LECTURE 15

Asynchronous JavaScript: Promises and Fetch

How JavaScript waits on things that take time — without ever freezing the page.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. Synchronous vs. asynchronous execution, the event loop, task queue, and "callback hell"
2. Promises: their three states, .then/.catch/.finally, chaining, and Promise.all
3. async/await, and error handling with try...catch
4. Making real network requests with the Fetch API: GET/POST, headers, response parsing, and CORS

One Line at a Time

```
console.log("1");
console.log("2");
console.log("3");
// Always prints: 1, 2, 3 -- in that exact order, with no gaps
```

Each instruction must finish before the next one starts. This is how almost all the code you've written so far behaves.

Some Things Happen "Later"

```
console.log("1");
setTimeout(() => console.log("2"), 1000); // scheduled, doesn't block
console.log("3");
// Prints: 1, 3, 2 -- "2" appears about a second later!
```

JavaScript starts the operation, moves on immediately, and comes back to handle the result once it's ready.

The Exact Code, Live

```
console.log("1");
setTimeout(() =>
  console.log("2"), 150);
console.log("3");
```

```
> console.log("1")
> console.log("3")
> console.log("2") -- runs LAST, after the delay
```

WHY THIS MATTERS

JavaScript Is Single-Threaded

1. The browser can only do one thing at a time on that one thread
2. A blocking operation would freeze buttons, scrolling, and animations until it finished
3. Timers, network requests, and file reads are handed off so the thread stays free to keep the page responsive

How setTimeout(fn, 1000) Actually Runs

The event loop constantly checks: is the call stack empty? If yes, it takes the next item from the queue and runs it.

Microtasks Run Before Macrotasks

NOTE

Promise callbacks (.then, async/await) go into a special microtask queue that the event loop always empties completely before it looks at the regular task queue (where setTimeout callbacks live). This is why Promise.resolve().then(...) scheduled after setTimeout(..., 0) usually still runs first.

Callback Hell

1. Before Promises existed, async code was handled with callbacks -- functions passed in to be called later
2. When one async step depends on the result of the previous one, callbacks nest deeper and deeper
3. Nicknamed "callback hell" or the "pyramid of doom" -- hard to read, hard to maintain

BEFORE PROMISES

The Pyramid of Doom

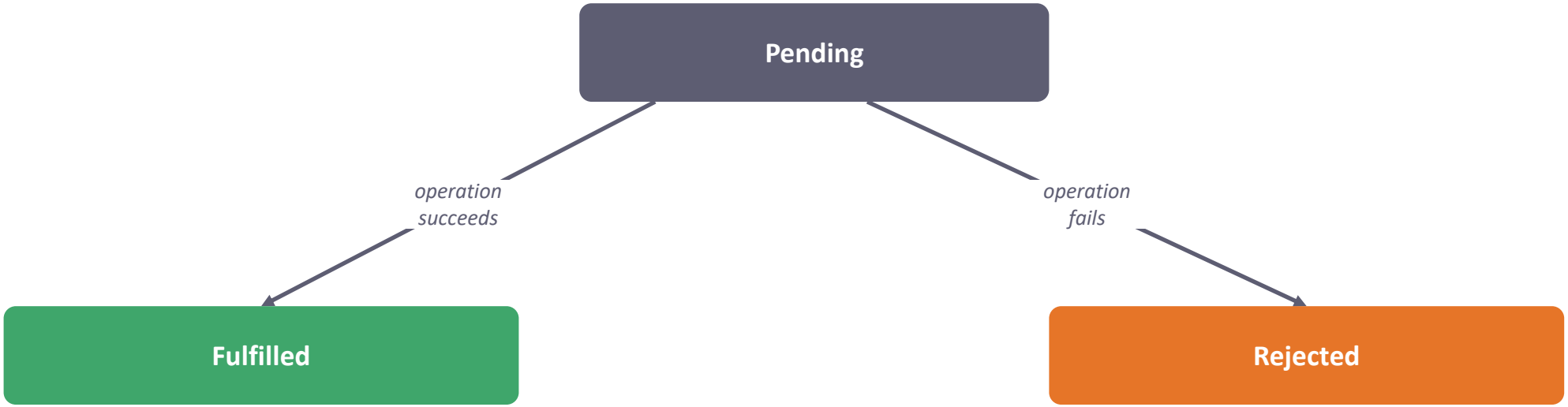
```
getUser(userId, function (user) {
  getOrders(user.id, function (orders) {
    getOrderDetails(orders[0].id, function (details) {
      console.log(details);
      // ...and it keeps growing to the right with every extra step
    }, handleError);
  }, handleError);
}, handleError);
```

What Is a Promise?

1. An object representing a value that isn't available yet, but will be at some point -- either successfully, or with an error

2. Think of it as a receipt for a value that's still being prepared

The Three States



A promise is settled once it becomes fulfilled or rejected, and it can never change state again after that.

