Module 11 of 16

Functional patterns

Pure functions and immutability with structural sharing; pipe, compose, currying and data-last design; reduce as the universal fold, recursion limits and trampolines; memoization, once and laziness; Maybe, Result and railway-oriented pipelines.

Functional programming in JavaScript is a set of habits rather than a framework: functions that compute only from their inputs, data that is replaced rather than changed, and programs assembled from small functions that compose. The payoff is code you can test with a table, cache without fear, and reason about locally — and JavaScript's first-class functions, closures and array methods make every one of these habits a few lines to adopt.

The module starts with the two disciplines — purity and immutability — and where their boundary lies (a pure core, an impure shell). Composition follows: `pipe` / `compose`, currying and partial application, and the data-last convention that decides whether helpers compose. The fold lesson shows every list operation as `reduce`, the O(n²) trap of spreading the accumulator, and the honest state of recursion in V8 — no tail calls, so loops or trampolines. Memoization and laziness cover the two optimisations purity makes legal, with their failure modes. The last lesson makes absence and failure into values — `Maybe` and `Result` — and shows the railway pattern that turns a chain of fallible steps into a straight line.

The exercises deep-freeze state, implement undo/redo on immutable snapshots, write `curry` and data-last pipelines, rebuild the array toolbox with `reduce`, trampoline a million-step recursion, memoize Fibonacci and measure the calls, build lazy values and a `LazyList`, chain `Maybe`s and `Result`s — then a tiny immutable store with a memoized selector, a pipeline DSL compiled to functions, and a Result-based configuration loader.

CONTENTS

1	Pure functions and immutability — why, and how far to take it	12 min
2	Composition, currying and point-free style — building programs from small functions	13 min
3	Folds, recursion and trampolines — reduce as the universal loop	12 min
4	Memoization and laziness — computing once, and only when asked	12 min

5	Maybe, Result and railway-oriented code — making absence and failure values	13 min
✓	Checkpoint — Functional patterns	24 min

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

LESSON 1 · 12 MIN

Pure functions and immutability — why, and how far to take it

Functional programming in JavaScript is not about avoiding classes or writing everything as arrows; it is two disciplines that make code easier to reason about: functions that **only** compute their result from their inputs (**pure**), and data that is **not modified** after creation (**immutable**). Together they give you code you can test with a table of inputs and outputs, cache freely, run in any order, and debug by looking at values instead of histories. This lesson defines both precisely, shows what they buy, gives the update idioms and their costs, and — importantly — says where the boundary is: programs must have effects, and the skill is keeping them at the edges.

Purity

A function is pure when it (1) returns the same result for the same arguments, every time, and (2) has no observable effect other than returning: no mutation of arguments or outer state, no I/O, no reading of anything that can change (the clock, `Math.random`, a global, a file). `(a, b) ⇒ a + b` is pure. `arr.push(x)`, `Date.now()`, `console.log`, `fetch`, and `let counter; () ⇒ counter++` are not.

What purity buys:

Testability — a pure function is a table: inputs in, outputs out, no mocks, no setup.

Referential transparency — a call can be replaced by its result, so you can cache it (memoisation), skip it when the inputs did not change (React's rendering model), or reorder and parallelise it.

Local reasoning — to understand the function you read the function; nothing elsewhere changes its behaviour.

Impurity is not a sin; it is where the program touches the world. The design goal is a **pure core, impure shell**: parsing, validation, computation and decisions as pure functions; reading, writing, time and randomness pushed to a thin outer layer that calls them. A function that needs "now" takes `now` as a parameter; one that needs a random number takes an RNG; one that would log returns what should be logged. Then the core is testable and the shell is trivial.

Immutability

Immutable data is never changed after creation; an "update" produces a **new** value sharing what did not change. JavaScript's primitives are already immutable; objects and arrays are not, so immutability is a discipline, enforced by convention, by `Object.freeze` in development, or by types (`readonly` in TypeScript).

```
const user = { name: "Ada", tags: ["a"], address: { city: "London" } };
const moved = { ...user, address: { ...user.address, city: "Paris" } };   // r
const tagged = { ...user, tags: [...user.tags, "b"] };                      /
const without = (({ tags, ...rest }) => rest)(user);                       //
const nextList = list.map((x) => (x.id === id ? { ...x, done: true } : x));  /
const removed = list.filter((x) => x.id !== id);
const inserted = [...list.slice(0, i), item, ...list.slice(i)];
const sorted = [...list].sort(cmp);
```

Structural sharing: only the objects on the changed path are new; everything else is the same reference. That makes immutable updates cheap (O(depth), not O(size)) and makes **change detection** trivial — `prev.address === next.address` answers "did the address change" without a deep compare. It is the mechanism behind React/Redux re-rendering, memoised selectors, and undo/redo (keep the old root; it still exists).

Mutating versus non-mutating methods

Mutate the receiver: `push pop shift unshift splice sort reverse fill copyWithin`, and assignment/`delete` on properties. Return new values: `map filter reduce slice concat flat flatMap` + spread, `Object.assign({}, …)`, `Object.entries/fromEntries`. Node 20 added `toSorted/toReversed/toSpliced/with` — on Node 16 copy first (`[...a].sort()`). Mutating methods are fine on data you **own** and have not shared (a local accumulator inside a function); the hazard is mutating something a caller or another module also holds.

Freezing

`Object.freeze` is shallow (module 4). A `deepFreeze` recurses; in strict mode a write to a frozen property throws, which turns "someone mutated my config" from a mystery into a stack trace. Freeze constants, defaults, and fixtures in development; in production, freezing large

structures costs time and is usually dropped (types or discipline do the job). Freezing does not make a *value* immutable — `const` prevents rebinding only.

Costs and limits

Copying has a cost proportional to the changed path's width (an object with 50 keys is copied on every update to one key). For most application state it is negligible; for hot loops over large arrays, a local mutable accumulator that is built and then never mutated again is both faster and still "immutable from the outside". Libraries handle the awkward middle: **Immer** lets you write mutating code against a draft and produces the immutable result with structural sharing; **Immutable.js**/persistent structures give O(log n) updates for very large collections.

Known trap: a shallow copy shares nested objects, so a later "harmless" mutation of `moved.tags.push(...)` changes `user.tags` too. Either never mutate nested values or copy the path you change.

Where effects belong

Sequence a program as: gather inputs (impure) → compute (pure) → apply outputs (impure). A request handler reads the body and the database (effects), calls `decide(order, inventory)` (pure), then writes and responds (effects). The pure function is where the logic and the tests live; the shell should be so thin it needs only an integration test.

Common mistakes

Calling a function pure while it reads a module-level variable that changes, or `Date.now()`.

Mutating an argument (`opts.retries = 3`) — callers' objects change under them.

Spread-copying one level and mutating deeper.

`sort` / `reverse` on shared arrays; `push` into an array received as a parameter.

Freezing everything in production, or believing `const` makes objects immutable.

Over-copying in hot loops where a local mutable accumulator was fine.

Interview angle

"What is a pure function?" Same inputs → same output, no side effects; testable as a table, cacheable, reorderable.

"Why immutability?" Local reasoning, cheap change detection by reference, safe sharing, undo/redo, no action-at-a-distance bugs.

"How do you update nested state immutably?" Spread each level on the changed path; untouched siblings stay shared (structural sharing).

"Isn't copying slow?" O(changed path), not O(size); measure before optimising; Immer/persistent structures for extreme cases.

"How do you handle effects then?" Pure core, impure shell — pass time/randomness in, return what should be written out.

Key takeaways

Pure = deterministic + effect-free; push effects (I/O, time, randomness) to the edges and pass them in as arguments.

Immutable updates spread the changed path; structural sharing makes them cheap and makes `===` a change detector.

Know the mutating methods; copy before them on shared data; mutate only what you own locally.

`Object.freeze` (deep, in development) turns accidental mutation into an error; `const` is not immutability.

Immer and persistent structures for the hard cases; a local mutable accumulator is fine when nothing else sees it.

EXERCISE 1

Deep freeze, then try to break it

Write `deepFreeze(value)` that recursively freezes every nested object and array (children first) and returns the value. Parse the JSON on line 1, deep-freeze it, then run commands in strict mode: `set <path> <json>` assigns through the dotted path, `push <path> <json>` pushes onto an array, `frozen <path>` prints `Object.isFrozen` of that value. Each command prints `<command>: changed`, `<command>: <error constructor name>`, or `<command>: <bool>`; finish with `state=<JSON>` to prove nothing moved.

Example: `{"a":{"b":1,"list":[1,2]},"c":"x"}` then `set c "y"`, `set a.b 5`, `push a.list 3`, `frozen a.list` →

```
set c "y": TypeError
set a.b 5: TypeError
push a.list 3: TypeError
frozen a.list: true
state={"a":{"b":1,"list":[1,2]},"c":"x"}
```

A shallow `Object.freeze` would have let `set a.b` and `push a.list` through.

INPUT 1

```
{"a":{"b":1,"list":[1,2]},"c":"x"}
set c "y"
set a.b 5
push a.list 3
frozen a
frozen a.list
frozen c
```

OUTPUT 1

```
set c "y": TypeError
set a.b 5: TypeError
push a.list 3: TypeError
frozen a: true
frozen a.list: true
frozen c: true
state={"a":{"b":1,"list":[1,2]},"c":"x"}
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const state = JSON.parse(lines[0]);
function deepFreeze(value) {
  if (value !== null && typeof value === "object") {
    for (const inner of Object.values(value)) deepFreeze(inner);   // children first
    Object.freeze(value);
  }
  return value;
}
deepFreeze(state);
const getPath = (obj, path) => path.split(".").reduce((o, k) => o[k], obj);
for (const line of lines.slice(1)) {
  const [cmd, path, ...rest] = line.trim().split(/\s+/);
  try {
    if (cmd === "set") {
      const keys = path.split(".");
      const target = keys.length > 1 ? getPath(state, keys.slice(0, -1).join(".")) :
      target[keys.at(-1)] = JSON.parse(rest.join(" "));
      console.log(`${line.trim()}: changed`);
    } else if (cmd === "push") {
      getPath(state, path).push(JSON.parse(rest.join(" ")));
      console.log(`${line.trim()}: changed`);
    } else if (cmd === "frozen") console.log(`${line.trim()}: ${Object.isFrozen(getP
  } catch (err) {
    console.log(`${line.trim()}: ${err.constructor.name}`);
  }
}
console.log(`state=${JSON.stringify(state)}`);
```

EXERCISE 2

Undo and redo on immutable snapshots

Keep a `history` array of states and a `cursor`. `set <path> <json>` builds the next state with an immutable `setIn` (spreading each object on the path), discards any redo future (`history.splice(cursor + 1)`), pushes it and prints `set <path>: shared=<top-level keys whose values are the same reference as in the previous state, or →`; `undo` / `redo` move the cursor within bounds and print `undo → <cursor>` / `redo → <cursor>`; `show` prints `state=<JSON> history=<length>`.

