JavaScript Promises — a Clear, Detailed Guide

What is an Asynchronous (Async) Operation?

An async operation is work that doesn't finish immediately and would otherwise block the single JavaScript thread if it ran synchronously. Instead, it's started now and completed later, with the result delivered when it's ready.

Examples of async operations: network requests (fetch, AJAX), timers (setTimeout, setInterval), file/database I/O in Node.js, and browser APIs like crypto.subtle or indexedDB.

What is a Promise?

A Promise is an object that represents the eventual result (success or failure) of an async operation.

States of a Promise: Pending (initial state), Fulfilled (completed successfully), and Rejected (failed with an error). Once fulfilled or rejected, the promise is settled and cannot change.

Creating a Promise

```
const delay = (ms) => new Promise((resolve, reject) => {
  if (ms < 0) reject(new Error("ms must be >= 0"));
  setTimeout(() => resolve(`Waited ${ms}ms`), ms);
});
```

Consuming a Promise: then, catch, finally

then(onFulfilled, onRejected?) runs on success and returns a new promise. catch(onRejected) runs on failure and also returns a new promise. finally(onFinally) runs always after settle, useful for cleanup.

```
delay(500)
  .then(msg => console.log(msg))
  .catch(err => console.error("Something failed:", err))
  .finally(() => console.log("cleanup done"));
```

Chaining & Error Propagation

```
delay(200)
  .then(v => { throw new Error("boom"); })
  .catch(err => console.log("handled:", err.message))
  .then(() => console.log("continues after catch"));
```

async / await

```
async function loadUser(id) {
  try {
    const res = await fetch(`/api/users/${id}`);
    if (!res.ok) throw new Error(`HTTP ${res.status}`);
    const data = await res.json();
    return data;
  } catch (err) {
    console.error("loadUser failed:", err);
```

```
    throw err;
  } finally {
    console.log("cleanup (e.g., hide spinner)");
  }
}
```

Promise Combinators

Promise.all waits for all promises, Promise.allSettled resolves regardless of outcome, Promise.race settles as soon as the first one settles, Promise.any resolves with the first successful one.

```
const a = delay(300).then(() => "A");
const b = delay(200).then(() => "B");

const both = await Promise.all([a, b]);     // ["A", "B"]
const first = await Promise.race([a, b]);   // "B"
```

Promisifying Callback APIs

```
import { readFile } from "fs";

const readFileP = (path, encoding) =>
  new Promise((resolve, reject) => {
    readFile(path, encoding, (err, data) => {
      if (err) reject(err);
      else resolve(data);
    });
  });
```

Common Pitfalls & Tips

1. Forgetting to return inside then. 2. Swallowing errors. 3. Misusing finally for changing values. 4. Creating unnecessary new Promise. 5. Unhandled rejections.

End-to-End Example

```
function getJson(url) {
  return fetch(url).then(res => {
    if (!res.ok) throw new Error(`HTTP ${res.status}`);
    return res.json();
  });
}

getJson("/api/products")
  .then(items => items.filter(p => p.inStock))
  .then(inStock => console.log("Available:", inStock.length))
  .catch(err => console.error("Failed:", err))
  .finally(() => console.log("Done"));
```