Creating and Using a Promise

```
function delay(ms) {
  return new Promise((resolve, reject) => {
    if (ms < 0) { reject(new Error("Delay cannot be negative")); return; }
    setTimeout(() => resolve(`Waited ${ms}ms`), ms);
  });
}

delay(1000)
  .then(result => console.log(result))    // fulfilled
  .catch(error => console.error(error))   // rejected
  .finally(() => console.log("Done, either way")); // always runs
```

delay().then/.catch/.finally, Live

```
delay(150)
  .then(result =>
    log(result))
  .catch(error =>
    log("ERROR: " +
      error.message))
  .finally(() =>
    log("Done, either way"));
```

```
> Waited 150ms
> Done, either way
```

Chaining Promises

```
getUser(userId)
  .then(user => getOrders(user.id))              // after getUser resolves
  .then(orders => getOrderDetails(orders[0].id)) // after getOrders resolves
  .then(details => console.log(details))         // after getOrderDetails resolves
  .catch(error => console.error("Something failed:", error)); // catches ANY step
```

Each .then() returns a NEW promise, which is what makes chaining possible -- one .catch() at the end catches an error from any step.

Promise.all

```
const promise1 = delay(1000).then(() => "First");
const promise2 = delay(500).then(() => "Second");
const promise3 = delay(1500).then(() => "Third");

Promise.all([promise1, promise2, promise3]).then(results => {
  console.log(results); // ["First","Second","Third"] -- after ~1500ms, not 3000ms
});
```

Fulfills only once ALL promises have fulfilled, or rejects immediately if any ONE of them rejects -- ideal for independent data loaded in parallel.

Promise.all, Live

```
const promise1 =
  delay(100).then(() => "First");
const promise2 =
  delay(50).then(() => "Second");
const promise3 =
  delay(150).then(() => "Third");

Promise.all(
  [promise1, promise2, promise3]
).then(results => { ... });
```

```
> Promise.all([promise1, promise2, promise3]).then(...)
results = ["First","Second","Third"]
elapsed ~159ms (not 300ms -- they ran in PARALLEL)
```

Two Rules

1. async before a function makes it always return a Promise, and allows await inside it

2. await pauses execution of THAT function (not the whole program) until the promise settles, then unwraps the resolved value -- or throws if it rejected

Reads Like Synchronous Code

```
async function loadOrderDetails(userId) {
  const user = await getUser(userId);       // "pause" until the promise settles
  const orders = await getOrders(user.id);
  const details = await getOrderDetails(orders[0].id);
  return details;
}
```

Error Handling with try...catch

```
async function loadOrderDetails(userId) {
  try {
    const user = await getUser(userId);
    const orders = await getOrders(user.id);
    const details = await getOrderDetails(orders[0].id);
    return details;
  } catch (error) {
    console.error("Failed to load order details:", error.message);
    throw error; // re-throw if the caller also needs to know
  }
}
```

try...catch, Live (getUser rejects)

```
async function loadOrderDetails(id) {
  try {
    const user =
      await getUser(id);
    return user;
  } catch (error) {
    log("caught: " +
      error.message);
    throw error;
  }
}
```

```
> loadOrderDetails(999)
caught in loadOrderDetails: No user with id 999
caller's .catch(): No user with id 999
```

async/await vs. .then Chains

TIP

They do the same job -- async/await is just easier to read, especially with multiple sequential steps and conditional logic. You will see both styles in real codebases, but prefer async/await for new code you write.

What Is AJAX?

1. The general technique of a web page requesting data from a server in the background, without a full page reload
2. The name is historical (Asynchronous JavaScript and XML) -- today it almost always means JSON, not XML
3. The modern, built-in tool for doing this is the Fetch API

A GET Request

```
async function loadStudents() {
  const response = await fetch("https://api.example.com/students");
  const students = await response.json(); // parses the body as JSON
  console.log(students);
}
```

fetch(url) resolves with a Response once headers arrive -- it resolves even for 404s; it only REJECTS on a genuine network failure.

Always Check response.ok

```
async function loadStudents() {
  const response = await fetch("https://api.example.com/students");
  if (!response.ok) {
    throw new Error(`Request failed with status ${response.status}`);
  }
  const students = await response.json();
  return students;
}
```

The Exact GET Code, Live

```
const response =
  await fetch(url);
response.ok;
response.status;

const students =
  await response.json();
```

```
> const response = await
fetch("https://api.example.com/students");
> response.ok = true    response.status = 200
> const students = await response.json();
[
  {
    "id": 1,
    "name": "Hina",
    "major": "Computer Science"
  },
  {
    "id": 2,
    "name": "Ali",
    "major": "Software Engineering"
  }
]
```

A POST Request with Headers

```
async function createStudent(newStudent) {
  const response = await fetch("https://api.example.com/students", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(newStudent), // JS object -> JSON text
  });
  if (!response.ok) throw new Error(`Failed to create student: ${response.status}`);
  return response.json();
}
```

Content-Type: application/json tells the server "the body I'm sending is JSON text," so it knows how to parse it.