Example: `{"user":{"name":"ada","age":36},"tags":["x"]}` then `set user.age 37`, `set user.name "Ada"`, `show`, `undo`, `show`, `undo`, `undo`, `show`, `redo`, `show`, `set tags []`, `redo`, `show` →

```
set user.age: shared=tags
set user.name: shared=tags
state={"user":{"name":"Ada","age":37},"tags":["x"]} history=3
undo → 1
state={"user":{"name":"ada","age":37},"tags":["x"]} history=3
undo → 0
undo → 0
state={"user":{"name":"ada","age":36},"tags":["x"]} history=3
redo → 1
state={"user":{"name":"ada","age":37},"tags":["x"]} history=3
set tags: shared=user
redo → 2
state={"user":{"name":"ada","age":37},"tags":[]} history=3
```

Because old states are never mutated, undo is just moving a pointer.

INPUT 1

```
{"user":{"name":"ada","age":36},"tags":
["x"]}
set user.age 37
set user.name "Ada"
show
undo
show
undo
undo
show
redo
show
set tags []
redo
show
```

OUTPUT 1

```
set user.age: shared=tags
set user.name: shared=tags
state={"user":
{"name":"Ada","age":37},"tags":["x"]}
history=3
undo → 1
state={"user":
{"name":"ada","age":37},"tags":["x"]}
history=3
undo → 0
undo → 0
state={"user":
{"name":"ada","age":36},"tags":["x"]}
history=3
redo → 1
state={"user":
{"name":"ada","age":37},"tags":["x"]}
history=3
set tags: shared=user
redo → 2
state={"user":
{"name":"ada","age":37},"tags":[]}
history=3
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
function setIn(obj, keys, value) {
  if (keys.length === 0) return value;
  const [head, ...tail] = keys;
  return { ...obj, [head]: setIn(obj?.[head] ?? {}, tail, value) };
}
const history = [JSON.parse(lines[0])];
let cursor = 0;
const shared = (a, b) => Object.keys(a).filter((k) => a[k] === b[k]).join(",") || "-
for (const line of lines.slice(1)) {
  const [cmd, path, ...rest] = line.trim().split(/\s+/);
  if (cmd === "set") {
    const next = setIn(history[cursor], path.split("."), JSON.parse(rest.join(" ")))
    history.splice(cursor + 1);              // a new edit discards any redo future
    history.push(next);
    cursor++;
    console.log(`set ${path}: shared=${shared(history[cursor - 1], next)}`);
  } else if (cmd === "undo") {
    if (cursor > 0) cursor--;
    console.log(`undo → ${cursor}`);
  } else if (cmd === "redo") {
    if (cursor < history.length - 1) cursor++;
    console.log(`redo → ${cursor}`);
  } else if (cmd === "show") console.log(`state=${JSON.stringify(history[cursor])} h
}
```

QUIZ

1. A function is pure when…

A. It has no parameters

B. Its result depends only on its arguments and it has no observable side effects — no mutation, I/O, time or randomness

C. It returns an object

D. It is an arrow function

2. After `next = { ...prev, address: { ...prev.address, city } }`, `prev.tags === next.tags` is…

A. `false` — everything was copied

B. `true` — only the changed path is new; untouched values are shared (structural sharing), which makes `===` a change detector

C. An error

D. `undefined`

3. Where should effects (I/O, time, randomness) live?

A. Everywhere

B. In a thin outer shell that passes results into a pure core and applies its outputs

C. In constructors

D. Nowhere — programs must be pure

4. `Object.freeze(config)` followed by `config.hosts.push("x")` …

A. Throws

B. Succeeds — freeze is shallow; deep-freeze recursively to lock nested values

C. Freezes the array

D. Is ignored

5. Mutating a local accumulator inside a function that returns a new value is…

A. Impure

B. Fine — nothing outside can observe it; immutability matters at the function's boundary

C. Slow

D. A memory leak

ANSWER KEY · PURE FUNCTIONS AND IMMUTABILITY — WHY, AND HOW FAR TO TAKE IT

1. B Pure functions are testable as tables and safe to cache and reorder.

2. B That is the mechanism behind React/Redux change detection and cheap undo.

3. B A function that needs "now" takes it as a parameter.

4. B In strict mode a write to a frozen object's own property throws; nested objects are separate.

5. B Copy-per-step "to stay immutable" inside a loop is the $O(n^2)$ mistake.

LESSON 2 · 13 MIN

Composition, currying and point-free style — building programs from small functions

Once functions are pure, they compose: the output of one is the input of the next, and a program becomes a pipeline of small, named, testable steps. Two mechanical tools make that pleasant in JavaScript — `pipe` / `compose` to chain unary functions, and **currying**/partial application to turn multi-argument helpers into the unary pieces a pipeline wants — plus one design convention, **data-last**, that decides whether a library composes well. This lesson builds the tools, shows the style they enable (including "point-free" code with no named parameters), compares it honestly with method chaining, and marks the places where the style stops helping.

`pipe` and `compose`

```
const pipe = (...fns) ⇒ (x) ⇒ fns.reduce((acc, f) ⇒ f(acc), x);         // 
const compose = (...fns) ⇒ (x) ⇒ fns.reduceRight((acc, f) ⇒ f(acc), x); //

const slugify = pipe(
  (s) ⇒ s.trim().toLowerCase(),
  (s) ⇒ s.replace(/[^a-z0-9]+/g, "-"),
  (s) ⇒ s.replace(/^⊣-$/g, ""),
);
slugify("  Hello, World! ");   // "hello-world"
```

`pipe` reads in execution order and is what most JavaScript code uses; `compose` matches mathematical notation. Both work only with **unary** functions — each step takes one value — which is why the next two tools exist. A pipeline is a value: store it, pass it, test each step on its own, insert a `tap` (`(x) ⇒ { log(x); return x; }`) to inspect the middle.

Currying and partial application

```
const curry = (fn) => function curried(...args) {
  return args.length >= fn.length ? fn(...args) : (...more) => curried(...args
};
const add = curry((a, b, c) => a + b + c);
add(1)(2)(3); add(1, 2)(3); add(1)(2, 3);   // 6 — all shapes work

const partial = (fn, ...preset) => (...later) => fn(...preset, ...later);
const greetAda = partial(greet, "Hello", "Ada");
```

Currying transforms `f(a, b, c)` into `f(a)(b)(c)` — a chain of unary functions, where each call fixes one argument; **partial application** fixes some arguments now and takes the rest later, in one call. Auto-currying (above) accepts arguments in any grouping by comparing against `fn.length` — which is why it breaks on functions with default or rest parameters (they do not count toward `length`) and on variadic functions; state the arity explicitly (`curryN(3, fn)`) in those cases.

Data-last

```
const map = curry((fn, xs) => xs.map(fn));
const filter = curry((pred, xs) => xs.filter(pred));
const sortBy = curry((key, xs) => [...xs].sort((a, b) => (a[key] < b[key] ? -1
const take = curry((n, xs) => xs.slice(0, n));

const topAdults = pipe(filter((u) => u.age >= 18), sortBy("age"), map((u) => u
topAdults(users);
```

Put the **configuration first and the data last**: `map(fn, xs)` not `map(xs, fn)`. Then partially applying the configuration yields exactly the unary function a pipeline needs, and the same `topAdults` works on any array. Lodash's `_.map(xs, fn)` is data-first (chain-friendly, `lodash/fp` flips it); Ramda and `lodash/fp` are data-last by design. If you write utilities meant to compose, make them data-last and curried.

Point-free style

"Point-free" (tacit) code names no arguments: `const names = map(prop("name"))` instead of `(users) => users.map((u) => u.name)`. It removes noise when the functions have good names — `pipe(trim, toLower, slugify)` — and adds a puzzle when they do not:

`compose(map(compose(add(1), multiply(2))), filter(complement(isNil)))` reads worse than a two-line arrow. Use it for straight pipelines of well-named steps; write the arrow when an argument name would help the reader. `Array.prototype.map(parseInt)` is the cautionary tale: point-free with a function that takes more arguments than you think.

Small combinators worth having

```
const identity = (x) ⇒ x;
const constant = (v) ⇒ () ⇒ v;
const tap = (f) ⇒ (x) ⇒ { f(x); return x; };
const prop = (k) ⇒ (o) ⇒ o[k];
const not = (pred) ⇒ (...a) ⇒ !pred(...a);
const flip = (f) ⇒ (a, b) ⇒ f(b, a);
const unary = (f) ⇒ (x) ⇒ f(x);                  // fixes .map(parseInt)
const juxt = (...fns) ⇒ (x) ⇒ fns.map((f) ⇒ f(x));   // apply several funct
const once = (f) ⇒ { let done = false, r; return (...a) ⇒ (done ? r : ((done
```

Each is five to ten characters of logic and saves an inline arrow at every use.

Pipelines versus method chains

`xs.filter(p).map(f).slice(0, 3)` is already a pipeline — for arrays, chaining is idiomatic and often clearer, and it needs no library. `pipe` wins when the steps are not array methods (string transforms, domain functions, validation), when you want to **reuse** the composed function, when steps come from different modules, or when the data is not an array (a single record, a promise via `pipeP`/async steps). Many codebases use both: method chains inside steps, `pipe` between steps. JavaScript's pipeline operator (`▷`) remains a proposal.

Async composition

`const pipeAsync = (...fns) ⇒ (x) ⇒ fns.reduce((p, f) ⇒ p.then(f), Promise.resolve(x));` — each step may be sync or async; the result is a promise. Error handling is the promise chain's: one `catch` at the end, or a `Result`-returning style (lesson 5).

When to stop

Composition shines for **transformations of data**: parsing, normalising, filtering, reporting. It fights you for stateful, event-driven, or performance-critical code — a game loop, a socket handler, a tight numeric kernel — where a plain loop with mutable locals is clearer and faster. A curried, point-free, five-combinator expression that takes a minute to decode has failed the reader; the test of a functional style is that the *next* developer finds it obvious.

Common mistakes

- Composing non-unary functions; `compose` when `pipe` order was intended.
- Auto-curry on functions with defaults/rest parameters (`length` is wrong) or on variadic functions.
- Data-first helpers in a pipeline (nothing partially applies cleanly).
- Point-free with functions that take extra arguments (`map(parseInt)`).
- Rewriting an obvious `for` loop into a compose tower for style points.
- Forgetting that a pipeline of arrays allocates per step — fine normally, measure on hot paths.

Interview angle

- *"Implement `pipe` and `compose`."* `reduce` / `reduceRight` over the functions with the value as accumulator.
- *"Currying versus partial application?"* Currying turns an n-ary function into a chain of unary calls; partial application fixes some arguments in one call.
- *"Why data-last?"* So partially applying the configuration yields a unary function that composes.
- *"What is point-free style, and when is it bad?"* Code without named arguments; bad when the combinator soup is harder to read than an arrow.
- *"How does `curry` know when to call the function?"* It compares collected arguments to `fn.length` — which fails for defaults, rest and variadics.

Key takeaways

- `pipe` (left→right) and `compose` (right→left) chain unary functions with `reduce`; pipelines are values you can store, reuse and `tap`.

Curry to get unary steps from multi-argument helpers; partial application fixes leading arguments; watch `fn.length`.

Data-last, curried utilities compose; data-first ones chain.

Point-free for clean pipelines of well-named functions; arrows when a parameter name helps.

Use composition for data transformation; loops and mutable locals for stateful or hot code.

