Module 3 of 16

Functions, scope and closures

Declarations, expressions and arrows; parameters with defaults and rest; lexical scope and hoisting; closures and their patterns; the five rules for this with call, apply and bind; higher-order functions, composition and currying.

Functions are the unit of everything in JavaScript — the module, the method, the callback, the component — and the three ideas that make them powerful are the three that trip people up: lexical scope, closures, and the way this is decided at the call rather than at the definition.

This module takes them in order. First the syntax: declarations, expressions and arrows, how each hoists, and parameters with defaults, rest and spread. Then scope: the chain the engine builds when it parses the file, hoisting stated precisely, and what modules do to the global namespace. Closures get a full lesson — the mechanism, private state, configured functions, memoisation, and the loop-variable and memory pitfalls. The this lesson states the five binding rules in precedence order and gives the three fixes for a detached method. The module ends with the higher-order style: pipe and compose, currying and partial application, debounce and throttle, and the error-first callback convention Node still uses.

The exercises build the machinery — a scope-chain resolver, a counter factory, a this-rule tracer, a pipeline interpreter — so that each idea is something you have implemented, not just read.

CONTENTS

1	Declaring functions — declarations, expressions, arrows and parameters	13 min
2	Lexical scope, the scope chain and hoisting	12 min
3	Closures — functions that remember	14 min
4	this — the five binding rules, call, apply and bind	15 min
5	Higher-order functions — composition, currying and callbacks	13 min
✓	Checkpoint — Functions, scope and closures	24 min

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

LESSON 1 · 13 MIN

Declaring functions — declarations, expressions, arrows and parameters

Functions are the unit of everything in JavaScript: the module, the class method, the callback, the component. The language gives you three ways to write one — a declaration, an expression, an arrow — and they differ in hoisting, in `this` (next lessons) and in how they read. Parameters have their own conveniences: defaults, rest, spread at the call site, destructuring. This lesson is the syntax and semantics of defining and calling functions, with the first-class nature of functions — values you can pass and return — as the thread running through it.

Three forms

```
function add(a, b) { return a + b; }                  // declaration: hoisted v
const sub = function (a, b) { return a - b; };        // expression: a value as
const mul = (a, b) => a * b;                          // arrow: concise, expres
const div = (a, b) => { const r = a / b; return r; } // arrow with a block bod
```

A **declaration** is hoisted *with its body*: you may call `add` above the line that defines it. An **expression** and an **arrow** are values; the `const` holding them is in the temporal dead zone until its line runs, so call them only afterwards. Arrows differ from the other two in three ways that later lessons need: no own `this`, no `arguments` object, and they cannot be constructors. Style: declarations for top-level named functions, arrows for callbacks and short helpers, and function expressions rarely (mostly when a callback needs its own `this`).

An arrow returning an **object literal** needs parentheses — `() => ({ ok: true })` — because braces alone start a block body.

Functions are values

```
const ops = { add, sub, mul };                  // stored in an object
[1, 2, 3].map((x) => x * 2);                    // passed as an argument
function twice(f) { return (x) => f(f(x)); }    // returned from a function
const inc2 = twice((x) => x + 1);  inc2(5);     // 7
typeof add;                                     // "function"
add.name;                                       // "add" — expressions get the
add.length;                                     // 2 — declared parameter count
```

A function that takes or returns functions is a **higher-order function** (lesson 5). Because functions are objects they have properties (`name`, `length`) and can carry more if you attach them — rarely wise.

Parameters

```
function greet(name = "world", punctuation = "!") { return `Hello, ${name}${pu
greet();               // Hello, world!
greet("Ada");          // Hello, Ada!
greet(undefined, "?"); // Hello, world?   — undefined triggers the default; nu
```

Defaults apply when the argument is `undefined` (missing or explicitly `undefined`), not for `null` or other falsy values, and may reference earlier parameters: `(a, b = a * 2)`. Too few arguments leave the rest `undefined`; too many are ignored (no error). **Rest parameters** collect the remainder into a real array:

```
function sum(...nums) { return nums.reduce((a, b) => a + b, 0); }
sum(1, 2, 3);                    // 6
function log(level, ...parts) { console.log(level, parts.join(" ")); }
```

Rest must be last. The older `arguments` object (array-like, all arguments, not in arrows) is what rest replaced; do not use it in new code. **Spread** at the call site does the inverse — `sum(...[1, 2, 3])`, `Math.max(...arr)` — and **destructuring** parameters unpack objects and arrays in the signature: `function draw({ x, y, color = "black" }) {}` called as `draw({ x: 1, y: 2 })` — the idiom for "named arguments" (module 4).

Return

A function with no `return`, or a bare `return;`, returns `undefined`. Only one value comes back; return an array or object for several — `return [min, max]` then `const [lo, hi] = range(xs)`. Early returns (guard clauses) keep the main path unindented. Remember the ASI trap: the value must start on the same line as `return`.

Calling

`f(a, b)` — positional; `f({ a, b })` — an options object, which reads better past two or three parameters and lets callers omit any; `f?.()` — call only if `f` is not nullish; `f.call(thisValue, a, b)`, `f.apply(thisValue, [a, b])` — explicit `this` (lesson 4); `new F()` — construct (module 5). A missing function is the most common runtime error: `undefined is not a function` means the thing before the parentheses was not what you thought.

