Module 8 of 16

Asynchronous JavaScript

The event loop and its two queues; error-first callbacks; promises and the chaining rules; async/await semantics, sequential versus concurrent, return await; the combinators, concurrency pools, timeouts and AbortController; timers, debounce and throttle.

JavaScript never waits. One thread runs your code to completion, and everything that takes time — a timer, a file, a request — hands its result back later through the event loop. Every asynchronous abstraction the language has grown since then is a way of *writing down* what happens later: a callback, a promise, an `await`. Understanding the loop is what makes the ordering predictable; understanding promises is what makes composition possible; `async` / `await` is what makes it readable.

The module builds up in that order. The event-loop lesson gives the two-queue model that answers every "why did this print first" question and the error-first callback convention Node was built on. The promise lesson pins down the state machine and the chaining rules — return the inner promise, end in a catch, never construct a promise around a promise. `async` / `await` follows with the rules people learn by getting wrong: awaits in a loop are sequential, `forEach(async)` waits for nothing, `return await` matters inside `try`, state changes across an `await`. The combinators lesson covers `all` / `allSettled` / `race` / `any`, a concurrency pool, timeouts, cancellation with `AbortController`, retry with backoff and `for await`. Timers close it: minimum delays, recursive `setTimeout` over `setInterval`, the "soon" family, `timers/promises`, debounce and throttle.

Every exercise runs on the real event loop with real timers: predict the print order, write `series` / `parallel` with callbacks, build a promise chain and a `promisify`, prove that `forEach(async)` does not wait, drive the four combinators, write a pool and a timeout, debounce and throttle, and finish with a dependency-aware job scheduler, a fetch pipeline with retry and timeout, and an async queue.

CONTENTS

1	Callbacks and the event loop — one thread, two queues	14 min
2	Promises — a value that is not here yet	14 min
3	async/await — promises with the syntax of ordinary code	14 min

4	Combinators, concurrency limits, timeouts and cancellation	14 min
5	Timers and scheduling — setTimeout, setInterval, nextTick, and debounce/throttle	12 min
✓	Checkpoint — Asynchronous JavaScript	25 min

Exercises in this module compile on Node 16. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

Callbacks and the event loop — one thread, two queues

JavaScript runs your code on **one thread**. It never waits: when something takes time — a timer, a file read, a network request — the runtime hands the work to the system, keeps executing, and later runs a function you left for it. The machinery that decides *when* is the **event loop**, and its two queues — one for tasks, one for microtasks — explain every ordering puzzle: why `setTimeout(fn, 0)` runs after a `Promise.then`, why a long loop freezes the UI, why an `await` in a loop is sequential. Module 1 previewed this; this lesson is the real model, plus the callback style that Node was built on and that every later abstraction (promises, `async`) is sugar over.

The call stack and the loop

Synchronous code runs to completion on the **call stack**. When the stack is empty the event loop picks the next **task** (a timer that expired, a completed I/O operation, an incoming event) and runs its callback — again to completion. Nothing interrupts running code: a callback cannot start until the current one finishes. That is why a `while (Date.now() < deadline) {}` busy-wait blocks *everything*, including timers that already expired, and why the single-threaded model has no data races: only one piece of JavaScript is ever running.

```
console.log("A");
setTimeout(() ⇒ console.log("C"), 0);   // schedule a task for "as soon as p
console.log("B");
// A B C — C waits for the stack to empty, whatever the delay
```

Microtasks versus tasks

There are two queues, and the loop drains them differently:

Tasks (macrotasks): timers (`setTimeout`, `setInterval`), I/O callbacks, `setImmediate`. One task per loop iteration.

Microtasks: promise reactions (`.then`, `await` continuations), `queueMicrotask`, and in Node, `process.nextTick` (its own, even higher-priority queue). **All** pending microtasks run

after the current task, before the next task — including microtasks queued by other microtasks.

```
setTimeout(() ⇒ console.log("timeout"), 0);
Promise.resolve().then(() ⇒ console.log("promise"));
process.nextTick(() ⇒ console.log("tick"));
console.log("sync");
// sync, tick, promise, timeout
```

The rule: **synchronous code → nextTick queue → promise microtasks → the next task**. A microtask that keeps queueing microtasks starves the loop forever (no timer or I/O ever runs); a task that keeps scheduling tasks lets other tasks interleave. That is why microtasks are for "right after this, before anything else" and tasks are for "later, let other things happen".

Node's loop phases, briefly

Node's loop runs in phases: **timers** (expired `setTimeout` / `setInterval`), **pending callbacks**, **poll** (I/O — where the process idles when there is nothing to do), **check** (`setImmediate`), **close callbacks**. Between every callback the microtask queues drain. Consequences worth knowing: `setImmediate` runs after I/O in the same iteration and before the next timers phase — inside an I/O callback it always beats `setTimeout(fn, 0)`; from the main module their order is unspecified (depends on how long startup took). `process.nextTick` runs before *any* of them and before promise microtasks; the Node docs recommend `setImmediate` for "yield to the loop" and `queueMicrotask` / `nextTick` only for "before anything else".

Timers are minimums

`setTimeout(fn, 100)` means "not before 100 ms" — the callback runs at the first timers phase after the delay, once the stack is free. Under load it is late; two timers with the same delay run in registration order; delays below 1 ms are clamped to 1. Never rely on timers for exact timing or for ordering against I/O; rely on them for "at least this long". (Lesson 5 goes further.)

The callback style

Before promises, asynchronous APIs took a function to call when done. Node fixed a convention — **error-first callbacks**: `(err, result) ⇒ …`, `err` is `null` on success:

```
const fs = require("fs");
fs.readFile("config.json", "utf8", (err, text) ⇒ {
  if (err) return console.error("read failed:", err.message);   // handle, the
  const cfg = JSON.parse(text);
  fs.readFile(cfg.next, "utf8", (err2, more) ⇒ {                // the next st
    if (err2) return console.error(err2.message);
    …
  });
});
```

Rules: check `err` first and return; call the callback **exactly once**; call it asynchronously even when you have the answer already (a callback that sometimes runs synchronously — "releasing Zalgo" — makes callers' code run in two different orders); errors thrown *inside* a callback cannot be caught by the code that scheduled it (the scheduling stack is gone), so they must be passed to the callback, not thrown.

Why callbacks were abandoned

Sequential steps nest ("callback hell"), error handling repeats at every level, running things in parallel and waiting for all of them needs hand-written counters, and a thrown error inside a callback crashes the process because no `try` is on the stack. Promises solve the composition problem (next lesson) and `async` / `await` solves the syntax problem (lesson 3). Callbacks remain in event emitters (`on("data", …)`), older APIs and the innermost layer of libraries — `util.promisify` converts an error-first function into a promise-returning one in one line, which is how you meet them today.

Blocking, and why it matters

Because there is one thread, every millisecond of synchronous work delays every pending callback: a 200 ms JSON parse of a large payload freezes the browser's rendering or delays every other request in a Node server. Keep synchronous chunks short; move CPU-heavy work to a worker (`worker_threads` , Web Workers) or split it across tasks (`setImmediate` between chunks) so the loop can breathe. "Asynchronous" does not mean "parallel" — it means "not blocking the thread while waiting".

Common mistakes

Expecting `setTimeout(fn, 0)` to run before a pending `.then`.

Busy-waiting or long synchronous loops that starve timers and I/O.

Recursive `process.nextTick`/microtask scheduling that never yields.

Callbacks called twice, called synchronously sometimes, or not called on an error path.

Throwing inside a callback and expecting the caller's `try` / `catch` to see it.

Forgetting `return` after handling `err`.

Interview angle

"Explain the event loop." One thread; the loop runs a task to completion, drains all microtasks, then takes the next task; timers and I/O are tasks, promise reactions are microtasks.

"Order of `setTimeout(0)`, `Promise.then`, `process.nextTick`, sync?" sync → nextTick → then → timeout.

"Microtask versus macrotask?" Microtasks run immediately after the current task, all of them; tasks run one per iteration and let others interleave.

"Error-first callback convention?" `(err, result)`, check and return on `err`, call exactly once, asynchronously.

"Why does a long loop freeze the page?" The single thread cannot run rendering or event callbacks until the loop's current task completes.

Key takeaways

One thread; a task runs to completion; then every microtask; then the next task.

Microtasks: promise reactions, `queueMicrotask`, (Node) `process.nextTick` first; tasks: timers, I/O, `setImmediate`.

Timers are minimum delays; same-delay timers keep registration order; `setImmediate` beats `setTimeout(0)` after I/O.

Error-first callbacks: check `err` and return, call once, call asynchronously; errors inside callbacks cannot be caught by the scheduler.

Blocking the thread delays everything; split or offload CPU-heavy work.

EXERCISE 1

Predict the print order

Each input line schedules a print of its label: `sync <label>` prints immediately, `timeout <label> <ms>` uses `setTimeout`, `micro <label>` uses `Promise.resolve().then`, `tick <label>` uses `process.nextTick`. Schedule them all in input order; the output is the order in which the labels actually print, one per line.

Example: `sync A`, `timeout B 0`, `micro C`, `tick D`, `timeout E 10`, `sync F` →

```
A
F
D
C
B
E
```

Synchronous code first, then the nextTick queue, then promise microtasks, then timers in order of expiry.

