JavaScript Advanced

A Complete 5-Chapter Course

Topics covered:

Modules · Generators & Iterators · The Event Loop

Design Patterns · Working with APIs

Exercises: 15 hands-on challenges with sample solutions

Format: A4 · Dark-theme code examples · Quick-reference tables

Table of Contents

CHAPTER 1 OF 5

Modules — import/export, and Bundlers Conceptually

COURSE 3 · CH 1

Modules: import/export, and Bundlers Conceptually

Splitting code across multiple files properly, instead of one giant script — and what build tools actually do with that split

Every example across all 17 previous chapters lived in a single block of code. Real projects split logic across many files — a **module** is simply a single JS file that explicitly declares what it shares with other files (`export`) and what it borrows from them (`import`). This is the standard, native mechanism JavaScript now has for organizing larger codebases.

Named Exports

```
// math.js
export function add(a, b) {
  return a + b;
}

export const PI = 3.14159;
```

`export` in front of a function or variable declaration marks it as available to other files. A single file can have any number of named exports — there's no limit, and they can be functions, constants, classes (Intermediate Chapter 3), anything.

Importing Named Exports

```
// main.js
import { add, PI } from './math.js';

console.log(add(2, 3));   // 5
console.log(PI);          // 3.14159
```

`import { add, PI } from './math.js'` uses object-destructuring-like syntax (Intermediate Chapter 1) to pull in specific named exports — only what's listed inside the braces becomes available in this file, by exactly those names.

Default Exports

```
// logger.js
export default function log(message) {
  console.log(`[LOG] ${message}`);
}

// main.js
import log from './logger.js';   // no braces — and the name can be anything

log("App started");
```

A file can have at most **one** `export default` — used for a file's single "main" thing, conventionally one per file (one component, one class, one primary function). Importing a default export uses no braces, and the imported name doesn't need to match what it was called in the original file — `import customName from './logger.js'` would work identically.

> **A FILE CAN MIX NAMED AND DEFAULT EXPORTS**
> `export default function log(...)` alongside separate `export const LOG_LEVELS = [...]` in the same file is entirely valid — `import log, { LOG_LEVELS } from './logger.js'` imports both together in one statement.

The <script type="module"> Requirement

```
<script type="module" src="main.js"></script>
```

Browsers only allow `import` / `export` inside a script explicitly marked `type="module"` — a plain `<script src="main.js">` (Fundamentals Chapter 1) throws a syntax error the moment it hits an `import` statement. Module scripts also behave slightly differently: they're deferred automatically (run after the HTML is parsed) and run in strict mode by default.

> **ES MODULES REQUIRE A REAL SERVER, NOT FILE:// DIRECTLY**
> Opening an HTML file with `type="module"` scripts directly from disk (`file:///...`) typically fails with a CORS error in most browsers — modules need to be served over `http://` or `https://`, even just from a simple local development server, to load correctly.

Why Bundlers Exist

Native `import` / `export` works directly in modern browsers — no extra tooling needed for small projects. But real-world apps often have dozens or hundreds of module files, each requiring its own network request, plus a desire to support older browsers, minify code for smaller downloads, and use newer syntax that needs converting. A **bundler** (Webpack, Vite, esbuild, Rollup) solves this: it reads the entire web of `import` / `export` statements across every file and combines them into one (or a few) optimized output files.

```
// Conceptually, a bundler turns this:
// main.js  -> imports math.js, logger.js
// math.js  -> exports add, PI
// logger.js -> exports default log

// ...into ONE combined, minified file like:
const n=(e,t)=>e+t,o=3.14159;function r(e){console.log(`[LOG] ${e}`)}r("App started");
```

The shortened variable names, the removal of whitespace, and merging everything into one file are all things a bundler does automatically — none of it changes the actual behaviour, only the file size and number of network requests needed to load the app. This course doesn't configure a real bundler, but understanding *why* one exists is essential before reaching for React, Vue, or any modern framework's tooling.