EXERCISE 1

curry, partial, pipe and compose

Implement an auto-currying `curry(fn)` (calls `fn` once `fn.length` arguments have arrived, accepting any grouping). With the numbers on line 1 print `curry=<add3(a)(b)(c)>,<add3(a, b)(c)>,<add3(a)(b, c)>,<add3(a, b, c)>`; then `pipe(inc,double)(a)=<> compose(inc,double)(a)=<>`; then `partial=<partial(greet, "Hello", "!")("Ada")> | <partial(greet, "Hi")("?", "Bob")>` for `greet(greeting, punctuation, name)` → `<greeting>, <name><punctuation>`; then `slug=<slugify(line)>` for each remaining line via a `pipe` of trim+lower-case, non-alphanumerics → `-`, strip edge dashes; finally `arityTrap: length=<((x, y = 0) ⇒ x + y).length> curried(5)=<what curry gives for that function called with 5>`.

Example: `1 2 3` / `Hello, World!` →

```
curry=6,6,6,6
pipe(inc,double)(1)=4 compose(inc,double)(1)=3
partial=Hello, Ada! | Hi, Bob?
slug=hello-world
arityTrap: length=1 curried(5)=5
```

A default parameter does not count toward `length`, so the curried function fires after one argument.

INPUT 1

```
1 2 3
  Hello, World!
Already-slug
```

OUTPUT 1

```
curry=6,6,6,6
pipe(inc,double)(1)=4
compose(inc,double)(1)=3
partial=Hello, Ada! | Hi, Bob?
slug=hello-world
slug=already-slug
arityTrap: length=1 curried(5)=5
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const pipe = (...fns) => (x) => fns.reduce((acc, f) => f(acc), x);
const compose = (...fns) => (x) => fns.reduceRight((acc, f) => f(acc), x);
function curry(fn) {
  return function curried(...args) {
    return args.length >= fn.length ? fn(...args) : (...more) => curried(...args, ..
  };
}
const partial = (fn, ...preset) => (...later) => fn(...preset, ...later);
const [a, b, c] = lines[0].trim().split(/\s+/).map(Number);
const add3 = curry((x, y, z) => x + y + z);
console.log(`curry=${add3(a)(b)(c)},${add3(a, b)(c)},${add3(a)(b, c)},${add3(a, b, c
const inc = (x) => x + 1, double = (x) => x * 2;
console.log(`pipe(inc,double)(${a})=${pipe(inc, double)(a)} compose(inc,double)(${a}
const greet = (greeting, punctuation, name) => `${greeting}, ${name}${punctuation}`;
const hello = partial(greet, "Hello", "!");
console.log(`partial=${hello("Ada")} | ${partial(greet, "Hi")("?", "Bob")}`);
const slugify = pipe((s) => s.trim().toLowerCase(), (s) => s.replace(/[^a-z0-9]+/g,
for (const line of lines.slice(1)) console.log(`slug=${slugify(line)}`);
const withDefault = curry((x, y = 0) => x + y);
console.log(`arityTrap: length=${((x, y = 0) => x + y).length} curried(5)=${typeof w
```

EXERCISE 2

A data-last pipeline over records

Write curried, data-last helpers `map(f, xs)`, `filter(p, xs)`, `sortBy(key, xs)` (a sorted copy), `take(k, xs)`, `prop(k)` and `countBy(f, xs)` (a `Map`). Line 1 is `n`; the rest are `name age city`. Build `youngestAdults = pipe(filter(age ≥ 18), sortBy("age"), map(prop("name")), take(n))` and print `youngestAdults=<JSON>`; then `byCity=<city=count sorted by city>` via `countBy(prop("city"))`; then `over30=<names with age > 30 via a pipeline built by a function namesOver(age)> reusedOnSubset=<youngestAdults applied to the first two records>`.

Example: `2` then `ada 36 london`, `bob 17 paris`, `cy 22 london`, `dee 41 rome`, `eve 19 paris` →

```
youngestAdults=["eve","cy"]
byCity=london=2 paris=2 rome=1
over30=["ada","dee"] reusedOnSubset=["ada"]
```

The same composed function ran on two different arrays — that is what data-last buys.

INPUT 1

```
2
ada 36 london
bob 17 paris
cy 22 london
dee 41 rome
eve 19 paris
```

OUTPUT 1

```
youngestAdults=["eve","cy"]
byCity=london=2 paris=2 rome=1
over30=["ada","dee"] reusedOnSubset=
["ada"]
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const curry = (fn) => function c(...a) { return a.length >= fn.length ? fn(...a) : (
const pipe = (...fns) => (x) => fns.reduce((acc, f) => f(acc), x);
const map = curry((f, xs) => xs.map(f));
const filter = curry((p, xs) => xs.filter(p));
const sortBy = curry((key, xs) => [...xs].sort((p, q) => (p[key] < q[key] ? -1 : p[k
const take = curry((k, xs) => xs.slice(0, k));
const prop = (k) => (o) => o[k];
const countBy = curry((f, xs) => xs.reduce((m, x) => m.set(f(x), (m.get(f(x)) ?? 0)
const n = Number(lines[0]);
const people = lines.slice(1).map((l) => { const [name, age, city] = l.trim().split(
const youngestAdults = pipe(filter((p) => p.age >= 18), sortBy("age"), map(prop("nam
console.log(`youngestAdults=${JSON.stringify(youngestAdults(people))}`);
const byCity = pipe(countBy(prop("city")), (m) => [...m].sort(([x], [y]) => x.locale
console.log(`byCity=${byCity(people)}`);
const namesOver = (age) => pipe(filter((p) => p.age > age), map(prop("name")));   //
console.log(`over30=${JSON.stringify(namesOver(30)(people))} reusedOnSubset=${JSON.s
```

QUIZ

1. `pipe(f, g)(x)` equals…

A. `f(g(x))`

B. `g(f(x))` — left to right; `compose(f, g)(x)` is `f(g(x))`

C. `f(x) + g(x)`

D. `[f(x), g(x)]`

2. Auto-currying by `fn.length` fails for…

A. Arrow functions

B. Functions with default or rest parameters (they do not count) and variadic functions — state the arity explicitly

C. Methods

D. Async functions

3. Data-last means…

A. Returning the data

B. Putting configuration parameters first and the collection last, so partial application yields a unary function that composes

C. Sorting descending

D. Passing arrays only

4. Currying versus partial application:

A. Synonyms

B. Currying turns `f(a, b, c)` into `f(a)(b)(c)`; partial application fixes some arguments in one call and returns a function for the rest

C. Partial application is for arrays

D. Currying needs classes

5. Point-free style becomes a problem when…

A. Functions have names

B. The combinator expression is harder to read than an arrow with a named parameter — or a function takes more arguments than expected (`map(parseInt)`)

C. Used with `pipe`

D. Never

ANSWER KEY · COMPOSITION, CURRYING AND POINT-FREE STYLE — BUILDING PROGRAMS FROM SMALL FUNCTIONS

1. B Both are folds over the function list with the value as accumulator.

2. B `((x, y = 0) ⇒ …).length` is 1.

3. B Lodash is data-first (chains well); Ramda and `lodash/fp` are data-last (compose well).

4. B `partial(greet, "Hello")` versus `curry(greet)("Hello")` — same idea, different mechanics.

5. B The test of a style is whether the next developer finds it obvious.

LESSON 3 · 12 MIN

Folds, recursion and trampolines — reduce as the universal loop

Every list operation is a **fold**: walk the elements, carry an accumulator, produce a result. `reduce` is the fold; `map`, `filter`, `flatMap`, `groupBy`, `partition`, `zip`, `unique` and `sum` are all `reduce` with a particular accumulator, and being able to write each of them in one line is the difference between using the array methods and understanding them. Recursion is the other universal loop, and JavaScript has one hard limit on it — no tail-call elimination in V8 — which this lesson addresses with the two standard escapes: rewrite as a loop, or trampoline. It closes with the honest comparison: when a fold clarifies, when a loop is simply better, and how to avoid the O(n²) trap that lurks in "elegant" immutable reduces.

Everything is a fold

```
const map = (f, xs) ⇒ xs.reduce((acc, x) ⇒ (acc.push(f(x)), acc), []);
const filter = (p, xs) ⇒ xs.reduce((acc, x) ⇒ (p(x) ? acc.push(x) : 0, acc),
const flatMap = (f, xs) ⇒ xs.reduce((acc, x) ⇒ acc.concat(f(x)), []);
const groupBy = (key, xs) ⇒ xs.reduce((acc, x) ⇒ { (acc[key(x)] ??= []).push
const partition = (p, xs) ⇒ xs.reduce(([yes, no], x) ⇒ (p(x) ? yes.push(x)
const zip = (xs, ys) ⇒ xs.reduce((acc, x, i) ⇒ (i < ys.length ? (acc.push([x
const unique = (xs) ⇒ xs.reduce((acc, x) ⇒ (acc.includes(x) ? acc : (acc.pus
const countBy = (key, xs) ⇒ xs.reduce((m, x) ⇒ m.set(key(x), (m.get(key(x))
const indexBy = (key, xs) ⇒ new Map(xs.map((x) ⇒ [key(x), x]));
const sum = (xs) ⇒ xs.reduce((a, b) ⇒ a + b, 0);
const maxBy = (f, xs) ⇒ xs.reduce((best, x) ⇒ (best === undefined || f(x) >
const pipeline = (fns, x) ⇒ fns.reduce((acc, f) ⇒ f(acc), x);            //
```

Two things to notice. The accumulator is **mutated** in most of these (`acc.push`, `m.set`) — that is fine and fast because the accumulator was created by this very `reduce` and nobody else sees it until it is returned; the *function* stays pure. And each has a **seed**; without one `reduce` uses the first element and throws on an empty array.

The O(n²) trap

```
xs.reduce((acc, x) ⇒ [...acc, f(x)], []);              // copies the whole ac
xs.reduce((acc, x) ⇒ ({ ...acc, [x.id]: x }), {});      // same
```

Spreading the accumulator "to stay immutable" allocates n arrays of growing size — 10 000 items become 50 million element copies. Mutate the local accumulator (`acc.push`) or use `Object.fromEntries(xs.map(...))` / `new Map(...)`. Immutability matters at the boundary of a function, not inside its own scratch space.

`reduceRight`, and folds over other shapes

`reduceRight` folds from the end — used for `compose`, for building a right-nested structure, or for processing a list in reverse without copying. Folds are not only for arrays: a tree fold visits nodes recursively with an accumulator; a Map/Set can be folded with `for…of` or `[...map].reduce`; a generator can be folded lazily by a consuming loop.

Recursion

```
const depth = (node) ⇒ (node.children.length ? 1 + Math.max(...node.children
const flatten = (xs) ⇒ xs.flatMap((x) ⇒ (Array.isArray(x) ? flatten(x) : [x]
const permutations = (xs) ⇒ (xs.length ⩽ 1 ? [xs] : xs.flatMap((x, i) ⇒ per
```

Recursion is the natural fit for **recursive data** — trees, nested arrays, JSON, expression grammars — and for divide-and-conquer. Every recursive function has a base case (the smallest input, answered directly) and a recursive case (a smaller input plus one step). Write the base case first.

The stack limit and the missing tail calls

Each call uses a stack frame; V8's default stack holds roughly ten thousand frames before `RangeError: Maximum call stack size exceeded`. Many languages eliminate frames for **tail calls** — a call in return position needs no frame to return to — and ES2015 specified it, but V8 (and every other major engine except Safari) never shipped it. So `const sum = (n, acc = 0) ⇒ n ⩵ 0 ? acc : sum(n - 1, acc + n)` overflows at large `n` in Node regardless of being tail-recursive.