INPUT 1

```
sync A
timeout B 0
micro C
tick D
timeout E 10
sync F
```

OUTPUT 1

```
A
F
D
C
B
E
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
for (const line of lines) {
  const [kind, label, ms] = line.trim().split(/\s+/);
  if (kind === "sync") console.log(label);
  else if (kind === "timeout") setTimeout(() => console.log(label), Number(ms));
  else if (kind === "micro") Promise.resolve().then(() => console.log(label));
  else if (kind === "tick") process.nextTick(() => console.log(label));
}
```

EXERCISE 2

series and parallel, callback style

The starter's `loadUser(name, cb)` is an error-first callback API (it fails for the name `bad`, and its delay depends on the name's length). Write `series(names, cb)` — load one after another, stop at the first error calling `cb(err, resultsSoFar)`, else `cb(null, results)` — and `parallel(names, cb)` — start all at once, place results **by index** so input order is preserved, call `cb` exactly once (with the first error, or with all results). Line 1 is the series input, line 2 the parallel input. Print `series: <results joined by ,>` or `series error: <message> after <n> loaded`, then `parallel: <results>` or `parallel error: <message>`.

Example: `ann bob cy` / `dave eve` →

```
series: ANN,BOB,CY
parallel: DAVE,EVE
```

`ann bad cy` / `bob bad` → `series error: no user bad after 1 loaded` and `parallel error: no user bad`.

INPUT 1

```
ann bob cy
dave eve
```

OUTPUT 1

```
series: ANN,BOB,CY
parallel: DAVE,EVE
```

INPUT 2

```
ann bad cy
bob bad
```

OUTPUT 2

```
series error: no user bad after 1 loaded
parallel error: no user bad
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
function loadUser(name, cb) {
  setTimeout(() => {
    if (name === "bad") return cb(new Error(`no user ${name}`));
    cb(null, name.toUpperCase());
  }, name.length * 3);
}
function series(names, cb) {
  const results = [];
  let i = 0;
  function next() {
    if (i === names.length) return cb(null, results);
    loadUser(names[i++], (err, value) => {
      if (err) return cb(err, results);
      results.push(value);
      next();
    });
  }
  next();
}
function parallel(names, cb) {
  const results = new Array(names.length);
  let pending = names.length, finished = false;
  if (pending === 0) return cb(null, results);
  names.forEach((name, i) => loadUser(name, (err, value) => {
    if (finished) return;                       // an earlier error already reported:
    if (err) { finished = true; return cb(err); }
    results[i] = value;                         // by index: completion order does no
    if (--pending === 0) { finished = true; cb(null, results); }
  }));
}
series(lines[0].trim().split(/\s+/), (err, results) => {
  console.log(err ? `series error: ${err.message} after ${results.length} loaded` :
  parallel(lines[1].trim().split(/\s+/), (err2, all) => {
    console.log(err2 ? `parallel error: ${err2.message}` : `parallel: ${all.join(",'
  });
});
```

QUIZ

1. Order of output: `setTimeout(() ⇒ log(1), 0); Promise.resolve().then(() ⇒ log(2)); process.nextTick(() ⇒ log(3)); log(4);`

A. 1 2 3 4

B. 4 3 2 1

C. 4 2 3 1

D. 4 1 3 2

2. A microtask that always queues another microtask…

A. Runs once per timer tick

B. Starves the event loop — no timer or I/O callback ever runs, because all microtasks drain before the next task

C. Is throttled by Node

D. Is a syntax error

3. `setTimeout(fn, 100)` guarantees…

A. `fn` runs at exactly 100 ms

B. `fn` runs no sooner than 100 ms, once the stack is free — later under load

C. `fn` runs before any pending promise

D. Nothing

4. In the error-first callback convention, after handling `err` you should…

A. Continue with the success path

B. `return` — otherwise the success code runs with an undefined result

C. Throw it

D. Call the callback again

5. An error thrown inside a `setTimeout` callback…

A. Is caught by a `try` around the `setTimeout` call

B. Cannot be caught by the scheduling code — its stack is gone; it becomes an uncaught exception

C. Is ignored

D. Rejects a promise

ANSWER KEY · CALLBACKS AND THE EVENT LOOP — ONE THREAD, TWO QUEUES

1. B Sync → nextTick → promise microtasks → timers.

2. B To really yield, use `setImmediate` or `setTimeout(fn, 0)`.

3. B Timers are minimums; same-delay timers fire in creation order.

4. B `if (err) return cb(err);` is the idiom; call the callback exactly once.

5. B Callbacks must pass errors to their callback, not throw; promises fix this.

LESSON 2 · 14 MIN

Promises — a value that is not here yet

A promise is an object standing in for a result that will arrive later: it is **pending**, then either **fulfilled** with a value or **rejected** with a reason, exactly once, forever. Everything else — `then`, chaining, `catch`, `async` / `await` — is built on that one idea. Promises fixed what callbacks could not: a result you can hold, pass around and attach to after the fact; errors that propagate along a chain to one `catch`; and composition operators (`all`, `race`) that work on any asynchronous operation. This lesson covers the state machine, the chaining rules that trip people up (return!), how rejections travel, how to build promises around callback code, and the mistakes that show up in code review every week.

States and settling

```
const p = new Promise((resolve, reject) ⇒ {     // the executor runs synchron
  setTimeout(() ⇒ resolve(42), 100);            // later: settle it
});
p.then((v) ⇒ console.log(v));                   // 42, when it settles — and
```

A promise settles once: further `resolve` / `reject` calls are ignored. Reactions attached with `then` run as **microtasks** — even if the promise is already settled, the callback runs after the current synchronous code finishes. That guarantee ("always async") is what makes ordering predictable. The value a promise holds can be anything except another promise: resolving with a promise (or any object with a `then` method — a *thenable*) **adopts** its state instead, which is why chains flatten.

`then` returns a new promise — the chaining rules

```
fetchUser(id)
  .then((user) ⇒ user.teamId)                  // return a value → the next pr
  .then((teamId) ⇒ fetchTeam(teamId))          // return a PROMISE → the next
  .then((team) ⇒ { console.log(team.name); }) // return nothing → the next pr
  .catch((err) ⇒ console.error(err))           // any rejection above lands he
  .finally(() ⇒ hideSpinner());                // runs either way, receives no
```

Each `then(onFulfilled, onRejected)` returns a **new** promise settled by what the handler does: return a value → fulfil with it; return a promise/thenable → follow it; throw → reject with the thrown value. This is the whole algebra. The single most common bug is a handler that *does* asynchronous work but forgets to **return** the promise: the chain moves on immediately with `undefined`, and the error from the un-returned promise is unhandled.

`catch(fn)` is `then(undefined, fn)`; a rejection skips every `then` handler down to the first `catch`; a `catch` that returns a value **recovers** — the chain continues fulfilled. `finally(fn)` runs on either outcome and, unless it throws or returns a rejected promise, is transparent to the value.

Errors: throw, reject, propagate

Inside an executor or a `then` handler, `throw` is the same as rejecting. Rejections propagate along the chain until handled. A promise that rejects with **no handler attached** is an *unhandled rejection*: Node prints a warning and (since Node 15) **crashes the process** by default; browsers fire `unhandledrejection`. Every chain you create must end in a `catch` or be returned to someone who will handle it. Reject with `Error` objects, for the same reasons you throw them.

Two handlers in one `then` versus `then` / `catch` differ: `then(ok, fail)` — `fail` does **not** see errors thrown by `ok`; `then(ok).catch(fail)` — it does. Prefer the latter.

Creating promises

`new Promise(executor)` — only when wrapping something callback- or event-based. If you find yourself writing `new Promise` around code that already returns promises, stop: that is the *explicit-construction antipattern*.

`Promise.resolve(v)` — a fulfilled promise (or `v` itself if it is already a promise; a thenable is adopted). `Promise.reject(err)`.

`util.promisify(fn)` (Node) — turns an error-first callback function into a promise-returning one; most of `fs` has a `fs.promises` version already.

An `async` function — returns a promise automatically (next lesson).

```
function sleep(ms) { return new Promise((resolve) ⇒ setTimeout(resolve, ms));
function once(emitter, event) {
  return new Promise((resolve, reject) ⇒ {
    emitter.once(event, resolve);
    emitter.once("error", reject);
  });
}
```

A promise is a value

You can store it, return it, pass it, attach handlers any number of times and at any time — even years after it settled — and every handler gets the same result. Start an operation *now* and await it *later*: `const userP = fetchUser(id); const teamP = fetchTeam(tid); const [user, team] = await Promise.all([userP, teamP]);` runs both concurrently. Cache the promise, not the value, to dedupe concurrent requests for the same resource (`cache.get(key) ?? cache.set(key, load(key)).get(key)`).

Anatomy of a hand-written `then` (for understanding)

A promise keeps its state, its value, and a list of reactions. `then` pushes `{ onFulfilled, onRejected, resolveNext, rejectNext }` onto that list and returns the next promise. When the promise settles, each reaction is queued as a microtask that calls the appropriate handler and settles the next promise from the handler's return or throw — with the adoption rule for returned thenables. Thirty lines implement it; the "Promises/A+" test suite has 800 cases for the corners. You will not write one in production, but interviewers ask for a minimal version and the exercise in this lesson makes you trace the rules by hand.

Common mistakes

- Forgetting to `return` a promise from inside `then` — the chain runs ahead; the error is unhandled.
- Nesting `then` inside `then` instead of returning and chaining flat.
- No terminal `catch` (unhandled rejection crashes Node).
- `new Promise` around promise-returning code; `resolve` called with a callback-style `(err, value)` pair.

Expecting `then` callbacks to run synchronously for already-settled promises.

Using `then(ok, fail)` and missing errors thrown in `ok`.

Rejecting with strings.