SYNTAX	HOW MANY PER FILE?	IMPORT SYNTAX
`export function/const/class`	Any number	`import { name } from '...'`
`export default ...`	At most one	`import anyName from '...'`

Coding Challenges

CHALLENGE 1

Write the contents of a file shapes.js with two named exports: a function circleArea(radius) and a constant TAX_RATE = 0.2. Then write the contents of main.js that imports both and logs circleArea(5) and TAX_RATE.

View solution

CHALLENGE 2

Write the contents of a file validator.js with a default export — a function isValidEmail(email) doing a simple check for the presence of "@" and ".". Then write main.js importing it under the name checkEmail (a different name than it was defined with) and testing it with two example strings.

View solution

CHALLENGE 3

Write the contents of a file stringUtils.js that has BOTH a default export (a function capitalize(str)) and a named export (a function reverse(str)). Write main.js that imports both together in a single import statement and tests each one.

View solution

CHAPTER 1 QUICK REFERENCE

- **export function/const/class name** — a named export; any number per file
- **export default ...** — the default export; at most one per file
- **import { name } from './file.js'** — imports named exports by exact name
- **import anyName from './file.js'** — imports the default export; name is up to you
- **import def, { named } from './file.js'** — imports both kinds together
- **<script type="module">** — required in the browser for import/export to work at all
- **Module scripts need a real server** — file:// alone typically fails with a CORS error
- **Bundlers** (Webpack, Vite, esbuild) combine and optimize many module files into fewer, smaller ones
- **Next chapter:** generators and iterators

CHAPTER 2 OF 5

Generators and Iterators

COURSE 3 · CH 2

Generators and Iterators

Functions that can pause mid-execution and hand back control — the mechanism quietly powering for...of itself

Fundamentals Chapter 6 used `for...of` on arrays without explaining how it actually works under the hood. This chapter reveals the mechanism: the **iterator protocol** — and **generator functions**, a special syntax that makes writing a custom iterator dramatically simpler than doing it by hand.

The Iterator Protocol, By Hand

```
function makeRangeIterator(start, end) {
  let current = start;
  return {
    next() {
      if (current < end) {
        return { value: current++, done: false };
      }
      return { value: undefined, done: true };
    }
  };
}

const it = makeRangeIterator(1, 4);
console.log(it.next());   // { value: 1, done: false }
console.log(it.next());   // { value: 2, done: false }
console.log(it.next());   // { value: 3, done: false }
console.log(it.next());   // { value: undefined, done: true }
```

An **iterator** is just an object with a `next()` method, returning `{ value, done }` every call — Intermediate Chapter 2's closure pattern, used here to remember `current` between calls. `for...of` on an array calls exactly this kind of `next()` repeatedly behind the scenes, stopping once `done` becomes `true`.

Generator Functions — The Same Thing, Far Less Code

```
function* range(start, end) {
  for (let i = start; i < end; i++) {
    yield i;
  }
}

const it = range(1, 4);
console.log(it.next());   // { value: 1, done: false }
console.log(it.next());   // { value: 2, done: false }
```

`function*` (an asterisk after `function`) declares a **generator function**. Calling it doesn't run the body immediately — it returns an iterator automatically, with all the `{ value, done }` bookkeeping handled for free. `yield` is the key new keyword: it pauses execution exactly where it appears, hands back a value, and resumes from that exact point the next time `next()` is called.

> **YIELD IS GENUINELY A PAUSE, NOT JUST A RETURN**
>
> Unlike `return`, which ends a function permanently, `yield` freezes the function's local state — including the loop variable `i` above — and resumes from that exact line on the next `next()` call. This is the entire reason generators can produce values one at a time, on demand, rather than computing everything up front.

Using a Generator Directly with for...of

```
function* range(start, end) {
  for (let i = start; i < end; i++) {
    yield i;
  }
}

for (const n of range(1, 4)) {
  console.log(n);
}
// 1 2 3
```

Because a generator's return value already follows the iterator protocol, `for...of` works on it directly — no manual `.next()` calls needed at all. This is the practical payoff: write a generator once, then loop over it exactly like an array.