Two escapes:

Rewrite as a loop. Any tail-recursive function converts mechanically: the parameters become variables, the recursive call becomes reassignment inside `while`. This is the right answer for linear recursion over big inputs.

Trampoline. Return a *thunk* (a zero-argument function describing the next call) instead of calling; a driver loop keeps calling until a non-function comes back. The recursion becomes iteration without changing the function's shape:

```
const trampoline = (f) => (...args) => {
  let result = f(...args);
  while (typeof result === "function") result = result();    // each bounce is
  return result;
};
const sumTo = trampoline(function go(n, acc = 0) {
  return n === 0 ? acc : () => go(n - 1, acc + n);           // return the NE
});
sumTo(1_000_000);                                            // 500000500000,
```

Trampolines cost an allocation per step and only handle tail calls; for tree recursion (two recursive calls whose results combine) use an explicit stack or queue instead. Depth-first over a tree with an explicit stack array is the standard iterative form.

Fold versus loop, honestly

A `reduce` is right when the operation *is* a fold that reads as one idea: sum, max, group, index, pipe. It is wrong — for readers — when the callback grows past four or five lines, mixes several accumulators, or needs early exit (`reduce` cannot `break`; `for…of` can, and so can `some` / `find`). Performance: a well-written `for` loop is faster than `reduce` by a small constant; nested spreads in a `reduce` are catastrophically slower. Choose by readability first, then measure.

Common mistakes

`reduce` without a seed on possibly-empty input.

`[...acc, x]` / `{ ...acc, k }` inside `reduce` — $O(n^2)$.

Recursion on large linear inputs without a loop or trampoline; expecting tail-call optimisation in Node.

`unique` via `includes` (O(n²)) instead of a `Set`.

Forcing early-exit logic into `reduce` with a flag instead of using a loop, `some` or `find`.

A recursive function whose base case is unreachable for some inputs (negative numbers, empty arrays).

Interview angle

"Implement `map` using `reduce`." `xs.reduce((acc, x) ⇒ (acc.push(f(x)), acc), [])`.

"Why is `reduce((acc, x) ⇒ [...acc, x], [])` slow?" Copies the accumulator each step — O(n²).

"Does JavaScript have tail-call optimisation?" Specified in ES2015, but V8 does not implement it; use a loop or a trampoline.

"What is a trampoline?" A driver that repeatedly calls returned thunks, turning tail recursion into iteration with constant stack.

"When would you not use `reduce`?" Long callbacks, several accumulators, or early exit — a loop reads better and can `break`.

Key takeaways

`map`, `filter`, `flatMap`, `groupBy`, `partition`, `zip`, `countBy`, `pipe` — all folds over `reduce` with a seed; mutate the local accumulator.

Never spread the accumulator inside `reduce`; `Set` / `Map` for uniqueness and lookup.

Recursion for recursive data; base case first; V8 has ~10k frames and no tail calls.

Linear recursion → loop; tail recursion → trampoline (thunks + driver); tree recursion → explicit stack.

Fold when it reads as one idea; loop when it needs `break`, several accumulators or speed.

EXERCISE 1

The array toolbox, rebuilt with reduce

Using **only** `reduce` inside each (no `map` / `filter` calls), implement `map(f, xs)`, `filter(p, xs)`, `flatMap(f, xs)`, `groupBy(key, xs)`, `partition(p, xs)`, `zip(xs, ys)`, `unique(xs)` (via a `Set` accumulator) and `countBy(key, xs)` (a `Map`). Over the input numbers print, one per line: `map=<squares>`, `filter=<evens>`, `flatMap=<[x, -x] for positives>`, `groupBy=<odd/even>`, `partition=<[≥5, <5]>`, `zip=<with ["a","b","c"]>`, `unique=<>`, `countBy=<entries of x % 3 sorted by key>`, `sum=<> max=<>`.

Example: `3 8 0 12 5 8 -1` →

```
map=[9,64,0,144,25,64,1]
filter=[8,0,12,8]
flatMap=[3,-3,8,-8,12,-12,5,-5,8,-8]
groupBy={"odd":[3,5,-1],"even":[8,0,12,8]}
partition=[[8,12,5,8],[3,0,-1]]
zip=[[3,"a"],[8,"b"],[0,"c"]]
unique=[3,8,0,12,5,-1]
countBy=[[-1,1],[0,3],[2,3]]
sum=35 max=12
```

INPUT 1

```
3 8 0 12 5 8 -1
```

OUTPUT 1

```
map=[9,64,0,144,25,64,1]
filter=[8,0,12,8]
flatMap=[3,-3,8,-8,12,-12,5,-5,8,-8]
groupBy={"odd":[3,5,-1],"even":
[8,0,12,8]}
partition=[[8,12,5,8],[3,0,-1]]
zip=[[3,"a"],[8,"b"],[0,"c"]]
unique=[3,8,0,12,5,-1]
countBy=[[-1,1],[0,3],[2,3]]
sum=35 max=12
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const map = (f, xs) => xs.reduce((acc, x) => (acc.push(f(x)), acc), []);
const filter = (p, xs) => xs.reduce((acc, x) => (p(x) && acc.push(x), acc), []);
const flatMap = (f, xs) => xs.reduce((acc, x) => (acc.push(...f(x)), acc), []);
const groupBy = (key, xs) => xs.reduce((acc, x) => { (acc[key(x)] ??= []).push(x); r
const partition = (p, xs) => xs.reduce(([yes, no], x) => (p(x) ? yes.push(x) : no.pu
const zip = (xs, ys) => xs.reduce((acc, x, i) => (i < ys.length && acc.push([x, ys[i
const unique = (xs) => [...xs.reduce((set, x) => set.add(x), new Set())];
const countBy = (key, xs) => xs.reduce((m, x) => m.set(key(x), (m.get(key(x)) ?? 0)
const nums = lines[0].trim().split(/\s+/).map(Number);
console.log(`map=${JSON.stringify(map((x) => x * x, nums))}`);
console.log(`filter=${JSON.stringify(filter((x) => x % 2 === 0, nums))}`);
console.log(`flatMap=${JSON.stringify(flatMap((x) => (x > 0 ? [x, -x] : []), nums))}
console.log(`groupBy=${JSON.stringify(groupBy((x) => (x % 2 ? "odd" : "even"), nums)
console.log(`partition=${JSON.stringify(partition((x) => x >= 5, nums))}`);
console.log(`zip=${JSON.stringify(zip(nums, ["a", "b", "c"]))}`);
console.log(`unique=${JSON.stringify(unique(nums))}`);
console.log(`countBy=${JSON.stringify([...countBy((x) => x % 3, nums)].sort(([a], [b
console.log(`sum=${nums.reduce((a, b) => a + b, 0)} max=${nums.reduce((m, x) => Math
```

EXERCISE 2

A trampoline for deep recursion

Write `trampoline(f)`: returns a function that calls `f` and, while the result is a function, keeps calling it. Using it, write a tail-recursive `sumTo(n)` whose recursive step returns a thunk instead of recursing, a plain-loop `sumLoop(n)`, and mutually recursive `isEven` / `isOdd` via thunks. Print `sumTo=<> sumLoop=<> equal=<>`; `isEven(n)=<> isEven(n+1)=<>`; `bounces for <n>: <thunks executed by a counting version> result=<its return for 0>`; and `plainRecursion(2000)=<the naive recursive sum for 2000> (small depths are fine without a trampoline)`.

Example: `100000` →

```
sumTo=5000050000 sumLoop=5000050000 equal=true
isEven(100000)=true isEven(100001)=false
bounces for 100000: 100001 result=done
plainRecursion(2000)=2001000 (small depths are fine without a trampoline)
```

Plain recursion to 100000 would overflow the stack; the trampolined version uses constant stack.

INPUT 1

```
100000
```

OUTPUT 1

```
sumTo=5000050000 sumLoop=5000050000
equal=true
isEven(100000)=true isEven(100001)=false
bounces for 100000: 100001 result=done
plainRecursion(2000)=2001000 (small
depths are fine without a trampoline)
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const n = Number(lines[0]);
function trampoline(f) {
  return (...args) => {
    let result = f(...args);
    while (typeof result === "function") result = result();     // each bounce is a
    return result;
  };
}
const sumTo = trampoline(function go(k, acc = 0) { return k === 0 ? acc : () => go(k
const sumLoop = (k) => { let acc = 0; while (k > 0) acc += k--; return acc; };
const isEven = trampoline(function even(k) { return k === 0 ? true : () => odd(k - 1
function odd(k) { return k === 0 ? false : () => even(k - 1); }
function even(k) { return k === 0 ? true : () => odd(k - 1); }
let bounces = 0;
const counted = trampoline(function go(k) { bounces++; return k === 0 ? "done" : ()
const t = sumTo(n), l = sumLoop(n);
console.log(`sumTo=${t} sumLoop=${l} equal=${t === l}`);
console.log(`isEven(${n})=${isEven(n)} isEven(${n + 1})=${isEven(n + 1)}`);
console.log(`bounces for ${n}: ${(counted(n), bounces)} result=${counted(0)}`);
const plain = (k, acc = 0) => (k === 0 ? acc : plain(k - 1, acc + k));
console.log(`plainRecursion(2000)=${plain(2000)} (small depths are fine without a tr
```

QUIZ

1. `xs.reduce((acc, x) => [...acc, f(x)], [])` is…

A. The idiomatic `map`

B. O(n²) — the accumulator is copied on every step; push into the local accumulator instead

C. Faster than `map`

D. Lazy

2. Does V8 perform tail-call optimisation?

A. Yes, since ES2015

B. No — it was specified but never shipped; deep tail recursion overflows the stack, so use a loop or a trampoline

C. Only in strict mode

D. Only for arrow functions

3. A trampoline works by…

A. Increasing the stack size

B. Having the recursive function return a thunk for the next step while a driver loop calls thunks until a value comes back — constant stack depth

C. Caching results

D. Running in a worker

4. `reduce` without an initial value on `[]` …

A. Returns `undefined`

B. Throws a `TypeError` — always pass the seed

C. Returns `0`

D. Returns `[]`

5. You need to stop iterating as soon as a condition holds. Use…

A. `reduce` with a flag

B. `for…of` with `break`, or `some` / `find` — `reduce` cannot exit early

C. `forEach`

D. `map`

ANSWER KEY · FOLDS, RECURSION AND TRAMPOLINES — REDUCE AS THE UNIVERSAL LOOP

1. B Ten thousand items become fifty million element copies.

2. B Safari is the one major engine that implemented it.

3. B It handles tail calls only; tree recursion needs an explicit stack.

4. B The seed also makes the accumulator's type explicit.

5. B Choose the construct by readability: folds for one-idea folds, loops for early exit and several accumulators.

LESSON 4 · 12 MIN

Memoization and laziness — computing once, and only when asked

Purity makes two optimisations legal that impure code cannot have: **memoization** — remember a function's result for given arguments and return it next time — and **laziness** — do not compute a value until something needs it, then perhaps never. Both are old ideas; JavaScript makes them small: a `Map` and a wrapper function, a thunk or a generator. Both also have failure modes that look like success (a cache that grows forever, a key function that collides, a "lazy" pipeline that is forced immediately). This lesson covers the generic `memoize` and its knobs, memoized recursion as dynamic programming, `once`, lazy values and sequences, and the rules for when caching costs more than it saves.