Interview angle

"What are the states of a promise?" Pending, fulfilled, rejected; settles once; reactions run as microtasks.

"What does `then` return?" A new promise settled by the handler's return value (adopting a returned promise) or thrown error.

"Difference between `then(ok, fail)` and `then(ok).catch(fail)`?" Only the second catches errors thrown inside `ok`.

"What is an unhandled rejection?" A rejected promise with no handler; Node exits on it by default.

"How do you convert a callback API to promises?" `util.promisify` or `new Promise` wrapping the callback, resolving/rejecting from it.

Key takeaways

Pending → fulfilled/rejected, once; `then` handlers always run asynchronously as microtasks.

`then` returns a new promise; return a value to pass it on, a promise to wait for it, throw to reject; **always return** the inner promise.

Rejections skip to the nearest `catch`; recovery continues the chain; end every chain in `catch` or return it.

Create with `new Promise` only around callbacks/events; `Promise.resolve`, `promisify`, `async` functions otherwise.

A promise is a value: start now, await later, cache the promise to dedupe.

EXERCISE 1

A chain with one catch

Given `parse` (throws `TypeError("not a number: <text>")`), `validate` (returns a rejected promise `RangeError("negative: <n>")` for negatives, else the number), `double` (asynchronous, returns a

promise) and `format`, process each input token through the chain `parse → validate → double → format` using `then`, print `<token> → <result>`, recover in a single `catch` printing `<token> → <name>: <message>`, and count settled chains in `finally`. Process the tokens **one after another** (a `reduce` over a promise chain) so the lines print in input order; end with `settled=<n>`.

Example: `3 -2 x 7` →

```
3 → 6.0
-2 → RangeError: negative: -2
x → TypeError: not a number: x
7 → 14.0
settled=4
```

INPUT 1

```
3 -2 x 7
```

OUTPUT 1

```
3 → 6.0
-2 → RangeError: negative: -2
x → TypeError: not a number: x
7 → 14.0
settled=4
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sleep = (ms, value) => new Promise((resolve) => setTimeout(resolve, ms, value)
const parse = (text) => { const n = Number(text); if (Number.isNaN(n)) throw new Typ
const validate = (n) => (n < 0 ? Promise.reject(new RangeError(`negative: ${n}`)) :
const double = (n) => sleep(2, n * 2);        // asynchronous step
const format = (n) => `${n.toFixed(1)}`;
let settled = 0;
const tokens = lines[0].trim().split(/\s+/);
tokens
  .reduce((chain, token) => chain.then(() =>
    Promise.resolve(token)
      .then(parse)                    // a throw here becomes a rejection
      .then(validate)                 // returns a value or a rejected promise — bo
      .then(double)                   // returns a promise: the chain waits for it
      .then(format)
      .then((out) => console.log(`${token} → ${out}`))
      .catch((err) => console.log(`${token} → ${err.name}: ${err.message}`))
      .finally(() => { settled++; })
  ), Promise.resolve())
  .then(() => console.log(`settled=${settled}`));
```

EXERCISE 2

promisify by hand

Write `promisify(fn)` that turns an error-first callback function into one returning a promise. Use it on the starter's `readFile(name, cb)` (fails with `code: "ENOENT"` for names starting with `missing`). Read the input names one after another, printing `<name>: <contents>` or `<name>: <error.code>`. Then print `identity=<Promise.resolve(p) === p for a promise p>` and resolve a thenable `{ then(resolve) { resolve("adopted"); } }` through `Promise.resolve`, printing `thenable=<value>`.

Example: `a.txt missing.txt b.txt` →

```
a.txt: <contents of a.txt>
missing.txt: ENOENT
b.txt: <contents of b.txt>
identity=true
thenable=adopted
```

INPUT 1

```
a.txt missing.txt b.txt
```

OUTPUT 1

```
a.txt: <contents of a.txt>
missing.txt: ENOENT
b.txt: <contents of b.txt>
identity=true
thenable=adopted
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
function readFile(name, cb) {
  setTimeout(() => {
    if (name.startsWith("missing")) return cb(Object.assign(new Error(`ENOENT: ${nam
    cb(null, `<contents of ${name}>`);
  }, 3);
}
function promisify(fn) {
  return (...args) => new Promise((resolve, reject) => {
    fn(...args, (err, value) => (err ? reject(err) : resolve(value)));
  });
}
const read = promisify(readFile);
const names = lines[0].trim().split(/\s+/);
names
  .reduce((chain, name) => chain.then(() => read(name).then(
    (text) => console.log(`${name}: ${text}`),
    (err) => console.log(`${name}: ${err.code}`),
  )), Promise.resolve())
  .then(() => {
    const p = Promise.resolve(1);
    console.log(`identity=${Promise.resolve(p) === p}`);
    const thenable = { then(resolve) { resolve("adopted"); } };
    return Promise.resolve(thenable);        // a thenable is adopted, not wrapped
  })
  .then((v) => console.log(`thenable=${v}`));
```

QUIZ

1. Inside a `then` handler you call an async function but do not `return` its promise. The consequence is…

A. Nothing

B. The chain continues immediately with `undefined`, and the inner promise's rejection is unhandled

C. The chain waits anyway

D. A syntax error

2. `Promise.resolve(42).then(log); log("sync")` prints…

A. 42 then sync

B. sync then 42 — `then` callbacks always run asynchronously as microtasks, even for a settled promise

C. Only 42

D. Only sync

3. `p.then(ok, fail)` versus `p.then(ok).catch(fail)`:

A. Identical

B. Only the second catches an error thrown inside `ok`

C. Only the first catches rejections of `p`

D. The second is slower

4. A rejected promise with no handler attached, in Node 16…

A. Is silently ignored

B. Triggers an unhandled rejection that terminates the process by default

C. Retries

D. Is logged once

5. Wrapping `fetchUser()` (which already returns a promise) in `new Promise((res, rej) ⇒ fetchUser().then(res, rej))` is…

A. Required

B. The explicit-construction antipattern — just use `fetchUser()`; `new Promise` is for callback/event APIs

C. Faster

D. Safer

ANSWER KEY · PROMISES — A VALUE THAT IS NOT HERE YET

1. B Always return the inner promise so the chain adopts it.

2. B That guarantee is what makes ordering predictable.

3. B Prefer `then(ok).catch(fail)`.

4. B Every chain must end in `catch` or be returned to code that handles it.

5. B `util.promisify` or `async` functions cover the other cases.

LESSON 3 · 14 MIN

async/await — promises with the syntax of ordinary code

`async` / `await` (2017) did not add a capability; it added **readability**. An `async` function returns a promise; `await` pauses the function until a promise settles and gives you its value — or throws its rejection — so asynchronous steps read top to bottom with `try` / `catch`, loops and early returns like any synchronous code. The cost of that readability is a handful of rules people learn by getting them wrong: `await` in a loop is sequential, a missing `await` is a silent bug, `return promise` inside `try` does not get caught, and every `await` yields to the event loop even when the value is already there. This lesson covers the semantics precisely, the sequential-versus-concurrent decision, error handling, and the patterns that make async code correct as well as readable.

The two keywords

```
async function loadDashboard(userId) {            // returns a Promise, always
  const user = await fetchUser(userId);            // pause here until fetchUs
  const team = await fetchTeam(user.teamId);       // runs after — sequential
  return { user, team };                           // fulfils the returned pro
}
loadDashboard(1).then(render).catch(showError);    // callers see a normal pro
```

`async` marks a function whose return value is wrapped in a promise: `return x` fulfils with `x` (a returned promise is adopted), `throw` rejects. Works on declarations, expressions, arrows, methods (`async method() {}`), and generators (`async function*`).

`await p` takes a promise (or any value — a non-promise is wrapped, so `await 5` is `5` one microtask later), suspends the async function, lets the event loop run, and resumes when `p` settles: the fulfilled value becomes the expression's value; a rejection is **thrown** at the `await`. Only valid inside `async` functions and at the top level of ES modules.

Everything before the first `await` runs synchronously when the function is called; the function returns its promise at the first `await`. After each `await` the continuation runs as a microtask.

Sequential or concurrent — you decide

```
// sequential: ~600 ms total if each takes 200
const a = await fetchA();
const b = await fetchB();
const c = await fetchC();

// concurrent: ~200 ms — start all three, then wait
const [a2, b2, c2] = await Promise.all([fetchA(), fetchB(), fetchC()]);

// also concurrent, without all(): start first, await later
const pA = fetchA(); const pB = fetchB();
const a3 = await pA; const b3 = await pB;
```

`await` waits for one thing. Independent operations awaited one after another run one after another — the most common performance bug in async code. Start them (call the functions, holding the promises), then await. Dependent operations (`fetchTeam(user.teamId)`) must be sequential by nature.

The same applies to loops: `for (const id of ids) results.push(await load(id))` loads one at a time; `await Promise.all(ids.map((id) ⇒ load(id)))` loads all at once. Choose sequential when order or rate matters (writing in order, a rate-limited API), concurrent when it does not; bound the concurrency for large lists (next lesson). `forEach` with an `async` callback awaits **nothing** — the loop finishes immediately and errors are lost; use `for…of` or `map` + `Promise.all`.

Error handling

```
async function readConfig(path) {
  try {
    const text = await fs.promises.readFile(path, "utf8");
    return JSON.parse(text);
  } catch (err) {
    if (err.code === "ENOENT") return {};                    // expected: mis
    throw new ConfigError(`cannot load ${path}`, { cause: err });   // otherwi
  } finally {
    metrics.increment("config.read");
  }
}
```

A rejected `await` throws at that line, so `try` / `catch` covers asynchronous failures exactly like synchronous ones — including errors from several awaited steps in one block. The module-7

rules apply unchanged: catch at boundaries, guard types, wrap with `cause`.