An Infinite Generator — Why "Lazy" Evaluation Matters

```
function* infiniteCounter() {
  let n = 1;
  while (true) {
    yield n++;
  }
}

const counter = infiniteCounter();
console.log(counter.next().value);   // 1
console.log(counter.next().value);   // 2
console.log(counter.next().value);   // 3
// Could keep calling .next() forever — values are produced ON DEMAND, never all at once
```

`infiniteCounter()` contains a genuinely infinite `while (true)` loop, yet calling it doesn't hang the program — nothing inside the loop body actually runs until `.next()` is explicitly called. This "lazy" behaviour (computing only what's actually requested) is impossible with a regular array, which would need to exist completely in memory before any of it could be used.

NEVER USE FOR...OF ON AN INFINITE GENERATOR DIRECTLY

`for (const n of infiniteCounter())` would loop forever, since `for...of` keeps calling `.next()` until `done` becomes `true` — which never happens here. Infinite generators must be driven manually with `.next()`, typically combined with a separate stopping condition.

CONCEPT	DESCRIPTION
`Iterator`	Any object with a next() method returning { value, done }
`function* name() { }`	A generator function; calling it returns an iterator automatically
`yield value`	Pauses the generator, hands back a value, remembers where it left off
`for (const x of generatorCall())`	Works directly — generators already satisfy the iterator protocol
`Lazy evaluation`	Values are computed only when actually requested via next()

Coding Challenges

CHALLENGE 1

Write a generator function* evens(max) that yields every even number from 0 up to max (inclusive). Use for...of to print every value it produces for evens(10).

View solution

CHALLENGE 2

Write a generator function* fibonacci() that yields an infinite sequence of Fibonacci numbers (1, 1, 2, 3, 5, 8, ...), starting from two seed values. Manually call .next().value six times on an instance and log each result, WITHOUT using for...of (since it's infinite).

View solution

CHALLENGE 3

Write a generator function* idGenerator(prefix) that yields strings like "prefix-1", "prefix-2", "prefix-3", incrementing forever. Create two SEPARATE generators with different prefixes ("user" and "order") and call .next().value three times on each, interleaved, to confirm they don't interfere with each other (similar in spirit to Intermediate Chapter 2's closure challenge).

View solution

CHAPTER 2 QUICK REFERENCE

- **Iterator** — any object with next(), returning { value, done }
- **function* name() { }** — generator function syntax; the asterisk is required
- **yield value** — pauses, returns a value, remembers exactly where execution stopped
- **Calling a generator** returns an iterator immediately; the body doesn't run until .next() is called
- **for...of works directly** on a generator's return value — no manual .next() needed for finite generators
- **Infinite generators** are safe — nothing runs until .next() actually requests a value
- **Never for...of an infinite generator** — drive it manually with .next() instead
- **Next chapter:** the event loop in depth — microtasks vs macrotasks

CHAPTER 3 OF 5

The Event Loop In Depth

COURSE 3 · CH 3

The Event Loop In Depth: Microtasks vs Macrotasks

Why a promise's .then() always runs before a setTimeout, even one scheduled for 0 milliseconds

Fundamentals Chapter 10 explained that JavaScript doesn't wait for asynchronous work — `console.log` lines after a `setTimeout` run before its callback. What that chapter didn't cover: when multiple asynchronous things are pending at once, JavaScript runs them in a very specific, predictable order — governed by two separate queues, not one.

Recap: The Call Stack Runs Synchronous Code First

```
console.log("1");
setTimeout(() => console.log("2"), 0);
console.log("3");
// 1, 3, 2 — NOT 1, 2, 3, even with a 0ms delay
```

All ordinary, synchronous code finishes completely before the engine even considers running any scheduled callback — this is the **call stack** emptying out first. `setTimeout(..., 0)` doesn't mean "run immediately"; it means "run as soon as the call stack is empty and it's this callback's turn," which is always after every line of synchronous code that follows it.

