordering chai: you do not stand at the counter; the shop calls you back when the chai is ready.
- **Callbacks keep the page alive** — while the slow job runs in the background, the rest of the program keeps working. Nothing freezes.

  (Roman Urdu: Callback ki wajah se page rukta nahi — slow kaam peeche chalta rehta hai, baaki code aage barhta hai.)
- **Nested callbacks become the pyramid of doom** — a callback inside a callback inside a callback, each step waiting for the last. The code slides further right every line, hard to read and harder to fix.
- **A Promise is a receipt for a future value** — you get the promise object at once, and the real value arrives later. `new Promise((resolve, reject) => { ... })` — call `resolve(value)` on success, `reject(reason)` on failure.
- **A promise has exactly three states** — **pending** (still waiting), **fulfilled** (done — the value is in hand), **rejected** (failed — the reason is in hand). A promise settles once: pending becomes fulfilled **or** rejected, never both, never back.
- **`.then` runs on success, `.catch` runs on failure** — `promise.then(value => ...)` receives the fulfilled value. `promise.catch(error => ...)` catches **any** failure in the whole chain above it.

  (Roman Urdu: .then kamyabi par chalta hai, .catch kisi bhi nakami ko pakar leta hai.)
- **Chaining makes it flat** — each `.then` can **return** a value or a new promise, and the next `.then` receives it. The same work as the pyramid, written top to bottom in a straight line.
- **Errors jump straight to `.catch`** — if any step in a chain fails, the `.then` steps in between are **skipped**, and control lands in the nearest `.catch`.

Code examples

Example 1 — A callback: the chai order

```
function orderChai(callback) {
  console.log("Order placed");
  setTimeout(() => {
    console.log("Chai is ready");
    callback();
  }, 2000);
}

orderChai(() => {
  console.log("Customer takes the chai");
});

console.log("Customer sits and waits");
```

What it does: `orderChai` takes a callback — the function to run when the chai is ready. A 2-second `setTimeout` plays the role of the shop making chai. When the timer finishes, the shop logs that the chai is ready and **calls the callback**.

What you will see: at once: `Order placed` then `Customer sits and waits`. After 2 seconds: `Chai is ready` then `Customer takes the chai`. The customer never froze — the wait happened in the background, and the callback fired at the right moment.

Example 2 — The pyramid of doom

```
function getUser(callback) {
  setTimeout(() => {
    console.log("Step 1: user found");
    callback({ name: "Ali" });
  }, 500);
}

function getOrders(user, callback) {
  setTimeout(() => {
    console.log("Step 2: orders for " + user.name);
    callback(["Book", "Pen"]);
  }, 500);
}

function getPayment(orders, callback) {
  setTimeout(() => {
    console.log("Step 3: payment for " + orders.length + " items");
    callback("Paid");
  }, 500);
}

getUser((user) => {
  getOrders(user, (orders) => {
    getPayment(orders, (result) => {
      console.log("Final: " + result);
    });
  });
});
```

What it does: It runs three slow steps in order — find the user, get his orders, take payment. Each step can only start inside the previous step's callback, so the calls nest deeper and deeper, sliding right like a pyramid lying on its side.

What you will see: after about half a second: `Step 1: user found`. After about 1 second: `Step 2: orders for Ali`. After about 1.5 seconds: `Step 3: payment for 2 items`. Then: `Final: Paid`. The result is correct — but imagine ten steps, or an error to handle inside each layer. That pain is real, and it has a name: the pyramid of doom.

Example 3 — The same work as a promise chain

```
function getUser() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Step 1: user found");
      resolve({ name: "Ali" });
    }, 500);
  });
}

function getOrders(user) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Step 2: orders for " + user.name);
      resolve(["Book", "Pen"]);
    }, 500);
  });
}

function getPayment(orders) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Step 3: payment for " + orders.length + " items");
      resolve("Paid");
    }, 500);
  });
}

getUser()
  .then((user) => getOrders(user))
  .then((orders) => getPayment(orders))
  .then((result) => {
    console.log("Final: " + result);
  })
  .catch((error) => {
    console.log("Something failed: " + error.message);
  });
```

What it does: Each function now returns a **Promise** instead of taking a callback. The chain reads flat, top to bottom: get the user, **then** get orders, **then** get payment, **then** log the result. Each `.then` returns the next promise, and its value is handed to the next step. One `.catch` at the end guards the whole chain.

What you will see: exactly the same output, in the same order, at the same times as Example 2: `Step 1: user found` after about half a second, `Step 2: orders for Ali` after about 1 second, `Step 3: payment for`

`2 items` after about 1.5 seconds, then `Final: Paid`. Same work — but the chain stays flat, and adding a step means adding one `.then` line, not another layer of nesting.

Quick recap

- A **callback** is "when the job is done, run this" — simple for one step, a **pyramid of doom** when steps nest.
- A **Promise** is a receipt for a future value with three states: **pending, fulfilled, rejected**.
- `.then` handles success and chains flat, each step handing its value to the next; `.catch` catches any failure in the chain, and errors skip straight to it.

Common mistakes

1. **Expecting the callback to run at once** — a callback tied to a timer or a server runs **later**, so lines after the call see no result yet. The fix: put the work that needs the result **inside** the callback (or the `.then`), never below the call.
2. **Forgetting to return the promise inside** `.then` — writing `getUser().then((user) => { getOrders(user); })` breaks the chain; the next step gets `undefined` and errors lose their road to `.catch`. The fix: `return getOrders(user)` so the chain can wait for it.
3. **Handling errors inside every single step** — five small error handlers scattered through a pyramid is how failures get lost or doubled. The fix: let errors travel down the chain to **one** `.catch` at the end.

Practice

1. Write a `makeTea(callback)` function that logs `Tea is on the stove`, waits 2 seconds with `setTimeout`, logs `Tea is ready`, then calls the callback, which logs `Now we drink`. Run it and check the order of all four lines.
2. Take the three step functions from Example 2 and rewrite them as promise-returning functions like Example 3. Then make `getOrders` call `reject(new Error("No orders found"))` instead of `resolve`, run the chain, and watch the payment step get skipped while `.catch` prints the error.

Practice quiz

1. What is a callback?

- A function that runs at once, before anything else
- A function passed as an argument, called later when the other code's job is done
- A function that calls itself
- A promise with a different name

Answer: A function passed as an argument, called later when the other code's job is done

2. What are the three states of a promise?

- start, middle, end
- pending, fulfilled, rejected, and it settles once, never both and never back
- waiting, working, done, and it can go back
- yes, no, maybe

Answer: pending, fulfilled, rejected, and it settles once, never both and never back

3. In a promise chain, what do .then and .catch do?

- .then runs on failure, .catch runs on success
- .then runs on success and chains flat, .catch catches any failure in the chain above it
- Both run on every step
- .catch only catches errors in the last step

Answer: .then runs on success and chains flat, .catch catches any failure in the chain above it

---

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-callbacks-promises · More free tools and notes at https://scopetreasure.com