Recursion

A function may call itself; the call stack limits depth to roughly 10 000 frames (`RangeError: Maximum call stack size exceeded`). Node does not do tail-call optimisation, so a deep recursion — walking a long linked list, a naive `sum(n) = n + sum(n - 1)` for large `n` — must be rewritten as a loop or with an explicit stack.

Immediately invoked function expressions

```
(function () { const secret = 1; /* … */ })();
(() ⇒ { /* … */ })();
```

An **IIFE** creates a scope and runs once — the pre-2015 way to keep variables private in a script. Block scope and modules made it mostly unnecessary; you still see it in bundler output and older code, and occasionally for an async wrapper: `(async () ⇒ { await main(); })()`.

Pure functions and side effects

A **pure** function returns a value determined only by its arguments and changes nothing outside itself. Pure functions are trivially testable, safe to call repeatedly, and what array

methods expect. Push side effects (I/O, mutation, `Date.now()`) to the edges and keep the middle pure — the design habit that makes JavaScript codebases tractable.

Interview angle

"Function declaration versus expression?" Declarations are hoisted with their body; expressions are values assigned at run time (and TDZ-bound if `const`).

"What do arrow functions lack?" Their own `this`, `arguments`, `new`, and a prototype — they are for callbacks and helpers, not methods or constructors.

"When does a default parameter apply?" When the argument is `undefined` — not `null`, not `0`.

"Rest versus spread?" Rest gathers remaining arguments into an array (in a signature); spread expands an iterable into arguments (at a call site).

"What is a higher-order function?" One that takes or returns a function — `map`, `filter`, your own `twice`.

Key takeaways

Declarations hoist with their body; expressions and arrows are values available after their line.

Arrows: expression bodies, `({})` for object returns, no own `this` / `arguments` / `new`.

Defaults trigger on `undefined`; rest gathers; spread expands; destructured parameters give named arguments.

One return value — bundle several in an array or object; guard clauses first.

Functions are values with `name` and `length`; prefer pure functions and keep side effects at the edges.

EXERCISE 1

Three forms, one table

Define `add` as a function **declaration**, `sub` as a function **expression**, `mul` as an **arrow**, and `stats(...nums)` with a rest parameter returning `{ count, total }`. Read `n` pairs `a b` and print `<a> <b>: add=<> sub=<> mul=<>` for each, then `names=<add.name>,<sub.name>,<mul.name>`

`lengths=<add.length>,<sub.length>,<mul.length>,<stats.length>`, then `stats: count=<> total=<>` over every number read. Note that a rest parameter does not count toward `length`.

Example: `2` then `3 4`, `10 -2` →

```
3 4: add=7 sub=-1 mul=12
10 -2: add=8 sub=12 mul=-20
names=add,sub,mul lengths=2,2,2,0
stats: count=4 total=15
```

INPUT 1

```
2
3 4
10 -2
```

OUTPUT 1

```
3 4: add=7 sub=-1 mul=12
10 -2: add=8 sub=12 mul=-20
names=add,sub,mul lengths=2,2,2,0
stats: count=4 total=15
```

INPUT 2

```
1
5 5
```

OUTPUT 2

```
5 5: add=10 sub=0 mul=25
names=add,sub,mul lengths=2,2,2,0
stats: count=2 total=10
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
function add(a, b) { return a + b; }
const sub = function (a, b) { return a - b; };
const mul = (a, b) ⇒ a * b;
function stats(...nums) { return { count: nums.length, total: nums.reduce((s, x) ⇒

const n = Number(tokens[0]);
const all = [];
for (let i = 0; i < n; i++) {
  const a = Number(tokens[1 + 2 * i]), b = Number(tokens[2 + 2 * i]);
  all.push(a, b);
  console.log(`${a} ${b}: add=${add(a, b)} sub=${sub(a, b)} mul=${mul(a, b)}`);
}
const s = stats(...all);
console.log(`names=${add.name},${sub.name},${mul.name} lengths=${add.length},${sub.l
console.log(`stats: count=${s.count} total=${s.total}`);
```

EXERCISE 2

Defaults trigger on undefined

Given `greet(name = "world", punctuation = "!")` and `sum(...nums)`, read `n` commands: `greet <name> <punct>` where `-` means "pass `undefined`" and `null` means "pass `null`" (so the default applies only for `-`); `sum <k> <k numbers>` spreads the numbers into `sum`. Print each result.

Example: `4` then `greet - -`, `greet Ada ?`, `greet null -`, `sum 3 1 2 3` →

```
Hello, world!
Hello, Ada?
Hello, null!
sum=6
```

INPUT 1

```
4
greet - -
greet Ada ?
greet null -
sum 3 1 2 3
```

OUTPUT 1

```
Hello, world!
Hello, Ada?
Hello, null!
sum=6
```

INPUT 2

```
2
sum 0
greet - .
```

OUTPUT 2

```
sum=0
Hello, world.
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
function greet(name = "world", punctuation = "!") { return `Hello, ${name}${punctuat
function sum(...nums) { return nums.reduce((a, b) => a + b, 0); }
let pos = 0;
const next = () => tokens[pos++];
const arg = (t) => (t === "-" ? undefined : t === "null" ? null : t);
const n = Number(next());
for (let i = 0; i < n; i++) {
  const cmd = next();
  if (cmd === "greet") {
    console.log(greet(arg(next()), arg(next())));
  } else {
    const k = Number(next());
    const values = [];
    for (let j = 0; j < k; j++) values.push(Number(next()));
    console.log(`sum=${sum(...values)}`);
  }
}
```