Two Queues, Not One

Once the call stack is empty, JavaScript doesn't pick from a single waiting line — there are two: the **microtask queue** (promise callbacks, `async` function continuations) and the **macrotask queue** (`setTimeout`, `setInterval`, UI events). The rule: **the entire microtask queue is fully drained before even one macrotask runs.**

```
console.log("1: sync");

setTimeout(() => console.log("2: macrotask (setTimeout)"), 0);

Promise.resolve().then(() => console.log("3: microtask (promise)"));

console.log("4: sync");

// 1: sync
// 4: sync
// 3: microtask (promise)
// 2: macrotask (setTimeout)
```

Even though `setTimeout` was scheduled *before* the promise's `.then()`, the microtask (the promise callback) still runs first — because ALL synchronous code runs first, THEN every pending microtask, and only THEN the next macrotask. This ordering is fixed and predictable, not a coincidence of timing.

AWAIT IS BUILT ON EXACTLY THIS MECHANISM

Every `await` inside an `async` function (Fundamentals Chapter 10) schedules the rest of that function as a microtask — which is precisely why an `async` function's code after `await` consistently runs before any pending `setTimeout`, no matter how the code is arranged on the page.

async/await Through the Same Lens

```
async function run() {
  console.log("A: start of run()");
  await null;   // resolves instantly, but STILL yields to the microtask queue
  console.log("C: after await");
}

console.log("start");
run();
console.log("B: end of script");

// start
// A: start of run()
// B: end of script
// C: after await
```

Calling `run()` executes synchronously up to the first `await` — that part runs immediately, no different from a normal function. The moment `await` is hit, everything *after* it becomes a microtask, even though `await null` resolves essentially instantly. This is why `"B"` logs before `"C"`, despite `run()` being called before `"B"`'s own `console.log`.

Why This Matters in Practice

Most real bugs from this come from assuming code "after" an async operation in the source file also runs "after" it in time. A common mistake: updating a UI element inside a `.then()`, then immediately reading that same element's state on the very next line — that read happens before the microtask has had a chance to run, so it sees stale data. Understanding the queue order explains exactly why.

A LONG-RUNNING MICROTASK CHAIN CAN STILL STARVE MACROTASKS

Because the ENTIRE microtask queue must drain before any macrotask runs, a promise chain that keeps scheduling more microtasks (`.then()` returning another promise, repeatedly) can delay `setTimeout` callbacks and even UI rendering indefinitely. This is a real, if uncommon, performance pitfall — macrotasks aren't starved by one slow microtask, but they can be starved by an unbounded chain of them.

QUEUE	EXAMPLES	DRAINED WHEN?
`Call stack`	All synchronous code	Runs first, completely, every time
`Microtask queue`	.then(), catch(), finally(), code after await	Fully emptied before the next macrotask
`Macrotask queue`	setTimeout, setInterval, UI events, fetch's network completion	One at a time, after microtasks are clear

Coding Challenges

CHALLENGE 1

Without running it, write down (or comment) the exact console.log order for this code, then run it to check: console.log("1"); setTimeout(() => console.log("2"), 0); Promise.resolve().then(() => console.log("3")); Promise.resolve().then(() => console.log("4")); console.log("5");

View solution

CHALLENGE 2

Write an async function logSteps() that logs "1", then awaits a Promise.resolve(), then logs "2". Call logSteps(), then immediately log "3" right after the call (not inside it). Predict and then confirm the actual output order.

View solution

CHALLENGE 3

Write code with TWO setTimeout calls (both 0ms, logging "timeout A" and "timeout B" in that order) and ONE promise .then() (logging "promise") scheduled between them. Run it and explain in a comment why the promise callback runs before EITHER timeout, despite being scheduled in the middle.

View solution

CHAPTER 3 QUICK REFERENCE

- **Call stack** — all synchronous code; always finishes completely first
- **Microtask queue** — promise .then()/catch()/finally(), code after await
- **Macrotask queue** — setTimeout, setInterval, UI events
- **Order:** all sync code → entire microtask queue → ONE macrotask → check microtasks again → repeat
- **setTimeout(fn, 0)** means "as soon as possible," NOT "immediately" — sync code and microtasks always go first
- **await always yields** to the microtask queue, even when the awaited value resolves instantly
- **An unbounded chain of microtasks** can delay macrotasks indefinitely — a real, rare performance pitfall
- **Next chapter:** design patterns — module pattern, observer, debounce/throttle

CHAPTER 4 OF 5

Design Patterns

COURSE 3 · CH 4

Design Patterns: Module Pattern, Observer, Debounce/Throttle

Three named, reusable solutions to problems that have already shown up, unnamed, in earlier chapters

A **design pattern** is just a recognised, named solution to a recurring problem — none of the three covered here are new syntax; they're specific ways of combining things already learned (closures, callbacks, timers) to solve specific, common problems cleanly.