`memoize`

```
function memoize(fn, { key = (...args) ⇒ JSON.stringify(args), cache = new Ma
  const wrapped = (...args) ⇒ {
    const k = key(...args);
    if (cache.has(k)) { wrapped.hits++; return cache.get(k); }
    wrapped.misses++;
    const value = fn(...args);
    cache.set(k, value);
    return value;
  };
  wrapped.hits = 0; wrapped.misses = 0; wrapped.cache = cache;
  return wrapped;
}
```

Three design points hide in those lines:

The key. Arguments must become a Map key. Primitives: use the value itself (`key: (x) ⇒ x`) — fastest and exact. Several primitives or plain data: `JSON.stringify(args)` works but is slow-ish, and collides for values JSON cannot distinguish (`undefined` vs missing, `NaN`, `-0`, key order in objects). Objects by identity: use the object as the key in a `WeakMap` (module 6) so the cache does not pin them. Never key on `String(obj)` (`[object Object]` for everything).

The store. A `Map` grows forever — a memoized function called with a million distinct inputs holds a million entries. Bound it: an **LRU** (module 6's checkpoint) or a TTL; or accept unbounded growth only for functions with a small finite domain.

What is cached. Only results that are stable: pure functions, or impure ones whose results you deliberately want frozen for the process lifetime (a config parse). Caching a promise (`memoize(fetchUser)`) dedupes concurrent calls — but a rejected promise stays cached too unless you evict it in a `catch`.

Memoized recursion is dynamic programming

```
const fib = memoize((n) => (n < 2 ? n : fib(n - 1) + fib(n - 2)), { key: (n) =
fib(50);        // 12586269025 — 99 calls instead of ~40 billion
```

The naive recursion recomputes the same subproblems exponentially many times; the memo turns the call tree into a table filled once per distinct argument — exactly what a dynamic-programming solution does with an explicit array. Any recursion with **overlapping subproblems** and a **small argument space** (grid paths, edit distance, coin change, longest common subsequence) benefits. The recursive call must go through the *memoized* name (`fib`, not the inner function), or the inner calls bypass the cache. The recursion depth still counts against the stack; for very large `n` fill a table iteratively.

`once`

```
const once = (fn) => { let called = false, result; return (...args) => (calle
const getConfig = once(() => JSON.parse(fs.readFileSync("config.json", "utf8")
```

Memoization with a single slot and no key — for initialisation that must run exactly once (reading config, creating a connection pool, registering handlers). The lazy variant of a module-level constant: the work happens on the first call, not at import time, so importing the module stays cheap and side-effect-free.

Laziness: thunks and lazy values

A **thunk** is a zero-argument function standing for a computation not yet performed: `const value = () ⇒ expensive()`. Pass thunks where a value might not be needed — a default that is expensive to build (`opts.logger ?? makeLogger()` computes the logger even when `opts.logger` exists; `opts.logger ?? makeLogger` with a call at the use site does not), a message that is only formatted when the log level is enabled, the branches of a custom control-flow function. Combine with `once` for a **lazy value**: computed on first read, cached thereafter — a getter that does `this._v ??= compute()` is the same idea inside a class.

Lazy sequences

Generators (module 6) are lazy sequences: `take(5, map(f, filter(p, naturals())))` computes exactly five results. The pattern generalises: a `LazyList` object with `map` / `filter` / `take` methods that each return another lazy object and a terminal `toArray()` / `forEach()` that pulls — the shape of Java streams, Rust iterators and Node 22's iterator helpers. Laziness pays when the source is infinite or expensive per element, when only a prefix is consumed, or when intermediate arrays would be large. It costs a closure or generator step per element; on a 20-element array, `arr.filter(p).map(f)` is faster and clearer.

Forcing, and the trap

Laziness is only useful if something *stays* unevaluated. `[...lazySeq]`, `Array.from`, `Promise.all`, `JSON.stringify` and a debugger's inspection all **force** the whole thing. A lazy pipeline over an infinite source that is spread by accident hangs; a lazy value whose thunk captures a variable that later changes computes with the later value (a closure-over-loop-variable style bug). Make forcing explicit — one `toArray()` or `take(n)` at the end — and keep thunks pure.

When caching costs more than it saves

The function is already cheap (a property read, a small arithmetic) — key computation and Map lookup cost more.

Arguments are rarely repeated (unique ids, timestamps) — every call is a miss plus a store.

Results are large and inputs are many — memory balloons; bound the cache.

The function is impure and its inputs do not capture what it depends on — the cache returns stale data (memoizing `getUser(id)` when users change).

Object arguments keyed by `JSON.stringify` — slow and collision-prone; key by identity or by a chosen field.

Measure: hit rate and time saved against memory held. React's `useMemo` / `memo` are memoization with a one-slot cache keyed by dependency identity — the same trade-offs apply.

Common mistakes

Unbounded caches in long-running processes.

Keys that collide (`String(obj)`, `JSON.stringify` with key-order differences) or that are slower than the function.

Recursive calls that bypass the memoized wrapper.

Caching rejected promises; caching results of impure functions without an invalidation story.

Spreading a lazy or infinite sequence; thunks capturing mutable variables.

Memoizing everything by reflex.

Interview angle

"Implement `memoize`." A wrapper with a `Map` from key to result; return the cached value on a hit; discuss the key function and bounding the cache.

"How does memoization relate to dynamic programming?" Memoized recursion fills the DP table on demand; same complexity, different order.

"How do you memoize a function taking objects?" Key by identity with a `WeakMap`, or by a stable id — not `JSON.stringify`.

"What is a thunk?" A zero-argument function representing a deferred computation; the unit of laziness.

"When should you not memoize?" Cheap functions, rarely repeated arguments, large results, impure functions, unbounded input domains.

Key takeaways

- `memoize` = key function + cache + wrapper; primitives keyed directly, objects by `WeakMap` identity; bound the cache (LRU/TTL).
- Memoized recursion is dynamic programming; recurse through the memoized name; watch stack depth.
- `once` for run-exactly-once initialisation; thunks for deferred computation; lazy values combine both.
- Generators give lazy sequences; force explicitly with `take` / `toArray`; never spread the infinite.
- Cache only pure (or deliberately frozen) results with repeated inputs; measure hit rate versus memory.

EXERCISE 1

memoize, and Fibonacci as dynamic programming

Write `memoize(fn, key = (...args) ⇒ JSON.stringify(args))` with a `Map` cache and `hits` / `misses` counters on the returned function. Count calls of a naive recursive `fibNaive(n)` and of a memoized `fib` (keyed by `n`, recursing through the memoized name) and print `fib(n)=<> naiveCalls=<> memoCalls=<> hits=<> misses=<>`; call `fib(n)` again and print `secondCall: memoCalls=<> hits=<>`; then a memoized two-argument function called with `(1,2)`, `(1,2)`, `(2,1)` → `jsonKey: hits=1 misses=2 keys=["[1,2]","[2,1]"]`; finally cache `area(rect)` by object identity in a `WeakMap` for two objects with equal contents → `identity: 6,6,6 areaCalls=2 (equal contents, two objects)`.

Example: `20` →

```
fib(20)=6765 naiveCalls=21891 memoCalls=21 hits=18 misses=21
secondCall: memoCalls=21 hits=19
jsonKey: hits=1 misses=2 keys=["[1,2]","[2,1]"]
identity: 6,6,6 areaCalls=2 (equal contents, two objects)
```

INPUT 1

```
20
```

OUTPUT 1

```
fib(20)=6765 naiveCalls=21891
memoCalls=21 hits=18 misses=21
secondCall: memoCalls=21 hits=19
jsonKey: hits=1 misses=2 keys=["[1,2]","
[2,1]"]
identity: 6,6,6 areaCalls=2 (equal
contents, two objects)
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
function memoize(fn, key = (...args) => JSON.stringify(args)) {
  const cache = new Map();
  const wrapped = (...args) => {
    const k = key(...args);
    if (cache.has(k)) { wrapped.hits++; return cache.get(k); }
    wrapped.misses++;
    const value = fn(...args);
    cache.set(k, value);
    return value;
  };
  wrapped.hits = 0; wrapped.misses = 0; wrapped.cache = cache;
  return wrapped;
}
const n = Number(lines[0]);
let naiveCalls = 0;
const fibNaive = (k) => { naiveCalls++; return k < 2 ? k : fibNaive(k - 1) + fibNaiv
let memoCalls = 0;
const fib = memoize((k) => { memoCalls++; return k < 2 ? k : fib(k - 1) + fib(k - 2)
console.log(`fib(${n})=${fib(n)} naiveCalls=${(fibNaive(n), naiveCalls)} memoCalls=$
fib(n);
console.log(`secondCall: memoCalls=${memoCalls} hits=${fib.hits}`);
const slowKeys = memoize((a, b) => `${a}|${b}`);
slowKeys(1, 2); slowKeys(1, 2); slowKeys(2, 1);
console.log(`jsonKey: hits=${slowKeys.hits} misses=${slowKeys.misses} keys=${JSON.st
const byIdentity = new WeakMap();
let areaCalls = 0;
const area = (rect) => { if (!byIdentity.has(rect)) { areaCalls++; byIdentity.set(re
const r1 = { w: 2, h: 3 }, r2 = { w: 2, h: 3 };
console.log(`identity: ${area(r1)},${area(r1)},${area(r2)} areaCalls=${areaCalls} (e
```

EXERCISE 2

once, thunks and a LazyList

The input is a number of reads. Print `builds: eager=<> defaultParam=<> thunk=<>` counting how many times `buildLogger` ran when a logger **was** supplied for three fallback styles: `opts.logger ?? buildLogger()`, a default parameter `fallback = buildLogger()`, and a thunk default `fallback = buildLogger` called only on the fallback path. Then `once`: wrap a config builder, call it `reads` times, print `once: reads=<> builds=1 sameObject=true`. Then a lazy getter that computes once and replaces itself → `lazyGetter: reads=<> computed=1 total=42`. Finally a `LazyList` over the naturals with `filter` / `map` / `take` / `toArray` built on generators, counting how many naturals were produced: print `beforeForce=0` and `first <reads>: <squares of the first reads odd numbers> evaluated=<count>`.

Example: `4` →

```
builds: eager=0 defaultParam=1 thunk=0
once: reads=4 builds=1 sameObject=true
lazyGetter: reads=4 computed=1 total=42
beforeForce=0
first 4: [1,9,25,49] evaluated=7
```

Only seven naturals were generated for four results, and nothing ran until `toArray`.