The trap: `try { return doWork(); } catch …` — returning a promise **without awaiting** exits the `try` before the promise rejects; the `catch` never fires and the caller gets the rejection. Inside a `try` (or when `finally` cleanup must wait), write `return await doWork()`. Elsewhere a bare `return doWork()` is fine and saves a microtask.

The missing `await`

`const data = fetchData();` (no `await`) gives a promise, so `data.items` is `undefined` — the classic "Cannot read properties of undefined". `fetchData();` as a statement with no `await` and no `catch` is a **floating promise**: its work runs, its result is dropped, and its rejection is unhandled (a crash in Node). Linters (`@typescript-eslint/no-floating-promises`) exist for this. Every promise must be awaited, returned, or explicitly handled (`void p.catch(log)` when fire-and-forget is intended).

Concurrency without threads

An `async` function's body runs in slices between `await`s; no other JavaScript runs *during* a slice, but other code runs *between* slices. Shared state read before an `await` may have changed after it:

```
if (!cache.has(key)) {                    // two callers both see "missing"
  const value = await load(key);          // both load
  cache.set(key, value);
}
```

Fix by caching the **promise** (`cache.set(key, load(key))` before awaiting) or with an explicit lock/queue. Check-then-act across an `await` is the async race condition.

Top-level `await` and startup

In ES modules `await` works at module top level (Node 14.8+), which makes `const config = await loadConfig();` in an entry module legal and makes module evaluation asynchronous — importers wait. In CommonJS (`require`) it is a syntax error; wrap in an `async function`

`main()` and call `main().catch((err) ⇒ { console.error(err); process.exit(1); })` — the standard entry pattern that also handles startup failures.

Under the hood

`async` / `await` desugars to a generator-like state machine driven by promises: each `await` is a `yield` of the promise to a driver that resumes the function with the settled value in a `then` callback. That is why continuations are microtasks, why `await` on an already-fulfilled promise still yields once, and why stack traces across `await` need V8's async-stack-trace reconstruction.

Common mistakes

Sequential awaits for independent work; `forEach(async …)`.

Missing `await`: reading a promise as a value, or a floating promise with an unhandled rejection.

`return promise` inside `try` — the `catch` cannot see the rejection.

`await` inside a non-async callback (`arr.map((x) ⇒ await f(x))` is a syntax error; mark the callback `async`).

Assuming nothing changes across an `await` (check-then-act races).

Wrapping an async function in `new Promise` — it already returns one.

Interview angle

"What does `async` do to a function?" Makes it return a promise; `return` fulfils, `throw` rejects.

"What does `await` do?" Suspends the async function until the promise settles, yielding to the event loop; resumes with the value or throws the rejection.

"How do you run async operations concurrently with `await`?" Start them first (or `map` to promises), then `await Promise.all`.

"`return promise` versus `return await promise`?" Inside `try` / `finally` only `return await` lets the block see the rejection; elsewhere both are fine.

"Why does `forEach(async …)` not work?" `forEach` ignores the returned promises; nothing waits and errors are lost.

Key takeaways

`async` → returns a promise; `await` → pause, yield, resume with the value or throw the rejection; continuations are microtasks.

Sequential awaits are sequential; start independent work first, then `await Promise.all`; never `forEach(async)`.

`try` / `catch` / `finally` work across awaits; `return await` inside `try`; wrap with `cause`, catch at boundaries.

Every promise is awaited, returned, or handled — a floating promise is a bug and, on rejection, a crash.

State can change across an `await`: cache promises, not values; check-then-act needs a lock.

EXERCISE 1

Sequential versus concurrent, observed

Each line is a task `name ms`. Run them **sequentially** (`await` in a `for…of`, recording the completion order) and then **concurrently** (`Promise.all` over async callbacks that each record their completion). Print `sequential: <order>`, `parallel: <order>`, and `sequentialTime=<sum of ms> parallelTime=<max ms>` (computed from the numbers, not measured).

Example: `a 30`, `b 10`, `c 20` →

```
sequential: a b c
parallel: b c a
sequentialTime=60 parallelTime=30
```

Sequential completes in input order; concurrent completes in duration order.

INPUT 1

```
a 30
b 10
c 20
```

OUTPUT 1

```
sequential: a b c
parallel: b c a
sequentialTime=60 parallelTime=30
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sleep = (ms, value) => new Promise((resolve) => setTimeout(resolve, ms, value)
const tasks = lines.map((l) => { const [name, ms] = l.trim().split(/\s+/); return {
async function main() {
  const sequential = [];
  for (const t of tasks) {
    await sleep(t.ms);
    sequential.push(t.name);          // one at a time, in input order
  }
  const parallel = [];
  await Promise.all(tasks.map(async (t) => {
    await sleep(t.ms);
    parallel.push(t.name);            // all started together: finishes in duration
  }));
  console.log(`sequential: ${sequential.join(" ")}`);
  console.log(`parallel: ${parallel.join(" ")}`);
  const total = tasks.reduce((s, t) => s + t.ms, 0);
  const longest = tasks.reduce((m, t) => Math.max(m, t.ms), 0);
  console.log(`sequentialTime=${total} parallelTime=${longest}`);
}
main();
```

EXERCISE 2

return await, and the forEach that waits for nothing

The starter's `work(id)` rejects for ids ending in `x`. `withoutAwait(id)` does `try { return work(id); } catch { … }`; write `withAwait(id)` identically but with `return await work(id)`. For each id call both inside an outer `try` / `catch` and print `<id>: withoutAwait → <result> | withAwait → <result>`, where a rejection caught inside prints `caught inside: <message>` and one caught by the outer block prints `caught outside: <message>`. Then run `ids.forEach(async (id) => { await work(id).catch(() => {}); finished++; })` and immediately print `afterForEach=<finished>`; then `await Promise.all(ids.map(...))` the same way and print `afterPromiseAll=<finished>`.

Example: `a bx c` →

```
a: withoutAwait → a done | withAwait → a done
bx: withoutAwait → caught outside: bx failed | withAwait → caught inside: bx faile
c: withoutAwait → c done | withAwait → c done
afterForEach=0
afterPromiseAll=3
```

INPUT 1

```
a bx c
```

OUTPUT 1

```
a: withoutAwait → a done | withAwait →
a done
bx: withoutAwait → caught outside: bx
failed | withAwait → caught inside: bx
failed
c: withoutAwait → c done | withAwait →
c done
afterForEach=0
afterPromiseAll=3
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sleep = (ms, value) => new Promise((resolve) => setTimeout(resolve, ms, value)
async function work(id) {
  await sleep(1);
  if (id.endsWith("x")) throw new Error(`${id} failed`);
  return `${id} done`;
}
async function withoutAwait(id) {
  try { return work(id); } catch (err) { return `caught inside: ${err.message}`; }
}
async function withAwait(id) {
  try { return await work(id); } catch (err) { return `caught inside: ${err.message}
}
async function main() {
  const ids = lines[0].trim().split(/\s+/);
  for (const id of ids) {
    let a, b;
    try { a = await withoutAwait(id); } catch (err) { a = `caught outside: ${err.mes
    try { b = await withAwait(id); } catch (err) { b = `caught outside: ${err.messag
    console.log(`${id}: withoutAwait → ${a} | withAwait → ${b}`);
  }
  let finished = 0;
  ids.forEach(async (id) => { await work(id).catch(() => {}); finished++; });
  console.log(`afterForEach=${finished}`);                // forEach did not wait f
  await Promise.all(ids.map((id) => work(id).catch(() => {})));
  console.log(`afterPromiseAll=${finished}`);
}
main();
```

QUIZ

1. `const a = await f(); const b = await g();` for independent `f` and `g` …

A. Runs them concurrently

B. Runs them one after another — start both first (`const pa = f(), pb = g()`) or use `Promise.all` to overlap

C. Is a syntax error

D. Runs `g` first

2. `items.forEach(async (x) ⇒ { await save(x); })` followed by `console.log("done")` …

A. Logs after every save

B. Logs immediately — `forEach` ignores the returned promises; errors are lost too

C. Throws

D. Saves sequentially

3. Inside `try { return doWork(); } catch (e) { … }` in an async function, a rejection from `doWork()` …

A. Is caught by the `catch`

B. Skips the `catch` — the function already returned the promise; write `return await doWork()`

C. Is swallowed

D. Crashes

4. `const data = fetchData(); data.items` (no `await`) fails because…

A. `fetchData` is slow

B. `data` is a promise, so `.items` is `undefined`

C. `items` is private

D. It does not fail

5. `if (!cache.has(k)) { cache.set(k, await load(k)); }` called twice at once loads twice because…

A. Maps are slow

B. Both callers pass the check before either `await` resumes — state changes across an `await`; cache the promise instead

C. `await` is synchronous

D. `has` is async

ANSWER KEY · ASYNC/AWAIT — PROMISES WITH THE SYNTAX OF ORDINARY CODE

1. B Sequential awaits are the most common async performance bug.

2. B Use `for…of` with `await`, or `await Promise.all(items.map(...))`.

3. B Outside `try` / `finally`, a bare `return promise` is fine.

4. B A floating promise as a statement is the other form of the missing `await`.

5. B Check-then-act across an `await` is the async race condition.

LESSON 4 · 14 MIN

Combinators, concurrency limits, timeouts and cancellation

Once every asynchronous operation is a promise, coordinating many of them becomes algebra: wait for **all**, wait for **any**, wait for the **first**, wait for all **regardless** of outcome. The four `Promise` combinators cover the common shapes; three patterns you write yourself cover the rest — a **concurrency limit** so a thousand requests do not all start at once, a **timeout** so nothing waits forever, and **cancellation** through `AbortController` so abandoned work stops. This lesson covers each, plus async iteration (`for await`) for sequences that arrive over time.