The Module Pattern — Closures as Encapsulation

```
const CounterModule = (function() {
  let count = 0;   // private — never exposed directly

  return {
    increment() { count++; return count; },
    reset() { count = 0; }
  };
})();   // IIFE — called immediately, only the returned object survives

console.log(CounterModule.increment());   // 1
console.log(CounterModule.count);         // undefined — no direct access
```

This is exactly Intermediate Chapter 2's `createBankAccount` private-state pattern, with one addition: an **IIFE** (Immediately Invoked Function Expression) — the surrounding `(function() { ... })()` — runs the function the instant it's defined, so `CounterModule` ends up holding the returned object directly, not the function itself. Before native ES modules (Chapter 1) existed, this was the standard way to create a private, self-contained unit of code.

WHY "MODULE PATTERN" IF REAL MODULES NOW EXIST?

Native `import` / `export` (Chapter 1) has mostly replaced this pattern's original purpose — but the underlying technique (closures hiding private state, exposing only a deliberate public interface) still shows up constantly inside individual files, classes, and libraries, regardless of module system.

The Observer Pattern — One-to-Many Notifications

```
class EventEmitter {
  constructor() {
    this.listeners = {};
  }

  on(event, callback) {
    if (!this.listeners[event]) this.listeners[event] = [];
    this.listeners[event].push(callback);
  }

  emit(event, data) {
    (this.listeners[event] || []).forEach(callback => callback(data));
  }
}

const emitter = new EventEmitter();
emitter.on("login", user => console.log(`Welcome, ${user}`));
emitter.on("login", user => console.log(`Logging access for ${user}`));

emitter.emit("login", "Philip");
// Welcome, Philip
// Logging access for Philip
```

The **observer pattern** lets multiple unrelated pieces of code (**observers**) react to something happening, without the thing that happened needing to know who's listening or how many there are. `addEventListener` (Fundamentals Chapter 8) is observer pattern, built into the browser — `EventEmitter` here is the same idea, generalized beyond DOM events to any custom event name.

Debounce — Wait Until Activity Stops

```
function debounce(fn, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => fn(...args), delay);
  };
}

const search = debounce(query => console.log("Searching for:", query), 300);

input.addEventListener("input", () => search(input.value));
// Typing "hello" fires the "input" event 5 times, but search() only RUNS once — 300ms after the last ke
```

`debounce` wraps a function so it only actually runs once activity has genuinely stopped for `delay` milliseconds — each new call cancels the previous pending timer and starts a fresh one. This is the standard fix for Fundamentals Chapter 9's `"input"` event firing on every keystroke when a search box should really only query once typing pauses.

Throttle — Allow at Most Once Every X Milliseconds

```
function throttle(fn, limit) {
  let inCooldown = false;
  return function(...args) {
    if (inCooldown) return;
    fn(...args);
    inCooldown = true;
    setTimeout(() => inCooldown = false, limit);
  };
}

const logScroll = throttle(() => console.log("Scroll position:", window.scrollY), 200);
window.addEventListener("scroll", logScroll);
```

`throttle` guarantees the wrapped function runs AT MOST once per `limit` milliseconds, no matter how many times it's called in that window — unlike debounce, the first call goes through immediately, and the function fires regularly during sustained activity rather than waiting for it to stop entirely. `scroll` events fire dozens of times per second; throttling keeps expensive work (logging, layout calculations) from running that often.

> **DEBOUNCE AND THROTTLE SOLVE DIFFERENT PROBLEMS — DON'T MIX THEM UP**
> Debounce: "wait for quiet, then run once" — right for search-as-you-type. Throttle: "run regularly, but cap the rate" — right for scroll/resize handlers that need periodic updates throughout continuous activity, not just at the end.