INPUT 1

```
4
```

OUTPUT 1

```
builds: eager=0 defaultParam=1 thunk=0
once: reads=4 builds=1 sameObject=true
lazyGetter: reads=4 computed=1 total=42
beforeForce=0
first 4: [1,9,25,49] evaluated=7
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const reads = Number(lines[0]);
let builds = 0;
const buildLogger = () => { builds++; return { level: "info" }; };
const once = (fn) => { let done = false, result; return (...a) => (done ? result : (
const eager = (opts) => opts.logger ?? buildLogger();                          // ??
const eagerArg = (opts, fallback = buildLogger()) => opts.logger ?? fallback;   // t
const lazy = (opts, fallback = buildLogger) => opts.logger ?? fallback();       // p
const supplied = { logger: { level: "debug" } };
builds = 0; eager(supplied); const b1 = builds;
builds = 0; eagerArg(supplied); const b2 = builds;
builds = 0; lazy(supplied); const b3 = builds;
console.log(`builds: eager=${b1} defaultParam=${b2} thunk=${b3}`);
const getConfig = once(() => { builds++; return { port: 3000 }; });
builds = 0;
for (let i = 0; i < reads; i++) getConfig();
console.log(`once: reads=${reads} builds=${builds} sameObject=${getConfig() === getC
let computed = 0;
const report = { get total() { computed++; return Object.defineProperty(this, "total
for (let i = 0; i < reads; i++) report.total;
console.log(`lazyGetter: reads=${reads} computed=${computed} total=${report.total}`)
let evaluated = 0;
class LazyList {
  constructor(gen) { this.gen = gen; }
  static naturals() { return new LazyList(function* () { let i = 1; while (true) { e
  map(f) { const src = this.gen; return new LazyList(function* () { for (const x of
  filter(p) { const src = this.gen; return new LazyList(function* () { for (const x
  take(k) { const src = this.gen; return new LazyList(function* () { if (k <= 0) ret
  toArray() { return [...this.gen()]; }
}
const squaresOfOdds = LazyList.naturals().filter((x) => x % 2 === 1).map((x) => x *
console.log(`beforeForce=${evaluated}`);
console.log(`first ${reads}: ${JSON.stringify(squaresOfOdds.take(reads).toArray())}
```

QUIZ

1. Memoizing a function that takes object arguments should key by…

A. `String(obj)`

B. Object identity in a `WeakMap` (or a stable id field) — not `JSON.stringify`, which is slow and collides on key order

C. `obj.length`

D. The call count

2. A memoized function called with a million distinct inputs in a long-running server…

A. Is fine

B. Holds a million cache entries forever — bound the cache (LRU/TTL) or do not memoize

C. Speeds up

D. Evicts automatically

3. In `const fib = memoize((n) ⇒ n < 2 ? n : inner(n - 1) + inner(n - 2))` where `inner` is the raw function…

A. Everything is cached

B. The recursive calls bypass the cache — recurse through the memoized `fib`

C. It throws

D. It is faster

4. A thunk is…

A. A cached value

B. A zero-argument function standing for a computation not yet performed — the unit of laziness

C. A promise

D. A generator

5. `[...lazySequence]` on a lazy pipeline over an infinite source…

A. Takes the first few

B. Forces the whole sequence and never terminates — force explicitly with `take(n)` first

C. Throws

D. Returns `[]`

ANSWER KEY · MEMOIZATION AND LAZINESS — COMPUTING ONCE, AND ONLY WHEN ASKED

1. B `String(obj)` is `[object Object]` for everything.

2. B Memoize functions with repeated inputs and a bounded domain.

3. B Memoized recursion is dynamic programming only if the subcalls hit the table.

4. B `once` + a thunk gives a lazy value: computed on first use, cached after.

5. B `Array.from`, `Promise.all` and `JSON.stringify` force too.

LESSON 5 · 13 MIN

Maybe, Result and railway-oriented code — making absence and failure values

`undefined` and exceptions are JavaScript's answers to "there is no value" and "this failed", and both are invisible in a function's signature: nothing tells the caller that `findUser` may return `undefined` or that `parseConfig` may throw. Two small wrapper types make those outcomes explicit **values**: `Maybe` (a value that may be absent) and `Result` (a success or a typed failure). With `map` and `flatMap` on them, a chain of steps that might each fail becomes a straight line — the "railway": success rides one track, the first failure switches to the other and skips the rest. This lesson builds both types in a few dozen lines, shows the railway, relates them to things you already use (`?.` , `??` , arrays, promises), and is frank about when plain JavaScript idioms are the better choice.

Maybe

```
const Some = (value) => ({
  isSome: true,
  map: (f) => Some(f(value)),
  flatMap: (f) => f(value),                    // f returns a Maybe — avoids S
  filter: (p) => (p(value) ? Some(value) : None),
  getOrElse: () => value,
  toString: () => `Some(${JSON.stringify(value)})`,
});
const None = {
  isSome: false,
  map: () => None, flatMap: () => None, filter: () => None,
  getOrElse: (fallback) => fallback,
  toString: () => "None",
};
const fromNullable = (v) => (v === null || v === undefined ? None : Some(v));

const city = fromNullable(users.find((u) => u.id === id))
  .flatMap((u) => fromNullable(u.address))
  .map((a) => a.city.toUpperCase())
  .getOrElse("unknown");
```

`map` transforms the value if present; `flatMap` chains a step that itself may produce absence; `None` short-circuits everything; `getOrElse` leaves the type at the end. Compare with the built-in: `users.find(…)?.address?.city?.toUpperCase() ?? "unknown"`. For a chain of property accesses, **optional chaining wins** — shorter, standard, no allocation. `Maybe` earns its place when steps are *functions* that may return nothing (lookups, parses, `find`s), when you want to `filter` in the middle, or when you pass the possibly-absent value around before deciding what to do with it.

Result

```
const Ok = (value) => ({
  ok: true, value,
  map: (f) => Ok(f(value)),
  mapErr: () => Ok(value),
  flatMap: (f) => f(value),
  getOrElse: () => value,
  match: ({ ok }) => ok(value),
});
const Err = (error) => ({
  ok: false, error,
  map: () => Err(error),
  mapErr: (f) => Err(f(error)),
  flatMap: () => Err(error),
  getOrElse: (fallback) => fallback,
  match: ({ err }) => err(error),
});
const attempt = (fn) => { try { return Ok(fn()); } catch (e) { return Err(e);
```

`Result` carries **why** it failed, which `Maybe` cannot. `map` transforms a success; `mapErr` transforms a failure (add context, translate to a user message); `flatMap` chains a step that may fail; `match` forces the caller to handle both cases. `attempt` wraps a throwing function — the adapter between exception-based code (`JSON.parse`) and Result-based code.

The railway

```
const parseJson = (text) => attempt(() => JSON.parse(text)).mapErr(() => "not
const requireObject = (v) => (v !== null && typeof v === "object" && !Array.is
const requirePort = (cfg) => (Number.isInteger(cfg.port) && cfg.port > 0 && cf
const withDefaults = (cfg) => ({ retries: 3, ...cfg });

const loadConfig = (text) => parseJson(text).flatMap(requireObject).flatMap(re

loadConfig('{"port":8080}').match({ ok: (cfg) => start(cfg), err: (msg) => cor
```

Each step is a small pure function; `flatMap` connects those that can fail, `map` those that cannot; the first `Err` rides the failure track to the end untouched, so no step needs an `if (!ok) return`. Errors are values: collect them, count them, log them, return them from an API. Module 7's result objects (`{ ok, value } | { ok, error }`) are this pattern without the methods; adding `map` / `flatMap` is what removes the `if (!r.ok) return r` from every line.

Collecting many results

`Result` short-circuits at the first failure. For a form or a batch you want **all** failures: fold the results — `const errors = results.filter((r) => !r.ok).map((r) => r.error)`; if none, `Ok(results.map((r) => r.value))`. That is a `sequence` / `traverse` in functional vocabulary; write it as a plain reduce. `Promise.allSettled` is the async version of the same idea.

You already use these shapes

Arrays have `map` / `flatMap`: an array is "zero or more values" and `flatMap` chains steps that return arrays.

Promises are `Result` for the future: `then` is `map` / `flatMap` (returned promises flatten), `catch` is `mapErr`+recovery, a rejection rides the failure track past every `then`.

`?.` / `??` are a built-in `Maybe` for property paths.

The word for a type with a lawful `map` is *functor*; with `flatMap` (`chain`, `bind`) and a constructor (`Some`, `Ok`, `Promise.resolve`), *monad*. The laws just say `map(identity)` changes nothing and chaining composes associatively — properties you can check in a test and otherwise ignore. You do not need the vocabulary to use the pattern well.

Where to draw the line

Use `Maybe` / `Result` when: a pipeline has several steps that may each fail or return nothing; failures must be data (APIs, validation reports, batch jobs); or you want the signature to *say* that absence/failure is possible (TypeScript makes this compelling — `Result<Config, ConfigError>`). Prefer plain idioms when: a chain is property access (`?.`), a single call may fail (`try` / `catch` at the boundary), or the code is a script whose failure should simply stop it. A codebase where every function returns `Result` and every call site unwraps it has replaced exceptions with ceremony; the railway is for the stretches where several fallible steps line up.

Common mistakes

- `map` where `flatMap` was needed — `Some(Some(x))` , `Ok(Ok(x))` , a `Result` inside a `Result` .
- Unwrapping (`getOrElse` , `.value`) in the middle of a chain instead of at the end.
- Losing the error: converting `Err` to `None` , or `mapErr` to a message too early to distinguish causes.
- Mixing exceptions and Results inside one layer — convert at the boundary with `attempt` .
- Building a `Maybe` for a three-level property access that `?.` already handles.
- Making the types heavier (classes, prototypes, generics) than the problem needs.

Interview angle

- *"What problem do `Maybe` / `Result` solve?"* They make absence and failure explicit values with `map` / `flatMap` , so chains of fallible steps read as straight lines and callers must handle both outcomes.
- *"`map` versus `flatMap` on these?"* `map` applies a plain function to the inner value; `flatMap` applies a function that itself returns the wrapper, avoiding nesting.
- *"How do promises relate?"* A promise is a `Result` over time — `then` maps/chains, rejection is the failure track.
- *"When is optional chaining better than `Maybe` ?"* For property paths; `Maybe` for chaining functions that may return nothing.
- *"How do you collect all errors instead of stopping at the first?"* Fold the results: partition into failures and successes (`sequence` / `traverse`).

Key takeaways

- `Maybe` = `Some(v)` | `None` with `map` / `flatMap` / `filter` / `getOrElse`; `Result` = `Ok(v)` | `Err(e)` with `map` / `mapErr` / `flatMap` / `match`; `attempt` converts throws.
- Railway: `flatMap` for fallible steps, `map` for safe ones; the first failure skips the rest; unwrap once at the end.
- Collect all failures with a fold when partial results matter.
- Arrays, promises and `?.` / `??` are the same shapes you already use; the laws are just sanity checks.
- Use the pattern for stretches of several fallible steps and for failures-as-data; plain idioms elsewhere.

EXERCISE 1

Chaining Maybes

Using the starter's `Some` / `None` / `fromNullable`, for each id on the input lines: find the user (`fromNullable`), `flatMap` to its `address`, `map` the city to upper case, `filter` cities longer than 3 characters, and print `<id>: maybe=<String(maybe)> value=<getOrElse("unknown")> optionalChaining=<the same lookup with ?. and ??> name=<user's name via Maybe or nobody>`. Finally print `nested: map → Some containing <Some(2).map((x) ⇒ Some(x)).getOrElse()> | flatMap → <Some(2).flatMap((x) ⇒ Some(x))>`.