The four combinators

```
const [user, team] = await Promise.all([fetchUser(id), fetchTeam(tid)]);
```

Combinator	Fulfils when	Rejects when	Result
`Promise.all(list)`	every promise fulfils	the **first** rejection (others keep running, results dropped)	array of values, in input order
`Promise.allSettled(list)`	always, once all settle	never	array of `{ status: "fulfilled", value }` / `{ status: "rejected", reason }`
`Promise.race(list)`	the first to **settle** fulfils	the first to settle rejects	that value / reason
`Promise.any(list)`	the first to **fulfil**	all reject → `AggregateError` with `.errors`	that value

All four accept any iterable of promises *or plain values* (values are wrapped). `all` is the workhorse: fan out, wait, get results in order. `allSettled` is for batches where one failure must not hide the others. `race` is the building block for timeouts. `any` is "first success wins" —

redundant sources, fastest mirror. An empty `all` / `allSettled` fulfils with `[]` immediately; an empty `race` stays pending forever; an empty `any` rejects.

Two details: rejection in `all` does not *cancel* the others — they finish and their results vanish; and each combinator attaches handlers to every input, so an input that rejects after `all` already rejected is still "handled" (no unhandled-rejection crash).

Bounded concurrency

`Promise.all(urls.map(fetch))` starts **every** request at once — fine for five, a self-inflicted denial of service for five thousand. A **pool** runs at most `n` at a time:

```
async function mapLimit(items, limit, worker) {
  const results = new Array(items.length);
  let next = 0;
  async function runner() {
    while (next < items.length) {
      const i = next++;                       // claim an index — safe: no awa
      results[i] = await worker(items[i], i);
    }
  }
  await Promise.all(Array.from({ length: Math.min(limit, items.length) }, runn
  return results;
}
```

`limit` runners each pull the next index when they finish — a work queue with no explicit queue object. Results land by index, so order is preserved. Real code adds error policy (fail fast versus collect via the `allSettled` shape) and this is exactly what libraries like `p-limit` / `p-map` do.

Timeouts

```
function withTimeout(promise, ms, message = `timed out after ${ms} ms`) {
  let timer;
  const timeout = new Promise((_, reject) => { timer = setTimeout(() => reject
  return Promise.race([promise, timeout]).finally(() => clearTimeout(timer));
}
```

`race` against a timer. The `finally` clears the timer so a fast result does not leave a pending timeout keeping the process alive (and firing a rejection nobody handles — `race` handled it, but the timer itself still runs). Node 16 also has `require("timers/promises").setTimeout(ms, value, { signal })` — a promise-based sleep that accepts an abort signal.

Cancellation: `AbortController`

Promises cannot be cancelled — a settled-once value has no "stop" — so cancellation is a **signal** passed into the operation, which checks it:

```
const controller = new AbortController();
setTimeout(() => controller.abort(), 500);                    // give up after
try {
  const res = await fetch(url, { signal: controller.signal });    // fetch re
} catch (err) {
  if (err.name === "AbortError") console.log("cancelled"); else throw err;
}

// honouring a signal in your own code
async function poll(fn, { signal }) {
  while (!signal.aborted) {
    await fn();
    await timersPromises.setTimeout(1000, undefined, { signal });   // sleeps
  }
  signal.throwIfAborted?.();                                      // Node 17
}
```

`signal.aborted`, `signal.addEventListener("abort", …)`, and `signal.reason` (Node 17.2+) are the surface. `fetch`, `timers/promises`, `fs.promises.readFile`, streams and `events.once` all accept a `signal`. Abort **rejects** with an error named `AbortError` — treat it as "stop", not as failure. `AbortSignal.timeout(ms)` (Node 16.14+) builds the timeout case directly — on older runtimes combine a controller with a timer.

Retry with delay

```
async function retry(fn, { attempts = 3, delayMs = 100, isTransient = () ⇒ tr
  for (let i = 1; ; i++) {
    try { return await fn(i); }
    catch (err) {
      if (i ⩾ attempts || !isTransient(err)) throw err;
      await sleep(delayMs * 2 ** (i - 1));      // exponential backoff: 100, 2
    }
  }
}
```

Module 7's rules — transient only, idempotent only, bounded — plus a delay that grows (and, in production, a little random *jitter* so a thousand clients do not retry in lockstep).

Async iteration

```
for await (const chunk of stream) { … }          // any object with [Symbol.as
async function* pages(url) {                      // an async generator: yield
  while (url) { const page = await fetchJson(url); yield* page.items; url = pa
}
for await (const item of pages(first)) { if (enough(item)) break; }   // breal
```

`for await` awaits each `next()`; it also accepts a *sync* iterable of promises, awaiting each in order. Async generators are the natural producer for paginated APIs, line-by-line file reading (`readline`), and event streams (`events.on(emitter, "data")` returns one). Items are processed **sequentially** — one at a time, in order — which is often the point.

Events to promises

`const [value] = await events.once(emitter, "ready")` (Node) resolves on the first emission and rejects on `"error"`. For a browser `EventTarget`, wrap `addEventListener(…, { once: true })` in `new Promise`. Streams: `stream/promises` `pipeline` and `finished`.

Common mistakes

- `Promise.all` for a batch where partial results matter (use `allSettled`); expecting `all` to cancel the rest on failure.

Unbounded `map(fetch)` over large lists.

A timeout `race` that does not clear its timer; a timeout that does not abort the underlying operation (it keeps running).

Treating `AbortError` as a failure to log loudly.

Retrying without backoff, jitter or a bound.

`for await` when concurrency was wanted (it is sequential by design).

Interview angle

"*`Promise.all` versus `allSettled`?*" `all` rejects on the first failure and drops the rest; `allSettled` always fulfils with every outcome.

"How do you limit concurrency?" A pool of `n` runners pulling from a shared index/queue; results stored by index.

"How do you add a timeout to a promise?" `race` against a timer, clear the timer in `finally`, and abort the operation with a signal if it supports one.

"How do you cancel a promise?" You cannot; pass an `AbortSignal` the operation honours and reject with `AbortError`.

"*`race` versus `any`?*" First to settle (including rejection) versus first to fulfil.

Key takeaways

`all` (values in order, fail fast), `allSettled` (every outcome), `race` (first settle), `any` (first fulfil, `AggregateError` if none).

Bound concurrency with a runner pool; it preserves order by index.

Timeout = `race` + timer + `clearTimeout` in `finally`; cancellation = `AbortController` → `AbortError`, honoured by `fetch`, timers and fs.

Retry transient, idempotent operations with bounded exponential backoff and jitter.

`for await` and async generators process sequences that arrive over time, one item at a time.

EXERCISE 1

The four combinators on one set of tasks

Each line is `name ms ok|fail`; `start(spec)` (in the starter) returns a promise that fulfils with the name or rejects with `Error("<name> failed")` after `ms`. Run fresh promises through each combinator and print: `all: <names joined by ,>` or `all: rejected <message>`; `allSettled: <name:status ...>`; `race: <name> fulfilled` or `race: <name> rejected`; `any: <name>` or `any: AggregateError with <n> errors`.

Example: `a 30 ok`, `b 10 ok`, `c 20 fail` →

```
all: rejected c failed
allSettled: a:fulfilled b:fulfilled c:rejected
race: b fulfilled
any: b
```

INPUT 1

```
a 30 ok
b 10 ok
c 20 fail
```

OUTPUT 1

```
all: rejected c failed
allSettled: a:fulfilled b:fulfilled
c:rejected
race: b fulfilled
any: b
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sleep = (ms, value) => new Promise((resolve) => setTimeout(resolve, ms, value)
const specs = lines.map((l) => { const [name, ms, outcome] = l.trim().split(/\s+/);
const start = (s) => sleep(s.ms).then(() => (s.ok ? s.name : Promise.reject(new Erro
async function main() {
  try {
    console.log(`all: ${(await Promise.all(specs.map(start))).join(",")}`);
  } catch (err) {
    console.log(`all: rejected ${err.message}`);
  }
  const settled = await Promise.allSettled(specs.map(start));
  console.log(`allSettled: ${settled.map((r, i) => `${specs[i].name}:${r.status}`).j
  await Promise.race(specs.map(start)).then(
    (v) => console.log(`race: ${v} fulfilled`),
    (err) => console.log(`race: ${err.message.split(" ")[0]} rejected`),
  );
  try {
    console.log(`any: ${await Promise.any(specs.map(start))}`);
  } catch (err) {
    console.log(`any: ${err.constructor.name} with ${err.errors.length} errors`);
  }
}
main();
```

EXERCISE 2

A concurrency pool and a timeout

Write `mapLimit(items, limit, worker)` — at most `limit` workers run at once, results stored by index — and `withTimeout(promise, ms)` — a `race` against a timer that rejects with `Error("timed out after <ms> ms")`, clearing the timer in `finally`. Line 1 is the limit; following lines are tasks `name ms`, plus one `timeout <taskMs> <limitMs>` line. Run the tasks through the pool (each worker sleeps `ms`, tracks the running count, records completion order and returns the upper-cased name) and print `results=<in input order> completed=<completion order> maxConcurrent=<n>`; then run `withTimeout(sleep(taskMs, "finished"), limitMs)` and print `timeout: finished` or `timeout: <message>`.