Parsing Different Response Types

Method	Parses the body as
response.json()	JSON -- the most common case
response.text()	Plain text
response.blob()	Raw binary data (e.g. an image)

You can only read a response body ONCE -- call one parsing method per response.

Putting It Together with Promise.all

```
async function fetchAllPages() {
  try {
    const [usersRes, postsRes] = await Promise.all([
      fetch("https://api.example.com/users"),
      fetch("https://api.example.com/posts"),
    ]);
    const users = await usersRes.json();
    const posts = await postsRes.json();
    console.log(users, posts);
  } catch (error) {
    console.error("Network error:", error.message);
  }
}
```

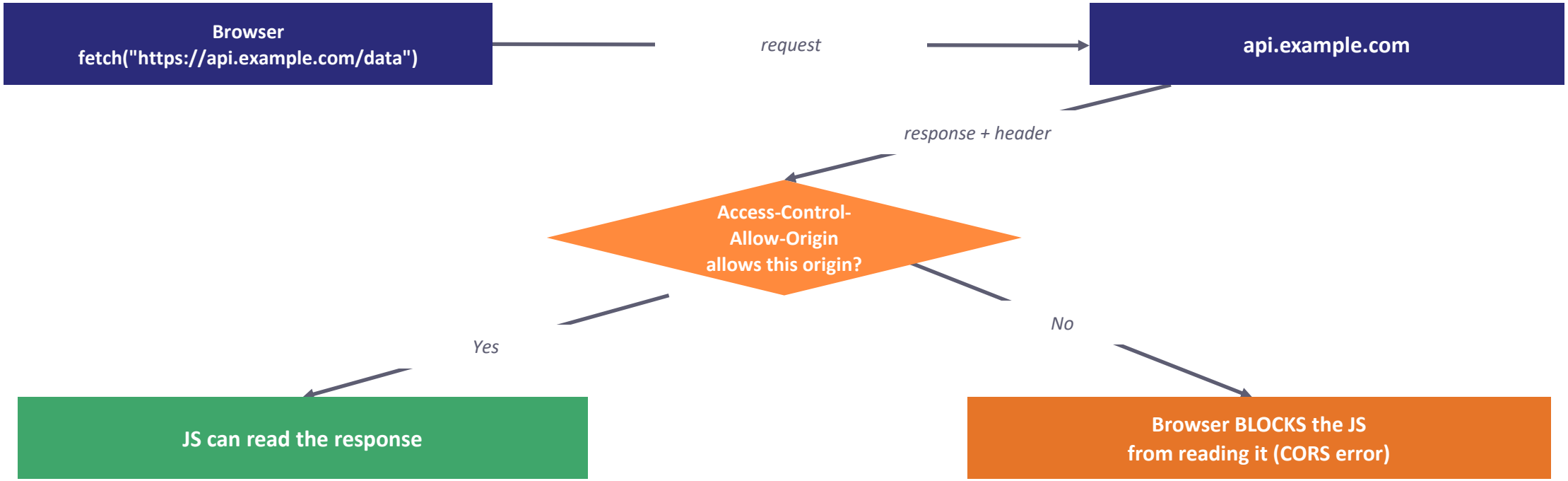
Cross-Origin Resource Sharing

1. A browser security rule restricting JS on one origin (protocol + domain + port) from freely reading responses from a different origin

2. The server decides who is allowed, via an Access-Control-Allow-Origin response header

3. CORS is enforced by the BROWSER, not your JavaScript -- you cannot "fix" a CORS error from the frontend

How the Browser Checks CORS



The network request may technically succeed either way -- CORS only controls whether your JavaScript is allowed to read the response.

CORS Is Not Something You Can Bypass

WARNING

If you see a CORS error, the fix belongs on the server (adding the right Access-Control-Allow-Origin header), not in your fetch call. You'll work with server configuration, including CORS, starting in Unit 5.

KEY TAKEAWAYS

Lecture 15 in Six Points

1. Synchronous code blocks the single JS thread; asynchronous code lets long operations happen in the background without freezing the page.
2. The event loop moves finished tasks from the queue onto the call stack whenever it's empty -- this is how setTimeout and Promise callbacks run.
3. A Promise moves through pending, fulfilled, and rejected; chaining .then() calls replaces nested callback hell.
4. Promise.all runs multiple promises in parallel and waits for all of them, or fails fast if any one rejects.
5. async/await is Promise-based syntax that reads like synchronous code; errors are handled with ordinary try...catch.
6. fetch() rejects only on true network failure, not on HTTP error codes -- always check response.ok; CORS is a browser rule enforced via response headers, fixable only on the server.