PATTERN	PROBLEM IT SOLVES	BUILT FROM
Module pattern	Private state, public interface	Closures + IIFE
Observer	Many listeners reacting to one event, decoupled	Arrays of callbacks + forEach
Debounce	"Run once activity stops" (search-as-you-type)	setTimeout + clearTimeout
Throttle	"Run at most once per interval" (scroll/resize)	setTimeout + a boolean flag

Coding Challenges

CHALLENGE 1

Using the module pattern (IIFE returning an object), build a TodoModule with private state (an array) and a public interface of addTodo(text) and getAll(). Add 3 todos and log the result of getAll(), then confirm TodoModule.todos is undefined.

View solution

CHALLENGE 2

Using the EventEmitter class from this chapter, create an instance and register two separate listeners for an "orderPlaced" event — one logging a confirmation message, one logging that an email should be sent. Emit the event once with an order ID and confirm both listeners run.

View solution

CHALLENGE 3

Using the debounce function from this chapter, wrap a function that logs "Saving..." with a 500ms delay. Call the debounced version 5 times in quick succession (e.g. in a loop with no delay between calls) and explain in a comment why "Saving..." only logs once.

View solution

CHAPTER 4 QUICK REFERENCE

- **Module pattern** — IIFE + closure, returning a deliberate public interface, hiding private state
- **IIFE:** (function() { ... })() — runs immediately, only what it returns survives
- **Observer pattern** — register listeners, emit events; addEventListener is this pattern, built-in
- **Debounce** — runs once, after activity stops for a set delay (search-as-you-type)
- **Throttle** — runs at most once per interval, during continuous activity (scroll/resize)
- **None of these are new syntax** — all three combine closures, timers, and callbacks already covered
- **Next chapter:** working with APIs — pagination, rate limits, caching strategies

CHAPTER 5 OF 5

Working with APIs

COURSE 3 · CH 5

Working with APIs: Pagination, Rate Limits, and Caching

The final chapter — combining everything from fetch through generators into the patterns real production API code actually uses

Fundamentals Chapter 10 covered a single `fetch` call in isolation. Real APIs rarely hand back everything in one response — they paginate, they rate-limit, and repeatedly re-fetching the same unchanged data wastes both time and the API's generosity. This final chapter brings together `fetch`, generators (Chapter 2), debounce (Chapter 4), and closures into the patterns that show up constantly in production code.

Pagination — Fetching Data in Pages

```
async function fetchPage(pageNumber) {
  const response = await fetch(`https://api.example.com/posts?page=${pageNumber}&limit=10`);
  return response.json();   // { data: [...10 items...], hasMore: true }
}

async function fetchAllPages() {
  let page = 1;
  let allPosts = [];
  let hasMore = true;

  while (hasMore) {
    const result = await fetchPage(page);
    allPosts = [...allPosts, ...result.data];
    hasMore = result.hasMore;
    page++;
  }

  return allPosts;
}
```

Most APIs return data in pages to limit response size — each call needs a page number, and a flag (here, `hasMore`) signals whether further pages exist. `while (hasMore)` keeps requesting and merging pages (using spread, Intermediate Chapter 1) until the API says there's nothing left.

A Generator Wrapping Pagination

```
async function* paginate(baseUrl) {
  let page = 1;
  let hasMore = true;

  while (hasMore) {
    const response = await fetch(`${baseUrl}?page=${page}`);
    const result = await response.json();
    yield result.data;   // hand back ONE page at a time, not all at once
    hasMore = result.hasMore;
    page++;
  }
}

for await (const pageOfPosts of paginate("https://api.example.com/posts")) {
  console.log("Got a page with", pageOfPosts.length, "items");
}
```

`async function*` combines Chapter 2's generators with `async` / `await` — each `yield` hands back one page as soon as it's fetched, rather than waiting for every page to load before returning anything. `for await (... of ...)` is `for...of`'s counterpart for these async generators, awaiting each yielded value automatically.