QUIZ

1. Which can be called before the line that defines it?

A. `const f = () ⇒ 1`

B. `function f() {}` — declarations hoist with their body

C. `const f = function () {}`

D. None

2. `(() ⇒ { a: 1 })()` returns…

A. `{ a: 1 }`

B. `undefined` — the braces are a block body with a label; wrap the object in parentheses

C. `1`

D. A syntax error

3. `function f(x = 5) {}; f(null)` — inside, `x` is…

A. `5`

B. `null` — defaults apply only for `undefined`

C. `0`

D. `undefined`

4. Rest parameters versus the `arguments` object:

A. Identical

B. Rest is a real array, works in arrows, and must be last; `arguments` is array-like and absent in arrows

C. `arguments` is newer

D. Rest cannot be empty

5. `f.length` for `function f(a, b = 1, ...rest) {}` is…

A. 3

B. 2

C. 1 — parameters before the first default

D. 0

ANSWER KEY · DECLARING FUNCTIONS — DECLARATIONS, EXPRESSIONS, ARROWS AND PARAMETERS

1. B Expressions and arrows assigned to `const` are in the temporal dead zone until their line.

2. B `() ⇒ ({ a: 1 })` returns the object.

3. B `f()` and `f(undefined)` would both give 5.

4. B Use rest in all new code.

5. C `length` counts leading parameters without defaults; rest never counts.

LESSON 2 · 12 MIN

Lexical scope, the scope chain and hoisting

Scope is the answer to "which `x` does this line mean?" JavaScript answers it **lexically** — by where the code is written, not by who called it — through a chain of nested scopes that the engine builds when it parses the file. Understanding the chain explains closures (next lesson), why `var` leaks, why a function can use a variable declared after it, and what a module does to the global namespace. This lesson is that model, made concrete.

Scopes nest, and lookups climb

```
const app = "shop";                          // module/script scope
function order(id) {                         // function scope: id, total
  let total = 0;
  for (const line of lines(id)) {            // block scope per iteration: line
    total += line.price;                     // total found one scope up; lines j
  }
  return `${app}:${id}:${total}`;            // app found at the top
}
```

Every function body and every block creates a scope; a name is resolved by looking in the current scope, then its enclosing scope, and so on to the global scope. The chain is fixed at **write time** — `order` can see `app` because it is *written* inside the scope where `app` lives, regardless of where `order` is later called from. This is lexical (static) scoping; nothing about the call site changes it. (Dynamic-looking behaviour comes only from `this`, lesson 4.)

Failing to find a name anywhere throws `ReferenceError` on read; on *write* in sloppy mode it creates a global — strict mode makes it the same `ReferenceError`.

The kinds of scope

Global — `globalThis` (`window` in browsers, `global` in Node). Top-level `var` / `function` in a *script* attach to it; `let` / `const` do not; a *module's* top level is its own scope and never leaks.

Module — each ES module (and each CommonJS file, which Node wraps in a function) has its own top-level scope. This is why two files can both declare `const config` without

clashing.

Function — parameters and everything declared in the body. `var` lives here.

Block — `{ }`, including loop and `if` bodies, `switch` blocks, `catch (e)`. `let`, `const`, `class` live here.

Hoisting, precisely

Before running a scope the engine registers every declaration in it:

Declaration	Registered as	Before its line
`function f() {}`	the complete function	callable
`var v`	`v = undefined`	`undefined`
`let` / `const` / `class`	uninitialised	`ReferenceError` (TDZ)
`import`	live binding, hoisted	usable — imports are resolved before the module body runs

So a top-level helper may be defined below the code that calls it — a common and readable layout (main logic first, helpers below) — as long as it is a **declaration**. `const helper = () ⇒ …` below its use fails with a TDZ error.

```
main();
function main() { console.log(square(3)); }   // fine: declarations hoist
function square(n) { return n * n; }
```

Shadowing

An inner scope may declare a name that also exists outside; inside, the inner wins and the outer is unreachable.

```
const x = "outer";
function f() { const x = "inner"; return x; }   // "inner"
```

Legal, sometimes useful, often a bug source (a loop variable shadowing a parameter). Linters offer `no-shadow`. A parameter shadows an outer variable of the same name too.

Closures preview: scopes outlive calls

```
function makeCounter() {
  let n = 0;
  return () => ++n;      // this arrow's scope chain includes makeCounter's sc
}
const c = makeCounter();
c(); c();                // 1, 2 — n lives on after makeCounter returned
```

A function keeps a reference to the scope chain it was created in. When `makeCounter` returns, its scope would normally be discarded — but the returned arrow still refers to it, so it stays. That retained scope is a **closure**, and the whole of the next lesson.

Blocks as scoping tools

```
{
  const tmp = compute();
  use(tmp);
}                            // tmp gone; the name is free again
switch (x) {
  case 1: { const y = 1; break; }   // braces per case give each its own y
}
```

Case clauses share the `switch` block's scope; braces around each case body avoid "identifier has already been declared".

Globals, and how not to make them

Every global is shared by every script on a page or every file in a process — a name clash waiting to happen. Modules fixed this: nothing leaks unless exported. In scripts, wrap in an IIFE or use `let` / `const` at top level. Node's `require`-based files are already wrapped in a function, which is why `var` at the top of a CommonJS file does *not* become a global (unlike a

browser script). Reserve `globalThis.x = …` for the rare genuinely global thing, and name it so nobody mistakes it.