Example: users `[{"id":1,"name":"Ada","address":{"city":"London"}},{"id":2,"name":"Bob"},{"id":3,"name":"Cy","address":{"city":"Rio"}}]` then `1`, `2`, `3`, `9` →

```
1: maybe=Some("LONDON") value=LONDON optionalChaining=LONDON name=Ada
2: maybe=None value=unknown optionalChaining=unknown name=Bob
3: maybe=None value=unknown optionalChaining=RIO name=Cy
9: maybe=None value=unknown optionalChaining=unknown name=nobody
nested: map → Some containing Some(2) | flatMap → Some(2)
```

`filter` is what the Maybe chain can do that optional chaining cannot — Rio is too short.

INPUT 1

```
[{"id":1,"name":"Ada","address":
{"city":"London"}},
{"id":2,"name":"Bob"},
{"id":3,"name":"Cy","address":
{"city":"Rio"}}]
1
2
3
9
```

OUTPUT 1

```
1: maybe=Some("LONDON") value=LONDON
optionalChaining=LONDON name=Ada
2: maybe=None value=unknown
optionalChaining=unknown name=Bob
3: maybe=None value=unknown
optionalChaining=RIO name=Cy
9: maybe=None value=unknown
optionalChaining=unknown name=nobody
nested: map → Some containing Some(2) |
flatMap → Some(2)
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) ⇒ l.trim
const Some = (value) ⇒ ({ isSome: true, map: (f) ⇒ Some(f(value)), flatMap: (f) ⇒
const None = { isSome: false, map: () ⇒ None, flatMap: () ⇒ None, filter: () ⇒ No
const fromNullable = (v) ⇒ (v === null || v === undefined ? None : Some(v));
const users = JSON.parse(lines[0]);
const findUser = (id) ⇒ fromNullable(users.find((u) ⇒ u.id === id));
for (const id of lines.slice(1).map(Number)) {
  const city = findUser(id)
    .flatMap((u) ⇒ fromNullable(u.address))
    .map((a) ⇒ a.city.toUpperCase())
    .filter((c) ⇒ c.length > 3);
  const chained = users.find((u) ⇒ u.id === id)?.address?.city?.toUpperCase() ?? "u
  console.log(`${id}: maybe=${city} value=${city.getOrElse("unknown")} optionalChair
}
const inner = Some(2).map((x) ⇒ Some(x)).getOrElse();       // map wrapped the Mayb
console.log(`nested: map → Some containing ${inner} | flatMap → ${Some(2).flatMap(
```

EXERCISE 2

A Result railway

With the starter's `Ok` / `Err` / `attempt`, build `loadUser(text) = parseJson → requireObject → requireName → requireAge → normalise`: `parseJson` wraps `JSON.parse` with `attempt` and maps the error to `"not valid JSON"`; `requireObject` rejects non-objects with `"expected an object"`; `requireName` needs a non-blank string (`"name is required"`); `requireAge` needs an integer 0–150 (`age must be an integer 0-150, got <JSON>`); `normalise` returns `{ name: trimmed lower-case, age, adult: age ⩾ 18 }`. Print `<i>: ok <JSON>` or `<i>: err <message>` per line, then a `sequence` over all results: `all ok: <n> users` or `<n> failed: <messages joined by " | ">`, then `recovered=<first result's name or anonymous> mapErrDemo=<Err("x").mapErr(upper).error>`.

Example: `{"name":" Ada ","age":36}`, `{"name":"Bob","age":-1}`, `not json`, `[1,2]`, `{"age":5}` →

```
0: ok {"name":"ada","age":36,"adult":true}
1: err age must be an integer 0-150, got -1
2: err not valid JSON
3: err expected an object
4: err name is required
4 failed: age must be an integer 0-150, got -1 | not valid JSON | expected an object
recovered=ada mapErrDemo=X
```

INPUT 1

```
{"name":" Ada ","age":36}
{"name":"Bob","age":-1}
not json
[1,2]
{"age":5}
```

OUTPUT 1

```
0: ok
{"name":"ada","age":36,"adult":true}
1: err age must be an integer 0-150, got
-1
2: err not valid JSON
3: err expected an object
4: err name is required
4 failed: age must be an integer 0-150,
got -1 | not valid JSON | expected an
object | name is required
recovered=ada mapErrDemo=X
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) ⇒ l.trim
const Ok = (value) ⇒ ({ ok: true, value, map: (f) ⇒ Ok(f(value)), mapErr: () ⇒ Ok
const Err = (error) ⇒ ({ ok: false, error, map: () ⇒ Err(error), mapErr: (f) ⇒ Er
const attempt = (fn) ⇒ { try { return Ok(fn()); } catch (e) { return Err(e); } };
const parseJson = (text) ⇒ attempt(() ⇒ JSON.parse(text)).mapErr(() ⇒ "not valid
const requireObject = (v) ⇒ (v ≠= null && typeof v ≡= "object" && !Array.isArray(
const requireAge = (u) ⇒ (Number.isInteger(u.age) && u.age ⩾ 0 && u.age ⩽ 150 ? O
const requireName = (u) ⇒ (typeof u.name ≡= "string" && u.name.trim() ? Ok(u) : Er
const normalise = (u) ⇒ ({ name: u.name.trim().toLowerCase(), age: u.age, adult: u.
const loadUser = (text) ⇒ parseJson(text).flatMap(requireObject).flatMap(requireNam
const results = lines.map(loadUser);
results.forEach((r, i) ⇒ console.log(r.match({ ok: (u) ⇒ `${i}: ok ${JSON.stringif
const sequence = (rs) ⇒ { const errors = rs.filter((r) ⇒ !r.ok).map((r) ⇒ r.error
console.log(sequence(results).match({ ok: (all) ⇒ `all ok: ${all.length} users`, er
console.log(`recovered=${results[0].map((u) ⇒ u.name).getOrElse("anonymous")} mapEr
```

QUIZ

1. `map` versus `flatMap` on a `Maybe`:

A. Identical

B. `map` applies a plain function inside; `flatMap` applies a function that itself returns a `Maybe`, avoiding `Some(Some(x))`

C. `flatMap` is for arrays only

D. `map` can fail

2. In a Result railway, after the first `Err`…

A. Later steps still run

B. Every following `flatMap` / `map` is skipped and the `Err` reaches the end unchanged

C. It throws

D. It becomes `None`

3. For `user?.address?.city ?? "unknown"`, a `Maybe` chain is…

A. Required

B. Overkill — optional chaining already handles property paths; `Maybe` earns its place when chaining functions that may return nothing, or filtering

C. Faster

D. Standard

4. Converting exception-based code into Results is done…

A. Everywhere

B. At the boundary with an `attempt(fn)` wrapper — inside one layer, use one style

C. Never

D. By catching in every function

5. A promise is like a `Result` because…

A. It is synchronous

B. `then` maps/chains, a rejection rides the failure track past every `then` to the `catch`

C. It has a `value` field

D. It cannot fail

ANSWER KEY · MAYBE, RESULT AND RAILWAY-ORIENTED CODE — MAKING ABSENCE AND FAILURE VALUES

1. B Same distinction as arrays and promises.

2. B No step needs `if (!ok) return`.

3. B Use the built-in idiom for property access.

4. B Mixing both inside a layer defeats the point.

5. B Arrays are the "zero or more" container with the same `map` / `flatMap` shape.

CHECKPOINT · 24 MIN

Checkpoint — Functional patterns

This checkpoint covers pure functions and the pure-core/impure-shell design, immutable updates with structural sharing, `pipe` / `compose`, currying, partial application and data-last design, `reduce` as the universal fold and its O(n²) trap, recursion limits and trampolines, memoization (keys, bounding, memoized recursion), `once`, thunks and lazy sequences, and `Maybe` / `Result` railways.

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 two conditions for purity; where effects belong; why `prev.x === next.x` works as a change detector after an immutable update.
- `pipe` versus `compose` order; why auto-curry breaks on default parameters; what data-last buys.
- `map` / `filter` / `groupBy` as `reduce`; why spreading the accumulator is O(n²); why V8 has no tail calls and what a trampoline does.
- What makes a good memoization key; when a cache must be bounded; why the recursive call must go through the memoized name.
- `map` versus `flatMap` on `Maybe` / `Result`; how the railway short-circuits; when `?.` beats `Maybe`.

The programs are a tiny immutable store with a reducer, history and a memoized selector; a pipeline DSL that parses a text description into composed data-last functions; and a Result-based configuration loader that mirrors module 7's exception-based one.

EXERCISE 1

A tiny immutable store with a memoized selector

Write a pure `reducer(state, action)` for `{ title, todos }`: `add <text>` appends `{ id, text, done: false }` (ids from 1), `toggle <id>` flips `done` via `map`, `remove <id>` filters, `rename <title>` changes only the title. Keep a `history` of states (`undo` pops the last one and prints `undo → <texts>`). A memoized selector `completedCount(state)` recomputes only when `state.todos` is a new reference and counts its runs. After each action print `<command>: todos=<["id:text*" for`

`done]> todosChanged=<next.todos !== prev.todos> sharedItems=<how many todo objects are the same references as before> completed=<selector> selectorRuns=<runs>`; finish with `history=<length> firstStateStillEmpty=<the initial state's todos is still empty>`.

Example: `add buy milk`, `add write code`, `toggle 1`, `rename shopping`, `remove 2`, `undo`, `toggle 1` →

```
add buy milk: todos=["1:buy milk"] todosChanged=true sharedItems=0 completed=0 selec
add write code: todos=["1:buy milk","2:write code"] todosChanged=true sharedItems=1
toggle 1: todos=["1:buy milk*","2:write code"] todosChanged=true sharedItems=1 compl
rename shopping: todos=["1:buy milk*","2:write code"] todosChanged=false sharedItems
remove 2: todos=["1:buy milk*"] todosChanged=true sharedItems=1 completed=1 selector
undo → ["buy milk","write code"]
toggle 1: todos=["1:buy milk","2:write code"] todosChanged=true sharedItems=1 comple
history=6 firstStateStillEmpty=true
```

`rename` left `todos` untouched, so the selector did not run — that is why immutable updates matter.