Rate Limiting — Respecting an API's Request Caps

```
function createRateLimiter(maxPerSecond) {
  let queue = [];
  let processing = false;

  async function processQueue() {
    if (processing) return;
    processing = true;
    while (queue.length) {
      const { task, resolve } = queue.shift();
      resolve(await task());
      await new Promise(r => setTimeout(r, 1000 / maxPerSecond));
    }
    processing = false;
  }

  return function limitedRequest(task) {
    return new Promise(resolve => {
      queue.push({ task, resolve });
      processQueue();
    });
  };
}

const limited = createRateLimiter(2);   // max 2 requests per second
for (let i = 1; i <= 5; i++) {
  limited(() => fetch(`https://api.example.com/item/${i}`));
}
```

Many APIs reject requests beyond a certain rate (e.g. "max 2 requests/second"). This rate limiter — using Chapter 4's module pattern, a queue, and a spaced-out `setTimeout` delay between each item — ensures requests are spread out automatically, regardless of how quickly the calling code tries to fire them off.

A 429 RESPONSE MEANS "SLOW DOWN," NOT "BROKEN"

HTTP status `429 Too Many Requests` is the standard signal an API sends when its rate limit is exceeded. Production code should check for it specifically (`response.status === 429`) and back off — retrying immediately just gets rejected again, and repeatedly hitting it can lead to a temporary or permanent IP ban from the API provider.

Caching — Avoiding Repeated, Unnecessary Requests

```
function createCachedFetcher(ttlMs) {
  const cache = new Map();

  return async function cachedFetch(url) {
    const cached = cache.get(url);
    if (cached && Date.now() - cached.timestamp < ttlMs) {
      return cached.data;   // still fresh — skip the network entirely
    }

    const response = await fetch(url);
    const data = await response.json();
    cache.set(url, { data, timestamp: Date.now() });
    return data;
  };
}

const fetchWithCache = createCachedFetcher(60000);   // cache for 60 seconds
```

Another module-pattern closure, this time holding a `Map` (a key/value structure similar to Go's map from the Go course, distinct from a plain object) keyed by URL. `ttlMs` ("time to live") decides how long a cached response stays valid — repeated calls within that window return instantly from memory, with no network request at all.

CONCERN	TOOL/PATTERN
Fetching multiple pages	`while loop checking a hasMore flag`
Streaming pages one at a time	`async function* + for await...of`
Respecting a rate limit	`A queue + spaced setTimeout delays between requests`
Avoiding duplicate requests	`A Map cache with a time-to-live check`

Coding Challenges

CHALLENGE 1

Write an async function fetchAllUsers() using the while-loop pagination pattern from this chapter, against https://jsonplaceholder.typicode.com/users (this particular API returns all results in one page, so simulate hasMore by stopping once the returned array's length is 0 on a fake "page 2"). Log the total count fetched.

View solution

CHALLENGE 2

Write the createCachedFetcher function from this chapter exactly as shown. Create fetchWithCache with a 5000ms TTL, call it twice in a row with the same URL (https://jsonplaceholder.typicode.com/posts/1), and log a message indicating whether each call hit the cache or made a real network request.

View solution

CHALLENGE 3

Write a function that checks a fetch Response object's status and, if it's 429, logs "Rate limited — backing off" and waits 2 seconds (using a Promise + setTimeout) before returning a retry signal; otherwise it returns the response's parsed JSON normally. Test it with a mocked response object of { status: 429 }.

View solution

CHAPTER 5 QUICK REFERENCE

- **Pagination:** loop while a hasMore-style flag is true, merging each page's results
- **async function* + for await...of** — stream paginated results one page at a time
- **429 Too Many Requests** — the standard "you've hit the rate limit" HTTP status
- **Rate limiting:** a queue + spaced delays, ensuring requests never exceed an allowed rate
- **Caching:** a Map keyed by URL, with a timestamp + TTL check before reusing a cached value
- **All of these patterns combine** closures, fetch, generators, and timers — no new core syntax
- **This completes JavaScript Advanced and the JavaScript course overall** as currently planned across all 3 courses.