Reading code with the model

To find what a name means: start at the line, look for a declaration in the innermost block; step out block by block to the function; step out function by function to the module; finally the global object. If the nearest declaration is a `let` below the current line, you are in its TDZ. If none exists anywhere, it is a `ReferenceError` (read) or an accidental global (sloppy write). Every scope question in the language yields to that walk.

Interview angle

"What is lexical scope?" Scope determined by where code is written; inner functions see outer variables through the scope chain, fixed at definition.

"What is hoisting?" Declarations are registered before a scope runs — functions fully, `var` as `undefined`, `let` / `const` uninitialised.

"Why can you call a function declared below its use?" Function declarations hoist with their body; `const f = () ⇒ …` does not.

"Script versus module top level?" Script `var` /functions become globals; a module's top level is private and strict.

"What is shadowing?" An inner declaration hiding an outer one of the same name.

Key takeaways

Lookups climb the lexical chain: block → function → module → global; the chain is fixed where the code is written.

Hoisting: functions callable early; `var` is `undefined` early; `let` / `const` / `class` throw early.

Modules and CommonJS files have private top-level scope; browser scripts leak `var` and functions to `window`.

A returned function keeps its defining scope alive — the closure mechanism.

Brace each `switch` case; avoid shadowing; never assign to undeclared names.

EXERCISE 1

A scope-chain resolver

Model lexical scope. Read `n` scopes, one per line: `<name> <parent or -> <declared names…>`. Then `q` queries `<scope> <identifier>`. Resolve each the way the engine does — look in the scope, then its parent, and so on — and print `<identifier> from <scope>: found in <scope> (<hops> up)` or `<identifier> from <scope>: ReferenceError`.

Example

```
3
global - app
order global id total
loop order line
2
loop total
loop app
```

→

```
total from loop: found in order (1 up)
app from loop: found in global (2 up)
```

INPUT 1

```
3
global - app
order global id total
loop order line
3
loop total
loop app
loop id2
```

OUTPUT 1

```
total from loop: found in order (1 up)
app from loop: found in global (2 up)
id2 from loop: ReferenceError
```

INPUT 2

```
1
g - x
2
g x
g y
```

OUTPUT 2

```
x from g: found in g (0 up)
y from g: ReferenceError
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const n = Number(lines[0]);
const parent = new Map(), vars = new Map();
for (let i = 1; i ⩽ n; i++) {
  const [name, p, ...declared] = lines[i].trim().split(/\s+/);
  parent.set(name, p ⩵ "-" ? null : p);
  vars.set(name, new Set(declared));
}
const q = Number(lines[n + 1]);
for (let i = 0; i < q; i++) {
  const [scope, name] = lines[n + 2 + i].trim().split(/\s+/);
  let cur = scope, hops = 0, found = null;
  while (cur ≠ null) {
    if (vars.get(cur).has(name)) { found = cur; break; }
    cur = parent.get(cur);
    hops++;
  }
  console.log(found ? `${name} from ${scope}: found in ${found} (${hops} up)` : `${r
}
```

EXERCISE 2

Hoisting, observed

Read `n`. Call `main()` on the line **above** its declaration. Inside `main`: call `square(n)` (a function declaration written below `main`) and print `declaration: square(<n>)=<…>`; try to call `cube(n)` (a `const` arrow written at the bottom of the file) inside `try` / `catch` and print `expression: cube(<n>)=<…>` or `expression: <error class>`; print `var before assignment: <typeof v>`, then `var v = n`, then `var after assignment: <typeof v>`.

Example: `3` →

```
declaration: square(3)=9
expression: ReferenceError
var before assignment: undefined
var after assignment: number
```

INPUT 1

```
3
```

OUTPUT 1

```
declaration: square(3)=9
expression: ReferenceError
var before assignment: undefined
var after assignment: number
```

INPUT 2

```
0
```

OUTPUT 2

```
declaration: square(0)=0
expression: ReferenceError
var before assignment: undefined
var after assignment: number
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);
main();

function main() {
  console.log(`declaration: square(${n})=${square(n)}`);
  try {
    console.log(`expression: cube(${n})=${cube(n)}`);
  } catch (e) {
    console.log(`expression: ${e.constructor.name}`);
  }
  console.log(`var before assignment: ${typeof v}`);
  var v = n;
  console.log(`var after assignment: ${typeof v}`);
}

function square(x) { return x * x; }
const cube = (x) => x * x * x;
```