INPUT 1

```
add buy milk
add write code
toggle 1
rename shopping
remove 2
undo
toggle 1
```

OUTPUT 1

```
add buy milk: todos=["1:buy milk"]
todosChanged=true sharedItems=0
completed=0 selectorRuns=1
add write code: todos=["1:buy
milk","2:write code"] todosChanged=true
sharedItems=1 completed=0 selectorRuns=2
toggle 1: todos=["1:buy milk*","2:write
code"] todosChanged=true sharedItems=1
completed=1 selectorRuns=3
rename shopping: todos=["1:buy
milk*","2:write code"]
todosChanged=false sharedItems=2
completed=1 selectorRuns=3
remove 2: todos=["1:buy milk*"]
todosChanged=true sharedItems=1
completed=1 selectorRuns=4
undo → ["buy milk","write code"]
toggle 1: todos=["1:buy milk","2:write
code"] todosChanged=true sharedItems=1
completed=0 selectorRuns=5
history=6 firstStateStillEmpty=true
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
let nextId = 1;
function reducer(state, action) {
  switch (action.type) {
    case "add": return { ...state, todos: [...state.todos, { id: nextId++, text: act
    case "toggle": return { ...state, todos: state.todos.map((t) => (t.id === actior
    case "remove": return { ...state, todos: state.todos.filter((t) => t.id !== acti
    case "rename": return { ...state, title: action.title };
    default: return state;
  }
}
let selectorRuns = 0;
const completedCount = (() => { let lastTodos, lastValue; return (state) => {
  if (state.todos !== lastTodos) { selectorRuns++; lastTodos = state.todos; lastValu
  return lastValue;
}; })();
const history = [{ title: "list", todos: [] }];
for (const line of lines) {
  const [cmd, ...rest] = line.trim().split(/\s+/);
  const state = history.at(-1);
  if (cmd === "undo") { if (history.length > 1) history.pop(); console.log(`undo →
  const action = cmd === "add" ? { type: "add", text: rest.join(" ") } : cmd === "re
  const next = reducer(state, action);
  history.push(next);
  const untouched = state.todos.filter((t) => next.todos.includes(t)).length;
  console.log(`${line.trim()}: todos=${JSON.stringify(next.todos.map((t) => `${t.id}
}
console.log(`history=${history.length} firstStateStillEmpty=${history[0].todos.lengt
```

EXERCISE 2

A pipeline DSL compiled to functions

Line 1 is a pipeline such as `filter age ⩾ 30 | sort age desc | map name`; the rest are JSON records. Write `compile(step)` returning a function over an array for: `filter <field><op><value>` (ops `> < ⩾ ⩽ = ≠`; numeric literals become numbers), `map <field>`, `sort [field] [desc]` (a sorted copy), `take <n>`, `uniq`, `count`, `sum [field]`. Split the spec on `|`, compile each step, `pipe` them and print `<spec> ⇒ <JSON result>`; an unknown step prints `SyntaxError: unknown step '<name>'`.

Example: `filter age ⩾ 30 | sort age desc | map name` over `{"name":"ada","age":36}`, `{"name":"bob","age":17}`, `{"name":"cy","age":41}` →

```
filter age ⩾ 30 | sort age desc | map name ⇒ ["cy","ada"]
```

`filter city=london | count` → `2`; `map city | uniq | sort` → `["oslo","rome"]`.

INPUT 1

```
filter age>=30 | sort age desc | map name
{"name":"ada","age":36}
{"name":"bob","age":17}
{"name":"cy","age":41}
```

OUTPUT 1

```
filter age>=30 | sort age desc | map name => ["cy","ada"]
```

INPUT 2

```
filter city=london | count
{"name":"ada","city":"london"}
{"name":"bob","city":"paris"}
{"name":"cy","city":"london"}
```

OUTPUT 2

```
filter city=london | count => 2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const [spec, ...recordLines] = lines;
const records = recordLines.map((l) => JSON.parse(l));
const OPS = { ">": (a, b) => a > b, "<": (a, b) => a < b, "⩾": (a, b) => a ⩾ b, "
const literal = (text) => (/^-?\d+(\.\d+)?$/.test(text) ? Number(text) : text.replac
const cmp = (a, b) => (a < b ? -1 : a > b ? 1 : 0);
function compile(step) {
  const [op, ...args] = step.trim().split(/\s+/);
  switch (op) {
    case "filter": {
      const m = /^(\w+)(⩾|⟸|≔|≠|▷|<)(.+)$/.exec(args.join(""));
      if (!m) throw new SyntaxError(`bad filter '${args.join(" ")}'`);
      const [, field, cmpOp, raw] = m;
      const value = literal(raw);
      return (xs) => xs.filter((x) => OPS[cmpOp](x[field], value));
    }
    case "map": return (xs) => xs.map((x) => x[args[0]]);
    case "sort": {
      const [field, dir] = args;
      const sign = dir ⩵ "desc" ? -1 : 1;
      return (xs) => [...xs].sort((a, b) => sign * (field ? cmp(a[field], b[field])
    }
    case "take": return (xs) => xs.slice(0, Number(args[0]));
    case "uniq": return (xs) => [...new Set(xs)];
    case "count": return (xs) => xs.length;
    case "sum": return (xs) => xs.reduce((s, x) => s + (args[0] ? x[args[0]] : x), 0
    default: throw new SyntaxError(`unknown step '${op}'`);
  }
}
const pipe = (...fns) => (x) => fns.reduce((acc, f) => f(acc), x);
try {
  const steps = spec.split("|").map(compile);
  console.log(`${spec.trim()} ⇒ ${JSON.stringify(pipe(...steps)(records))}`);
} catch (err) {
  console.log(`${err.name}: ${err.message}`);
}
```

EXERCISE 3

The config loader, Result style

Rebuild module 7's configuration loader with `Result` and **every** field error collected. `parseJson` maps a parse failure to `["config is not valid JSON"]`; `requireObject` rejects with `["config must be an object, got <null|array|typeof>"]`; field validators return `Ok(value)` or `Err(message)`: `host` (non-empty string, trimmed), `port` (integer 1–65535), `retries` (default 3;

non-negative integer) with messages `host must be a non-empty string, got <JSON>`, `port must be 1-65535, got <JSON>`, `retries must be a non-negative integer, got <JSON>`; `validateFields` runs all three and returns `Err(["<field>: <message>", …])` or `Ok(config)`. Print `ok: host=<> port=<> retries=<>` or `errors(<n>): <joined by "; ">` per line.

Example: `{"host":"db.local","port":5432}`, `{"host":"","port":70000,"retries":-1}`, `{oops`, `[1]` →

```
ok: host=db.local port=5432 retries=3
errors(3): host: host must be a non-empty string, got ""; port: port must be 1-65535
errors(1): config is not valid JSON
errors(1): config must be an object, got array
```

Module 7's version stopped at the first bad field; this one reports all three.

INPUT 1

```
{"host":"db.local","port":5432}
{"host":"","port":70000,"retries":-1}
{oops
[1]
{"host":"x","port":80,"retries":0}
```

OUTPUT 1

```
ok: host=db.local port=5432 retries=3
errors(3): host: host must be a non-empty string, got ""; port: port must be 1-65535, got 70000; retries: retries must be a non-negative integer, got -1
errors(1): config is not valid JSON
errors(1): config must be an object, got array
ok: host=x port=80 retries=0
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const Ok = (value) => ({ ok: true, value, map: (f) => Ok(f(value)), mapErr: () => Ok
const Err = (error) => ({ ok: false, error, map: () => Err(error), mapErr: (f) => Er
const attempt = (fn) => { try { return Ok(fn()); } catch (e) { return Err(e); } };
const parseJson = (text) => attempt(() => JSON.parse(text)).mapErr(() => ["config is
const requireObject = (v) => (v !== null && typeof v === "object" && !Array.isArray(
const fields = {
  host: (v) => (typeof v === "string" && v.trim() ? Ok(v.trim()) : Err(`host must be
  port: (v) => (Number.isInteger(v) && v >= 1 && v <= 65535 ? Ok(v) : Err(`port must
  retries: (v) => (v === undefined ? Ok(3) : Number.isInteger(v) && v >= 0 ? Ok(v) :
};
const validateFields = (raw) => {
  const results = Object.entries(fields).map(([name, check]) => [name, check(raw[nam
  const errors = results.filter(([, r]) => !r.ok).map(([name, r]) => `${name}: ${r.e
  return errors.length ? Err(errors) : Ok(Object.fromEntries(results.map(([name, r])
};
const loadConfig = (text) => parseJson(text).flatMap(requireObject).flatMap(validate
for (const line of lines) {
  console.log(loadConfig(line).match({
    ok: (cfg) => `ok: host=${cfg.host} port=${cfg.port} retries=${cfg.retries}`,
    err: (errors) => `errors(${errors.length}): ${errors.join("; ")}`,
  }));
}
```

CHECKPOINT QUESTIONS

1. `const b = a; b.x = 1;` — is this an immutable update?

A. Yes

B. No — `b` is an alias; `a.x` changed too. `const b = { ...a, x: 1 }` is the immutable form

C. Yes, in strict mode

D. Only for arrays

2. Why do memoized selectors (Redux `reselect`, React `useMemo`) rely on immutability?

A. They deep-compare

B. They compare inputs by reference — an unchanged slice keeps its reference, a changed one is a new object, so `===` decides whether to recompute

C. They copy the state

D. They do not

3. `const map = curry((f, xs) ⇒ xs.map(f))` — `map(double)` returns…

A. An array

B. A function waiting for the array — a unary step for `pipe`

C. `undefined`

D. An error

4. `groupBy` as a `reduce`:

A. `xs.reduce((g, x) ⇒ ({ ...g, [k(x)]: [...(g[k(x)] ?? []), x] }), {})`

B. `xs.reduce((g, x) ⇒ { (g[k(x)] ??= []).push(x); return g; }, {})` — mutating the local accumulator, O(n)

C. `xs.map(k)`

D. `xs.filter(k)`

5. Tree recursion (two recursive calls whose results combine) at great depth should use…

A. A trampoline

B. An explicit stack or queue — trampolines only handle tail calls

C. `setTimeout`

D. Deeper recursion

6. Memoizing `getUser(id)` when users change in the database…

A. Is fine

B. Returns stale data — the inputs do not capture everything the result depends on; add invalidation or do not cache

C. Speeds writes

D. Is required

7. `once(fn)` is best described as…

A. A timer

B. Memoization with a single slot and no key — run-exactly-once initialisation

C. A generator

D. A promise

8. `fallback = buildLogger()` as a default parameter…

A. Is lazy

B. Runs `buildLogger()` whenever the argument is omitted, even if the value is never used — pass the thunk `buildLogger` and call it only when needed

C. Runs once

D. Is cached

9. `Some(2).map((x) ⇒ Some(x))` yields…

A. `Some(2)`

B. `Some(Some(2))` — nested; `flatMap` is needed for a `Maybe`-returning function

C. `None`

D. `2`

10. To report every validation error rather than the first, you…

A. Use `flatMap`

B. Run all validators, collect the `Err`s, and return `Err(list)` or `Ok(values)` — a `sequence` / `traverse`

C. Throw

D. Use `Maybe`

11. Which is a legitimate reason to prefer a plain `for` loop over a fold?

A. Loops are always faster

B. The body needs early exit or several accumulators, or the callback would be long — readability

C. Folds are deprecated

D. Loops are immutable

12. The `attempt(fn)` adapter…

A. Retries

B. Runs a throwing function and returns `Ok(result)` or `Err(error)` — the bridge from exceptions to Results at a boundary

C. Logs errors

D. Makes functions pure

ANSWER KEY · CHECKPOINT — FUNCTIONAL PATTERNS

1. B `const` prevents rebinding, not mutation.

2. B Mutating in place would make the reference stale and the selector wrong.

3. B Data-last plus currying is what makes helpers composable.

4. B The first option is correct but O(n²) from spreading.

5. B Depth-first with a stack array is the standard iterative form.

6. B Cache pure results, or impure ones with an explicit invalidation story.

7. B The lazy alternative to doing work at import time.

8. B Default parameter expressions are evaluated per call.

9. B Same as `[1].map((x) ⇒ [x])` giving `[[1]]`.

10. B `flatMap` short-circuits by design; collection is a fold.

11. B Choose by readability first, then measure.

12. B `JSON.parse` becomes `parseJson` returning a Result.

CodeKairo · JavaScript study plan · Module 11: Functional patterns. Solve the exercises online to track your progress and earn XP.