Example: `2`, `a 40`, `b 10`, `c 15`, `d 5`, `timeout 50 10` →

```
results=A,B,C,D completed=b,c,d,a maxConcurrent=2
timeout: timed out after 10 ms
```

`b` finishes at 10 and frees a slot for `c` (done at 25), then `d` (30); `a` runs until 40.

INPUT 1

```
2
a 40
b 10
c 15
d 5
timeout 50 10
```

OUTPUT 1

```
results=A,B,C,D completed=b,c,d,a
maxConcurrent=2
timeout: timed out after 10 ms
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sleep = (ms, value) => new Promise((resolve) => setTimeout(resolve, ms, value)
async function mapLimit(items, limit, worker) {
  const results = new Array(items.length);
  let next = 0;
  async function runner() {
    while (next < items.length) {
      const i = next++;
      results[i] = await worker(items[i], i);
    }
  }
  await Promise.all(Array.from({ length: Math.min(limit, items.length) }, runner));
  return results;
}
function withTimeout(promise, ms) {
  let timer;
  const timeout = new Promise((_, reject) => { timer = setTimeout(() => reject(new E
  return Promise.race([promise, timeout]).finally(() => clearTimeout(timer));
}
async function main() {
  const limit = Number(lines[0]);
  const items = [];
  let timeoutSpec = null;
  for (const line of lines.slice(1)) {
    const [a, b, c] = line.trim().split(/\s+/);
    if (a === "timeout") timeoutSpec = { taskMs: Number(b), limitMs: Number(c) };
    else items.push({ name: a, ms: Number(b) });
  }
  let running = 0, maxConcurrent = 0;
  const completed = [];
  const results = await mapLimit(items, limit, async (item) => {
    running++;
    maxConcurrent = Math.max(maxConcurrent, running);
    await sleep(item.ms);
    running--;
    completed.push(item.name);
    return item.name.toUpperCase();
  });
  console.log(`results=${results.join(",")} completed=${completed.join(",")} maxConc
  if (timeoutSpec) {
    await withTimeout(sleep(timeoutSpec.taskMs, "finished"), timeoutSpec.limitMs).th
      (v) => console.log(`timeout: ${v}`),
      (err) => console.log(`timeout: ${err.message}`),
    );
  }
}
main();
```

QUIZ

1. `Promise.all([a, b, c])` when `b` rejects…

A. Waits for `a` and `c` then rejects

B. Rejects immediately with `b`'s reason; `a` and `c` keep running but their results are dropped

C. Fulfils with the two successes

D. Cancels `a` and `c`

2. `Promise.race` versus `Promise.any`:

A. Identical

B. `race` settles with the first to settle (fulfil **or** reject); `any` fulfils with the first to fulfil and rejects only if all reject

C. `any` is faster

D. `race` ignores rejections

3. Why bound concurrency instead of `Promise.all(urls.map(fetch))` for 5,000 URLs?

A. `Promise.all` has a size limit

B. All 5,000 requests start at once — memory, sockets and the remote server suffer; a pool runs `n` at a time

C. `map` is synchronous

D. Fetch cannot be mapped

4. Cancelling a promise-based operation is done by…

A. `promise.cancel()`

B. Passing an `AbortSignal` the operation honours; on abort it rejects with an `AbortError`

C. Rejecting the promise from outside

D. Garbage collection

5. A timeout built with `race` should also…

A. Retry

B. Clear its timer in `finally` (and abort the underlying work if it supports a signal) — otherwise the timer keeps running

C. Log

D. Use `setInterval`

ANSWER KEY · COMBINATORS, CONCURRENCY LIMITS, TIMEOUTS AND CANCELLATION

1. B `allSettled` when every outcome matters.

2. B `race` against a timer is how timeouts are built.

3. B A pool of `n` runners pulling from a shared index preserves order by index.

4. B Promises cannot be cancelled; `fetch`, `timers/promises` and `fs` accept `signal`.

5. B `race` does not stop the loser.

LESSON 5 · 12 MIN

Timers and scheduling — setTimeout, setInterval, nextTick, and debounce/throttle

Timers are the simplest asynchronous API and the most misunderstood: `setTimeout(fn, 100)` promises "not before 100 ms", not "at 100 ms"; `setInterval` drifts and overlaps; a forgotten timer keeps a Node process alive; and the three "run this soon" calls — `process.nextTick`, `queueMicrotask`, `setImmediate` — mean three different things. This lesson covers each primitive precisely, the promise-based timers in `timers/promises`, the two rate-control patterns every UI and every API client needs (debounce, throttle), and measuring time correctly.

`setTimeout` and `clearTimeout`

```
const id = setTimeout(() ⇒ console.log("later"), 100);   // returns a handle
clearTimeout(id);                                          // cancels if it h
setTimeout(greet, 100, "Ada");                             // extra arguments
```

The delay is a **minimum**: the callback runs at the first timers phase after the delay elapses *and* the stack is free. A delay of `0` (or omitted) means "next timers phase"; browsers clamp nested timeouts to ≥ 4 ms and background tabs to ≥ 1 s; Node clamps below 1 to 1 and above $2^{31}-1$ to 1 (with a warning — a 25-day timer overflows). Same-delay timers fire in creation order. The callback's `this` is `undefined`/the global — use an arrow.

`setInterval`, and why a recursive timeout is often better

```
const id = setInterval(tick, 1000);
clearInterval(id);
```

`setInterval` schedules every `n` ms regardless of how long `tick` took — if `tick` is slow or the loop is busy, callbacks bunch up and drift accumulates; an async `tick` can overlap with itself. A **recursive** `setTimeout` — `function loop() { tick(); setTimeout(loop, 1000); }` or, with `await`, `while (running) { await tick(); await sleep(1000); }` — waits for the

work to finish before scheduling the next run and never overlaps. Use `setInterval` for cheap, fixed-cadence work (a clock display); use the recursive form for anything that does I/O.

Keeping the process alive — `ref` / `unref`

In Node an active timer (or interval) keeps the event loop running: a script with a pending `setTimeout` does not exit until it fires. `timer.unref()` says "do not keep the process alive for me" — right for a periodic cache sweep or a heartbeat that should stop when the real work ends. `timer.ref()` reverses it. Forgetting to clear an interval is the usual reason a CLI "hangs" after printing its result.

The "soon" family

Call	When it runs	Use
`process.nextTick(fn)`	before any other queued microtask, right after the current operation	Node-specific; emit an event *after* the constructor returns; rarely in application code
`queueMicrotask(fn)`	after nextTicks, with promise reactions, before any task	standard; "after this sync code, before anything else"
`Promise.resolve().then(fn)`	same queue as `queueMicrotask`	idiomatic in promise code
`setImmediate(fn)`	the check phase — after I/O callbacks of this iteration, before the next timers	"yield to the event loop, then continue" (Node); breaking up CPU work
`setTimeout(fn, 0)`	the next timers phase	portable "yield"; slightly later than `setImmediate` after I/O

Microtasks (`nextTick` , `queueMicrotask`) do **not** let I/O or timers run in between — a recursive microtask starves the loop. To yield for real, use `setImmediate` (Node) or `setTimeout(fn, 0)` ; in browsers `requestAnimationFrame` runs before the next paint and `requestIdleCallback` when idle.

Promise-based timers

```
const { setTimeout: sleep, setInterval: every } = require("timers/promises");
await sleep(100);                                  // resolves after 100 ms
const v = await sleep(100, "value");               // resolves with "value"
await sleep(5000, undefined, { signal });          // rejects with AbortError when
for await (const tick of every(1000, Date.now())) { if (done()) break; }   //
```

`timers/promises` makes waits composable with `await`, `Promise.race` (timeouts) and `AbortSignal`. Without it, the one-liner is `const sleep = (ms) ⇒ new Promise((r) ⇒ setTimeout(r, ms))`.

Debounce and throttle

Both limit how often a frequently-fired function actually runs, differently:

```
// debounce: run once, `wait` ms after the LAST call — search-as-you-type, res
function debounce(fn, wait) {
  let timer;
  return (...args) ⇒ { clearTimeout(timer); timer = setTimeout(() ⇒ fn(...ar
}

// throttle: run at most once per `wait` ms, immediately on the first call — s
function throttle(fn, wait) {
  let last = -Infinity;
  return (...args) ⇒ { const now = Date.now(); if (now - last ⩾ wait) { last
}
```

Debounce waits for quiet: a burst of 20 keystrokes produces one call after the typing stops (the last arguments win). Throttle guarantees progress during the burst: one call per window (the first arguments in each window). Variants add a *leading/trailing* option, a `cancel()` method (clear the timer on unmount — a debounced call firing after a component is gone is a classic React bug) and a `flush()`. Lodash's `debounce` / `throttle` are the reference implementations.

Measuring time

`Date.now()` is wall-clock milliseconds — fine for timestamps, wrong for durations (it jumps when the clock is adjusted). `performance.now()` (global in Node 16+ and browsers) is a monotonic high-resolution timer for measuring: `const t0 = performance.now(); …; const ms = performance.now() - t0`. `process.hrtime.bigint()` is the nanosecond version. `console.time(label)` / `console.timeEnd(label)` is the quick-and-dirty form.

Common mistakes

Relying on exact timer timing or on `setTimeout(0)` ordering against I/O.

`setInterval` for work that does I/O — overlapping runs and drift.

A pending timer or interval keeping a script alive (or `unref` on a timer that *should* keep it alive).

Recursive `nextTick` / `queueMicrotask` starving the loop.