QUIZ

1. Lexical scope means a function sees…

A. The variables of whoever calls it

B. The variables of the scopes it was written inside, fixed at definition

C. Only its parameters

D. Only globals

2. In a Node CommonJS file, top-level `var x` …

A. Becomes a global

B. Stays private to the file — Node wraps each file in a function

C. Is a syntax error

D. Becomes `module.exports.x`

3. Two `const y` declarations in different `case` clauses of one `switch` …

A. Are fine

B. Clash — case clauses share the switch block; wrap each case body in braces

C. Shadow each other

D. Are hoisted

4. `import` declarations…

A. Run where written

B. Are hoisted and resolved before the module body runs

C. Are in a TDZ

D. Are lazy

5. Shadowing is…

A. A syntax error

B. An inner declaration hiding an outer one of the same name — legal, often confusing

C. Copying a variable

D. Hoisting

ANSWER KEY · LEXICAL SCOPE, THE SCOPE CHAIN AND HOISTING

1. B Dynamic scoping would use the call site; JavaScript never does (except for `this`).

2. B In a browser script it would become `window.x`.

3. B `SyntaxError: Identifier 'y' has already been declared`.

4. B You may use an import above the `import` line (though style says keep imports at the top).

5. B Linters offer `no-shadow` for a reason.

LESSON 3 · 14 MIN

Closures — functions that remember

A closure is a function together with the variables it can see from where it was defined. Every JavaScript function is one; the word matters when the function *outlives* the scope it was born in and keeps that scope alive — a counter that remembers its count, an event handler that remembers which button it belongs to, a memoised function that remembers past results. Closures are how JavaScript does private state, configuration, callbacks and modules, and they are the single most-asked concept in JavaScript interviews. This lesson builds them from the scope chain and covers the patterns and the two classic pitfalls.

The mechanism

```
function makeGreeter(greeting) {
  return function (name) {              // created inside makeGreeter's scope…
    return `${greeting}, ${name}!`;     // …and it reads greeting from that sc
  };
}
const hi = makeGreeter("Hi");
const yo = makeGreeter("Yo");
hi("Ada");   // "Hi, Ada!"
yo("Ada");   // "Yo, Ada!"
```

Each call to `makeGreeter` creates a new scope with its own `greeting`; the returned function holds a reference to *that* scope. Two calls, two independent closures. The inner function does not copy `greeting` — it references the live binding, so if the outer scope later changes the variable, the closure sees the change:

```
function counter() {
  let n = 0;
  return { inc: () => ++n, get: () => n };    // two closures over the same n
}
const c = counter();
c.inc(); c.inc(); c.get();   // 2 — inc and get share one live n
```

Private state

Before `#private` fields existed, this was *the* way to hide data, and it remains the lightest:

```
function makeAccount(opening) {
  let balance = opening;                                  // unreachable from ou
  return {
    deposit(amount) { if (amount > 0) balance += amount; return balance; },
    withdraw(amount) { if (amount ⩽ balance) balance -= amount; return balanc
    balance: () ⇒ balance,
  };
}
const acct = makeAccount(100);
acct.deposit(50);          // 150
acct.balance;              // a function, not the variable — the variable is r
```

Nothing can reach `balance` except through the methods. The pattern is called the **module pattern** (or a factory function) and it scales from a counter to a whole library.

Configuration and partial application

```
const multiplier = (k) ⇒ (x) ⇒ x * k;
const double = multiplier(2), triple = multiplier(3);
[1, 2, 3].map(double);                // [2, 4, 6]

const logAt = (level) ⇒ (msg) ⇒ console.log(`[${level}] ${msg}`);
const warn = logAt("WARN");
```

A function that takes some arguments now and returns a function that takes the rest is **partial application**; when every argument is taken one at a time it is **currying**. Both are closures over the earlier arguments. `bind` (next lesson) does the same for a fixed `this` and leading arguments.

Memoisation

```
function memoize(f) {
  const cache = new Map();
  return (x) => {
    if (!cache.has(x)) cache.set(x, f(x));
    return cache.get(x);
  };
}
const slowSquare = (n) => { /* expensive */ return n * n; };
const fastSquare = memoize(slowSquare);
fastSquare(9); fastSquare(9);      // computed once
```

The `cache` lives in the closure — private, per-memoised-function, alive as long as `fastSquare` is. For several arguments use a key like `JSON.stringify(args)`; for recursion, memoise the recursive function itself so inner calls hit the cache.

Pitfall one: the loop variable

```
const fns = [];
for (var i = 0; i < 3; i++) fns.push(() => i);
fns.map((f) => f());        // [3, 3, 3]
```

All three closures capture the **same** `i` (one `var` binding per function), read after the loop finished. Fixes, oldest to newest: an IIFE per iteration `((j) => fns.push(() => j))(i)`; `forEach`, whose callback gets a fresh parameter each time; or simply `let`, which gives every iteration its own binding — `[0, 1, 2]`. In modern code the bug cannot happen unless you write `var`.

Pitfall two: memory

A closure keeps its whole captured scope alive, not just the variables it uses (engines optimise this partly, but do not rely on it). A short callback that closes over a scope holding a large array keeps the array in memory as long as the callback is referenced — by an event emitter, a timer, a cache. The leak shape is "a long-lived thing holds a small function that holds a big scope". Fix by not capturing what you do not need (pass values as parameters), by clearing references when done, and by removing listeners.

Closures and asynchrony

Every callback you hand to `setTimeout`, `fs.readFile`, a promise `then` or an event listener is a closure — it runs later, and what it sees is the scope it was written in:

```
function fetchAndLabel(id) {
  const label = `item-${id}`;
  setTimeout(() => console.log(label), 100);   // label still available 100 ms
}
```

This is the reason asynchronous JavaScript works at all without passing state around by hand.

Seeing closures in the debugger

Node's inspector and browser devtools show a paused function's **Closure** scope alongside Local and Global — the exact set of outer variables it captured. Watching that panel once makes the concept permanent.

Interview angle

"What is a closure?" A function plus the lexical scope it was created in; the function can read and write those variables even after the outer function returned.

"Give a practical use." Private state (a counter or account factory), memoisation, configured functions (`multiplier(2)`), every asynchronous callback.

"Why does `for (var …)` with callbacks print the last value?" One shared binding captured by every closure; `let` gives one per iteration.

"Do closures capture values or variables?" Variables (live bindings): a later change in the outer scope is visible inside.

"Can closures leak memory?" Yes — a retained callback retains its whole scope; drop references and remove listeners.

Key takeaways

A closure references the live scope where it was defined; each outer call creates a fresh scope.

Factories with returned methods give private state; configured functions and memoisation are closures over arguments and caches.

`var` in a loop shares one binding — use `let`.

Closures keep scopes alive: mind what long-lived callbacks capture.

Every asynchronous callback is a closure; that is what makes async code carry context for free.

EXERCISE 1

A counter factory

Write `makeCounter(start = 0)` returning `{ inc, dec, get }` over a **private** `count` — no property holds it. Read `n` commands: `new <name> <start>`, `inc <name>`, `dec <name>`, `get <name>` (prints `<name>=<count>`). Keep the counters in a `Map`. Finish with `counters=<how many>`. Two counters never interfere: each call to the factory makes a fresh scope.

Example: `7` then `new a 5`, `inc a`, `inc a`, `get a`, `new b 0`, `dec b`, `get b` →

```
a=7
b=-1
counters=2
```

INPUT 1

```
7
new a 5
inc a
inc a
get a
new b 0
dec b
get b
```

OUTPUT 1

```
a=7
b=-1
counters=2
```

INPUT 2

```
3
new x 0
get x
get x
```

OUTPUT 2

```
x=0
x=0
counters=1
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
function makeCounter(start = 0) {
  let count = start;
  return {
    inc: () => ++count,
    dec: () => --count,
    get: () => count,
  };
}
let pos = 0;
const next = () => tokens[pos++];
const n = Number(next());
const counters = new Map();
for (let i = 0; i < n; i++) {
  const cmd = next(), name = next();
  if (cmd === "new") counters.set(name, makeCounter(Number(next())));
  else if (cmd === "inc") counters.get(name).inc();
  else if (cmd === "dec") counters.get(name).dec();
  else console.log(`${name}=${counters.get(name).get()}`);
}
console.log(`counters=${counters.size}`);
```

EXERCISE 2

Memoise, and count the hits

Write `memoize(f)` returning a function that caches results by argument in a closure-held `Map` and exposes `hits` and `misses` counters as properties on the returned function. Memoise the provided `collatzSteps`. Read `n` integers; print `steps(<x>)=<result>` for each, then `computed=<misses> cached=<hits>`.

Example: `5` then `6 6 27 6 27` →

```
steps(6)=8
steps(6)=8
steps(27)=111
steps(6)=8
steps(27)=111
computed=2 cached=3
```

INPUT 1

```
5
6 6 27 6 27
```

OUTPUT 1

```
steps(6)=8
steps(6)=8
steps(27)=111
steps(6)=8
steps(27)=111
computed=2 cached=3
```

INPUT 2

```
1
1
```

OUTPUT 2

```
steps(1)=0
computed=1 cached=0
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
function memoize(f) {
  const cache = new Map();
  const wrapped = (x) => {
    if (cache.has(x)) { wrapped.hits++; return cache.get(x); }
    wrapped.misses++;
    const value = f(x);
    cache.set(x, value);
    return value;
  };
  wrapped.hits = 0;
  wrapped.misses = 0;
  return wrapped;
}
const collatzSteps = (x) => { let s = 0; while (x !== 1) { x = x % 2 === 0 ? x / 2 :
const steps = memoize(collatzSteps);
const n = Number(tokens[0]);
for (let i = 1; i <= n; i++) {
  const x = Number(tokens[i]);
  console.log(`steps(${x})=${steps(x)}`);
}
console.log(`computed=${steps.misses} cached=${steps.hits}`);
```

QUIZ

1. A closure is…

A. A function with no parameters

B. A function together with the lexical scope it was created in, still usable after that scope's function returned

C. A sealed object

D. An IIFE

2. Closures capture…

A. Copies of values at creation time

B. Live variable bindings — later changes to the outer variable are visible inside

C. Only constants

D. Only parameters

3. `for (var i …) fns.push(() ⇒ i)` yields identical results because…

A. Arrows cannot close over loop variables

B. There is one `var i` binding shared by every closure, read after the loop ended

C. `push` copies the function

D. `i` is a constant

4. Private state before `#fields` was done with…

A. `private` keyword

B. A factory function whose returned methods close over local variables

C. `Object.freeze`

D. Underscored property names

5. A memory leak through a closure happens when…

A. The closure is called twice

B. A long-lived reference (listener, timer, cache) holds a callback that holds a large captured scope

C. Closures are used in loops

D. The closure returns an object

ANSWER KEY · CLOSURES — FUNCTIONS THAT REMEMBER

1. B Every function is technically one; the term matters when the scope outlives its call.

2. B Two closures over the same `n` (an `inc` and a `get`) see the same value.

3. B `let` creates a binding per iteration and fixes it.

4. B The module pattern; nothing outside can reach the variable.

5. B Capture only what you need and remove listeners when done.

LESSON 4 · 15 MIN

this — the five binding rules, call, apply and bind

`this` is the most misunderstood word in JavaScript because it is the one thing *not* decided lexically: its value depends on **how a function is called**, not where it was written — with the exception of arrow functions, which have no `this` of their own and borrow the enclosing one. There are exactly five rules, applied in order of precedence, and once you know them every "why is `this` undefined in my callback" has a one-line answer. This lesson states the rules, shows the classic loss of `this`, and covers the three methods — `call`, `apply`, `bind` — that set it explicitly.