Debounce without `cancel` on teardown; throttle where debounce was wanted (or vice versa).

Measuring durations with `Date.now()`.

Interview angle

"What does `setTimeout(fn, 0)` guarantee?" Only that `fn` runs no sooner than the next timers phase, after the current code and all microtasks.

"Debounce versus throttle?" Debounce fires once after calls stop; throttle fires at most once per interval during the calls.

"`setInterval` versus recursive `setTimeout`?" Fixed cadence regardless of duration (drift, overlap) versus scheduling the next run after the work completes.

"`process.nextTick` versus `setImmediate`?" nextTick runs before any I/O or timers (microtask-like, first); setImmediate runs in the check phase after I/O.

"How do you measure elapsed time?" `performance.now()` — monotonic; `Date.now()` is wall-clock and can jump.

Key takeaways

Timers are minimums; same-delay order is creation order; `clearTimeout` / `clearInterval` cancel; Node timers keep the process alive unless `unref()`.

Prefer recursive `setTimeout` / `await sleep` loops to `setInterval` for real work.

`nextTick` → microtasks → `setImmediate` /timers; only the last two truly yield to I/O.

`timers/promises` gives awaitable, abortable sleeps and intervals.

Debounce (after the burst) versus throttle (during the burst), both with `cancel`; measure with `performance.now()`.

EXERCISE 1

Debounce and throttle, side by side

Write `debounce(fn, wait)` (run `fn` once, `wait` ms after the **last** call, with the last arguments) and a leading-edge `throttle(fn, wait)` (run immediately, then at most once per `wait` ms, using `Date.now()`). Line 1 is `wait`; each following line `t label` schedules a call to **both** wrappers at `t` ms. The wrapped functions print `debounce:<label>` / `throttle:<label>`; after the last call plus `2 × wait` print `calls=<n> debounceFired=<n> throttleFired=<n>`.

Example: `40`, then calls `0 A`, `10 B`, `20 C`, `100 D`, `110 E` →

```
throttle:A
debounce:C
throttle:D
debounce:E
calls=5 debounceFired=2 throttleFired=2
```

Throttle fires at the start of each burst; debounce fires once each burst is over.

INPUT 1

```
40
0 A
10 B
20 C
100 D
110 E
```

OUTPUT 1

```
throttle:A
debounce:C
throttle:D
debounce:E
calls=5 debounceFired=2 throttleFired=2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
function debounce(fn, wait) {
  let timer;
  return (...args) => {
    clearTimeout(timer);                        // every call restarts the clock
    timer = setTimeout(() => fn(...args), wait);
  };
}
function throttle(fn, wait) {
  let last = -Infinity;
  return (...args) => {
    const now = Date.now();
    if (now - last >= wait) { last = now; fn(...args); }
  };
}
const wait = Number(lines[0]);
const calls = lines.slice(1).map((l) => { const [t, label] = l.trim().split(/\s+/);
let debounceFired = 0, throttleFired = 0;
const d = debounce((label) => { debounceFired++; console.log(`debounce:${label}`); }
const th = throttle((label) => { throttleFired++; console.log(`throttle:${label}`);
for (const c of calls) setTimeout(() => { th(c.label); d(c.label); }, c.t);
const last = Math.max(...calls.map((c) => c.t));
setTimeout(() => console.log(`calls=${calls.length} debounceFired=${debounceFired} t
```

EXERCISE 2

Intervals, promise timers and an aborted sleep

Write `ticks(n, everyMs)` returning a promise: a `setInterval` every `everyMs` prints `tick <i>` and, after `n` ticks, clears itself and resolves with the count. Then, using `require("timers/promises")`: `await timers.setTimeout(5, "value")` and print `promise timer resolved with <value>`; create an `AbortController`, abort it after 5 ms, and `await timers.setTimeout(1000, undefined, { signal })` inside `try` / `catch`, printing `long sleep aborted: <err.name> aborted=<signal.aborted>`; finally create an interval, call `.unref()` on it and print `unref: process may exit` — the program must end on its own.

Example: `3` →

```
tick 1
tick 2
tick 3
interval done after 3 ticks
promise timer resolved with value
long sleep aborted: AbortError aborted=true
unref: process may exit
```

INPUT 1

```
3
```

OUTPUT 1

```
tick 1
tick 2
tick 3
interval done after 3 ticks
promise timer resolved with value
long sleep aborted: AbortError
aborted=true
unref: process may exit
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const timers = require("timers/promises");
function ticks(n, everyMs) {
  return new Promise((resolve) => {
    let count = 0;
    const id = setInterval(() => {
      count++;
      console.log(`tick ${count}`);
      if (count === n) { clearInterval(id); resolve(count); }   // forgetting this k
    }, everyMs);
  });
}
async function main() {
  const n = Number(lines[0]);
  const total = await ticks(n, 5);
  console.log(`interval done after ${total} ticks`);
  const v = await timers.setTimeout(5, "value");
  console.log(`promise timer resolved with ${v}`);
  const controller = new AbortController();
  setTimeout(() => controller.abort(), 5);
  try {
    await timers.setTimeout(1000, undefined, { signal: controller.signal });
    console.log("long sleep finished");
  } catch (err) {
    console.log(`long sleep aborted: ${err.name} aborted=${controller.signal.aborted
  }
  const keepAlive = setInterval(() => {}, 1000);
  keepAlive.unref();                    // this interval will not hold the process o
  console.log("unref: process may exit");
}
main();
```

QUIZ

1. Debounce versus throttle:

   A. Synonyms

   B. Debounce runs once after the calls stop; throttle runs at most once per interval while the calls continue

   C. Throttle waits for quiet

   D. Debounce is for scroll, throttle for typing

2. `setInterval(job, 1000)` where `job` is an async I/O task risks…

A. Nothing

B. Overlapping runs and drift — the next run is scheduled regardless of whether the last one finished; use a recursive `setTimeout` / `await` loop

C. Running too rarely

D. A syntax error

3. A Node script prints its result but does not exit. The most likely cause is…

A. A syntax error

B. A pending timer or interval keeping the event loop alive — clear it or `unref()` it

C. Too much output

D. Missing `return`

4. `process.nextTick` versus `setImmediate`:

A. Same thing

B. nextTick runs before any I/O, timer or promise microtask; setImmediate runs in the check phase after I/O — the true "yield to the loop"

C. setImmediate is first

D. nextTick is a timer

5. To measure a duration use…

A. `Date.now()` — it is precise

B. `performance.now()` — monotonic and high-resolution; `Date.now()` is wall-clock and can jump

C. `new Date()`

D. `setTimeout`

ANSWER KEY · TIMERS AND SCHEDULING — SETTIMEOUT, SETINTERVAL, NEXTTICK, AND DEBOUNCE/THROTTLE

1. B Search-as-you-type debounces; scroll handlers throttle.

2. B `setInterval` suits cheap, fixed-cadence work like a clock display.

3. B Active handles keep the loop running.

4. B Recursive nextTick starves the loop; recursive setImmediate does not.

5. B `console.time` / `timeEnd` is the quick form.

CHECKPOINT · 25 MIN

Checkpoint — Asynchronous JavaScript

This checkpoint covers the event loop (one thread, tasks versus microtasks, `nextTick`, Node's phases), error-first callbacks, promise states and the chaining rules, `async` / `await` semantics (sequential versus concurrent, `return await`, floating promises, races across `await`), the four combinators, bounded concurrency, timeouts and `AbortController`, async iteration, and timers with debounce/throttle.

How it works. Twelve questions and three programs; 70% on the questions and every program accepted clears the module.

Before you start, make sure you can answer:

- The order of sync code, `process.nextTick`, `Promise.then`, `setTimeout(0)`; why a recursive microtask starves the loop.
- What `then` returns and what happens when a handler returns a value, a promise, or throws; what an unhandled rejection does in Node.
- Why `await` in a loop is sequential; how to run independent work concurrently; why `forEach(async)` is wrong; when `return await` matters.
- `all` versus `allSettled` versus `race` versus `any`; how a concurrency pool works; how a timeout and a cancellation are built.
- Debounce versus throttle; `setInterval` versus recursive `setTimeout`; what keeps a Node process alive.

The programs are a dependency-aware job scheduler that starts each job when its prerequisites finish, a fetch pipeline with retry, backoff, timeout and per-item outcomes, and an async queue with waiting consumers.

EXERCISE 1

A dependency-aware job scheduler

Each line is a job `name ms deps` where `deps` is a comma list of job names or `-`. Write a memoised `promiseFor(name)`: it waits for every dependency's promise, then `await sleep(ms)`, records the completion order, and resolves with `{ finish }` — a **virtual** finish time equal to the latest dependency finish plus `ms`. Start every job, wait for all, and print `order: <completion order>`, `finish: <name=finish ...>` (input order) and `makespan=<max finish>`.

Example: `a 20 -`, `b 40 -`, `c 10 a`, `d 10 a,b` →

```
order: a c b d
finish: a=20 b=40 c=30 d=50
makespan=50
```

`c` starts as soon as `a` is done; `d` must wait for `b` too.