The five rules, highest precedence first

Arrow function — no own `this`; uses the `this` of the enclosing scope at the time the arrow was *created*. Nothing can change it, not even `bind`.

`new` — `new F()` creates a fresh object and binds `this` to it for the duration of `F`.

Explicit binding — `f.call(obj, …)`, `f.apply(obj, […])`, and a function produced by `f.bind(obj)`: `this` is `obj`.

Method call — `obj.method()`: `this` is the object before the dot (the *receiver*), whatever object that is at the call.

Plain call — `f()`: `this` is `undefined` in strict mode (the global object in sloppy mode).

```
"use strict";
const user = {
  name: "Ada",
  hello() { return `hi from ${this.name}`; },
  later() { setTimeout(function () { console.log(this); }, 0); }        // pl
};
user.hello();                 // "hi from Ada"        — rule 4
const f = user.hello;
f();                          // TypeError: Cannot read properties of undefine
f.call({ name: "Bo" });       // "hi from Bo"         — rule 3
```

The dot decides. Detach the method from the object and the receiver is gone.

Losing `this`, and the three fixes

The classic: passing a method as a callback.

```
class Counter {
  count = 0;
  increment() { this.count++; }
}
const c = new Counter();
setTimeout(c.increment, 0);          // called as a plain function → this is u
button.addEventListener("click", c.increment);   // this would be the button,
```

Fixes, in order of preference:

```
setTimeout(() => c.increment(), 0);              // 1. wrap in an arrow: the c
setTimeout(c.increment.bind(c), 0);              // 2. bind: a new function wi
class Counter { increment = () => { this.count++; }; }   // 3. arrow class fie
```

The arrow class field (rule 1 inside a constructor context) is the modern idiom for methods used as callbacks — React components used it for years. Its cost: one function object per instance instead of one shared method on the prototype.

`call`, `apply`, `bind`

```
function describe(prefix, suffix) { return `${prefix}${this.name}${suffix}`; }
const cat = { name: "Tom" };
describe.call(cat, "<", ">");          // "<Tom>"   — arguments listed
describe.apply(cat, ["<", ">"]);       // "<Tom>"   — arguments as an array (sp
const tagTom = describe.bind(cat, "["); // this = cat, prefix = "[" fixed
tagTom("]");                           // "[Tom]"
```

`bind` returns a **new** function with `this` (and optionally leading arguments — partial application) permanently set; calling `bind` again on it does not rebind. `call` / `apply` invoke immediately. Modern uses: `bind` for callbacks and event handlers, `call` for borrowing methods (`Array.prototype.slice.call(arguments)` in old code; `Object.prototype.hasOwnProperty.call(obj, key)` still common), `apply` almost never now that `Math.max(...arr)` exists.

`this` in arrows, carefully

An arrow's `this` is whatever `this` was where the arrow was written:

```
const obj = {
  name: "X",
  regular() { return [1].map(function () { return this; }); },    // [undefine
  arrow() { return [1].map(() ⇒ this.name); },                    // ["X"] — ar
  wrong: () ⇒ this,                                               // arrow AT
};
```

Rule of thumb: **arrows for callbacks inside methods; regular functions (or method shorthand) for the methods themselves.** An arrow defined as an object-literal property does not see the object.

`this` at the top level and in modules

In a CommonJS file, top-level `this` is `module.exports`; in an ES module it is `undefined`; in a browser script it is `window`. Inside a plain function in strict mode it is `undefined`. None of these are useful — treat top-level `this` as something never to reference.

`this` in classes and constructors

```
function Person(name) { this.name = name; }            // rule 2 with new: th
const p = new Person("Ada");
Person("Bo");                                          // rule 5 without new:

class Point { constructor(x, y) { this.x = x; this.y = y; } dist() { return Ma
```

Class constructors throw if called without `new` — one of the reasons `class` replaced constructor functions. Inside methods, rule 4 applies as usual, and losing `this` when passing `p.dist` around is the same problem with the same three fixes.

`globalThis`, and why `this` is not for globals

If you need the global object, write `globalThis`; never `this` at top level (it differs by environment). The sloppy-mode rule that made plain-call `this` the global object is the reason old code accidentally created global variables from inside functions.

A method for reading any `this`

Is the function an arrow? Take the enclosing scope's `this` at definition.

Was it called with `new`? The new object.

`call` / `apply` / `bind`? The given object.

Called as `something.f()`? `something`.

Otherwise `undefined` (strict).

Run the list on any snippet and the answer falls out.

Interview angle

"How is `this` determined?" By the call: `new` → new object; explicit `call` / `apply` / `bind` → given object; method call → receiver; plain call → `undefined` in strict mode; arrows inherit lexically and cannot be rebound.

"Why is `this` undefined in my callback?" The method was passed and later called as a plain function; wrap in an arrow, `bind` it, or use an arrow class field.

"`call` versus `apply` versus `bind`?" Immediate with listed args; immediate with an array; returns a new bound function.

"Can you `bind` an arrow function?" No effect on `this` — arrows have none of their own.

"What is `this` in an arrow defined inside a method?" The method's `this` — that is the whole point of using an arrow there.