INPUT 1

```
a 20 -
b 40 -
c 10 a
d 10 a,b
```

OUTPUT 1

```
order: a c b d
finish: a=20 b=40 c=30 d=50
makespan=50
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sleep = (ms, value) => new Promise((resolve) => setTimeout(resolve, ms, value)
const jobs = new Map(lines.map((l) => {
  const [name, ms, deps] = l.trim().split(/\s+/);
  return [name, { name, ms: Number(ms), deps: deps === "-" ? [] : deps.split(",") }]
}));
const running = new Map();     // name → promise of { finish }
const order = [];
function promiseFor(name) {
  if (!running.has(name)) {
    const job = jobs.get(name);
    running.set(name, (async () => {
      const finished = await Promise.all(job.deps.map(promiseFor));     // wait for
      const startAt = finished.reduce((m, f) => Math.max(m, f.finish), 0);
      await sleep(job.ms);
      order.push(name);
      return { finish: startAt + job.ms };                              // virtual cl
    })());
  }
  return running.get(name);
}
Promise.all([...jobs.keys()].map(promiseFor)).then((results) => {
  console.log(`order: ${order.join(" ")}`);
  console.log(`finish: ${[...jobs.keys()].map((n, i) => `${n}=${results[i].finish}`)
  console.log(`makespan=${Math.max(...results.map((r) => r.finish))}`);
});
```

EXERCISE 2

A fetch pipeline with retry, backoff and timeout

The starter's `fakeFetch(spec)` takes `ms` and fails with a `TransientError` for its first `failures` calls. Write `withTimeout(promise, ms)` (race + timer, cleared in `finally`, `Error("timed out after <ms> ms")`) and `retry(fn, { attempts, isTransient })` that retries transient failures with a backoff of `2^(i-1)` ms and otherwise throws `Error("gave up after <i> attempts", { cause })`. Line 1 is `attempts timeoutMs`; each following line is `url ms failures`. Run every url through `retry(() ⇒ withTimeout(fakeFetch(spec), timeoutMs))` with `Promise.allSettled` and print, in input order, `<url>: ok after <n> attempt(s)` or `<url>: failed - <message> ← <cause name>: <cause message>`, then `summary: ok=<n> failed=<n>`.

Example: `3 50`, `u1 5 0`, `u2 5 2`, `u3 5 5`, `u4 100 0` →

```
u1: ok after 1 attempt(s)
u2: ok after 3 attempt(s)
u3: failed - gave up after 3 attempts ← TransientError: u3 attempt 3 failed
u4: failed - gave up after 1 attempts ← Error: timed out after 50 ms
summary: ok=2 failed=2
```

INPUT 1

```
3 50
u1 5 0
u2 5 2
u3 5 5
u4 100 0
```

OUTPUT 1

```
u1: ok after 1 attempt(s)
u2: ok after 3 attempt(s)
u3: failed - gave up after 3 attempts ←
TransientError: u3 attempt 3 failed
u4: failed - gave up after 1 attempts ←
Error: timed out after 50 ms
summary: ok=2 failed=2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sleep = (ms, value) => new Promise((resolve) => setTimeout(resolve, ms, value)
class TransientError extends Error { constructor(m) { super(m); this.name = "Transie
const attemptsSeen = new Map();
function fakeFetch(spec) {
  const n = (attemptsSeen.get(spec.url) ?? 0) + 1;
  attemptsSeen.set(spec.url, n);
  return sleep(spec.ms).then(() => {
    if (n <= spec.failures) throw new TransientError(`${spec.url} attempt ${n} faile
    return `${spec.url} body`;
  });
}
function withTimeout(promise, ms) {
  let timer;
  const timeout = new Promise((_, reject) => { timer = setTimeout(() => reject(new E
  return Promise.race([promise, timeout]).finally(() => clearTimeout(timer));
}
async function retry(fn, { attempts, isTransient }) {
  for (let i = 1; ; i++) {
    try { return await fn(i); }
    catch (err) {
      if (i >= attempts || !isTransient(err)) throw new Error(`gave up after ${i} at
      await sleep(2 ** (i - 1));
    }
  }
}
async function main() {
  const [attempts, timeoutMs] = lines[0].trim().split(/\s+/).map(Number);
  const specs = lines.slice(1).map((l) => { const [url, ms, failures] = l.trim().spl
  const outcomes = await Promise.allSettled(specs.map((spec) =>
    retry(() => withTimeout(fakeFetch(spec), timeoutMs), { attempts, isTransient: (e
  ));
  outcomes.forEach((o, i) => {
    const spec = specs[i];
    if (o.status === "fulfilled") console.log(`${spec.url}: ok after ${attemptsSeen.
    else console.log(`${spec.url}: failed - ${o.reason.message} <- ${o.reason.cause.
  });
  console.log(`summary: ok=${outcomes.filter((o) => o.status === "fulfilled").length
}
main();
```

EXERCISE 3

An async queue with waiting consumers

Implement `AsyncQueue` with a private items array and a private list of waiting resolvers: `put(value)` hands the value straight to the oldest waiting consumer if there is one, otherwise stores it; `take()` returns a promise — resolved immediately when an item is available, otherwise pending until a `put`; `get size` and `get waiting`. Commands: `take <consumer>` (the consumer prints `<consumer> got <value>` when its promise resolves), `put <value>`, `status` (prints `size=<items> waiting=<consumers>`). After every command `await null` so the pending `then` callbacks print before the next command runs.

Example: `take c1`, `take c2`, `status`, `put apple`, `put pear`, `status`, `put fig`, `status`, `take c3`, `status` →

```
size=0 waiting=2
c1 got apple
c2 got pear
size=0 waiting=0
size=1 waiting=0
c3 got fig
size=0 waiting=0
```

INPUT 1

```
take c1
take c2
status
put apple
put pear
status
put fig
status
take c3
status
```

OUTPUT 1

```
size=0 waiting=2
c1 got apple
c2 got pear
size=0 waiting=0
size=1 waiting=0
c3 got fig
size=0 waiting=0
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
class AsyncQueue {
  #items = [];
  #waiting = [];     // resolvers of consumers blocked in take()
  put(value) {
    if (this.#waiting.length) this.#waiting.shift()(value);   // hand it straight to
    else this.#items.push(value);
  }
  take() {
    if (this.#items.length) return Promise.resolve(this.#items.shift());
    return new Promise((resolve) => this.#waiting.push(resolve));
  }
  get size() { return this.#items.length; }
  get waiting() { return this.#waiting.length; }
}
async function main() {
  const q = new AsyncQueue();
  for (const line of lines) {
    const [cmd, arg] = line.trim().split(/\s+/);
    if (cmd === "take") q.take().then((v) => console.log(`${arg} got ${v}`));
    else if (cmd === "put") q.put(arg);
    else if (cmd === "status") console.log(`size=${q.size} waiting=${q.waiting}`);
    await null;                    // let the microtasks (the then callbacks) run be
  }
}
main();
```

CHECKPOINT QUESTIONS

1. How many threads run your JavaScript in Node?

A. One per request

B. One — asynchronous means non-blocking, not parallel; `worker_threads` add real threads with separate heaps

C. As many as CPU cores

D. Two

2. Node's `setImmediate` versus `setTimeout(fn, 0)` inside an I/O callback:

A. Same order always

B. `setImmediate` runs first — the check phase follows the poll phase before the next timers phase

C. `setTimeout` runs first

D. Unspecified

3. `await 5` …

A. Is a syntax error

B. Wraps the value in a resolved promise and yields one microtask before resuming with 5

C. Waits 5 ms

D. Returns a promise

4. An `async` function that throws before its first `await` …

A. Throws synchronously to the caller

B. Returns a rejected promise — `async` functions never throw synchronously

C. Crashes

D. Returns `undefined`

5. Resolving a promise with another promise…

A. Nests them

B. Adopts the inner promise's eventual state — chains flatten, which is why `then` can return promises

C. Throws

D. Fulfils with the promise object

6. `Promise.allSettled([])` and `Promise.race([])`:

A. Both fulfil with `[]`

B. `allSettled` fulfils with `[]` immediately; `race` stays pending forever

C. Both reject

D. Both pend

7. A retry loop should add jitter because…

A. Timers are inaccurate

B. Many clients retrying in lockstep after the same failure hit the server together; randomness spreads them out

C. Backoff is illegal

D. It is faster

8. `for await (const x of asyncGen())` processes items…

A. All at once

B. One at a time, in order — awaiting each `next()`; `break` calls `return()` on the generator

C. In reverse

D. Concurrently up to a limit

9. Callers of a function that `throw`s inside a `setTimeout` callback can catch it…

A. With `try` around the call

B. Not at all — wrap the timer in a promise so the failure becomes a rejection they can handle

C. With `finally`

D. With `process.on("exit")`

10. `timer.unref()` in Node…

A. Cancels the timer

B. Lets the process exit even though the timer is pending

C. Speeds it up

D. Makes it repeat

11. A debounced handler attached to a component that unmounts should…

A. Keep running

B. Be cancelled on teardown (`cancel()`), or it fires after the component is gone

C. Be throttled instead

D. Use `setInterval`

12. Node's `util.promisify(fn)` expects `fn` to…

A. Return a promise

B. Take an error-first callback as its last argument

C. Be synchronous

D. Be a class

ANSWER KEY · CHECKPOINT — ASYNCHRONOUS JAVASCRIPT

1. B That is why a long synchronous loop delays every pending callback.

2. B From the main module (no I/O callback) their order is unspecified.

3. B Every `await` yields to the loop, even on non-promises.

4. B The caller sees a rejection to `await` or `catch`.

5. B Any thenable is adopted the same way.

6. B An empty `any` rejects with an `AggregateError`.

7. B Exponential backoff with jitter is the standard.

8. B Use a pool when concurrency is wanted.

9. B Promises turn asynchronous throws into values.

10. B Right for heartbeats and periodic sweeps that should not hold the process open.

11. B A classic source of "setState on unmounted component" warnings.

12. B It resolves with the callback's value and rejects with its error.