Key takeaways

Five rules by precedence: arrow (lexical) → `new` → `call` / `apply` / `bind` → receiver of the dot → `undefined`.

Detaching a method loses its receiver; fix with an arrow wrapper, `bind`, or an arrow class field.

`bind` fixes `this` (and leading arguments) permanently; `call` / `apply` invoke now.

Arrows inside methods are right; arrows *as* methods are wrong.

Never rely on top-level `this`; use `globalThis` for the global object.

EXERCISE 1

Which rule decided?

Given `ada = { name: "Ada", who() {…}, viaArrow() {…} }` (`who` returns `this.name`, or `"undefined"` when `this` is undefined; `viaArrow` maps over `[1]` with an arrow returning `this.name`), `bo = { name: "Bo" }` and a constructor `Thing(name)`, read `n` call kinds and print `<kind>: <result>`: `method` (`ada.who()`), `detached` (`const f = ada.who; f()`), `bound` (`ada.who.bind(bo)()`), `call` (`ada.who.call({ name: "Cy" })`), `arrow-in-method` (`ada.viaArrow()`), `new` (`new Thing("instance").name`).

Example: `3` then `method detached bound` →

```
method: Ada
detached: undefined
bound: Bo
```

INPUT 1

```
3
method detached bound
```

OUTPUT 1

```
method: Ada
detached: undefined
bound: Bo
```

INPUT 2

```
3
call arrow-in-method new
```

OUTPUT 2

```
call: Cy
arrow-in-method: Ada
new: instance
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const ada = { name: "Ada", who() { return this === undefined ? "undefined" : this.na
const bo = { name: "Bo" };
function Thing(name) { this.name = name; }
const n = Number(tokens[0]);
for (let i = 1; i <= n; i++) {
  const kind = tokens[i];
  let result;
  switch (kind) {
    case "method": result = ada.who(); break;                          // receiver of t
    case "detached": { const f = ada.who; result = f(); break; }       // plain call: t
    case "bound": result = ada.who.bind(bo)(); break;                   // explicit bir
    case "call": result = ada.who.call({ name: "Cy" }); break;
    case "arrow-in-method": result = ada.viaArrow(); break;             // arrow borrow
    default: result = new Thing("instance").name;                       // new: a fresh
  }
  console.log(`${kind}: ${result}`);
}
```

EXERCISE 2

Fix the callback, three ways

A `Counter` class has `count`, a method `increment()` and an arrow field `incrementArrow`. Read `n`. Register `n` callbacks in a `later` array in four ways — the **broken** detached method `broken.increment`, an arrow wrapper `() => wrapped.increment()`, a bound copy `bound.increment.bind(bound)`, and the arrow field `field.incrementArrow` — each on its own `Counter` instance. Then run every callback inside `try` / `catch`, counting the errors, and print `broken=<count> errors=<n>` and `wrapped=<> bound=<> field=<>`.

Example: `3` →

```
broken=0 errors=3
wrapped=3 bound=3 field=3
```

INPUT 1

```
3
```

OUTPUT 1

```
broken=0 errors=3
wrapped=3 bound=3 field=3
```

INPUT 2

```
1
```

OUTPUT 2

```
broken=0 errors=1
wrapped=1 bound=1 field=1
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
class Counter {
  count = 0;
  increment() { this.count++; }
  incrementArrow = () => { this.count++; };
}
const n = Number(tokens[0]);
const later = [];
const broken = new Counter(), wrapped = new Counter(), bound = new Counter(), field
for (let i = 0; i < n; i++) {
  later.push(broken.increment);            // detached: this will be undefined
  later.push(() => wrapped.increment());   // arrow wrapper keeps the dot call
  later.push(bound.increment.bind(bound)); // bound copy
  later.push(field.incrementArrow);        // arrow class field: this fixed at const
}
let errors = 0;
for (const cb of later) {
  try { cb(); } catch (e) { errors++; }
}
console.log(`broken=${broken.count} errors=${errors}`);
console.log(`wrapped=${wrapped.count} bound=${bound.count} field=${field.count}`);
```

QUIZ

1. In strict mode, `const f = obj.method; f()` runs with `this` equal to…

A. `obj`

B. `undefined` — a plain call loses the receiver

C. The global object

D. `f`

2. An arrow function's `this` is…

A. The object it is stored on

B. The `this` of the enclosing scope at creation — and `bind` cannot change it

C. Always `undefined`

D. Decided at the call

3. `f.bind(a).bind(b)()` uses `this` equal to…

A. `b`

B. `a` — a bound function cannot be rebound

C. `undefined`

D. Both

4. `call` versus `apply`:

A. `apply` is faster

B. `call` takes listed arguments, `apply` takes an array — spread made `apply` mostly redundant

C. `call` returns a function

D. They differ in `this`

5. Predict: `const o = { n: 1, f: () ⇒ this.n }; o.f()`

A. `1`

B. `undefined` (or a `TypeError`) — an arrow at object-literal level does not see `o`

C. `o`

D. A syntax error

ANSWER KEY · THIS — THE FIVE BINDING RULES, CALL, APPLY AND BIND

1. B Wrap in an arrow, `bind`, or use an arrow class field.

2. B Right for callbacks inside methods; wrong for methods themselves.

3. B `call` on a bound function is likewise ignored for `this`.

4. B `bind` is the one that returns a new function instead of invoking.

5. B Use method shorthand `f() { return this.n; }` for methods.

LESSON 5 · 13 MIN

Higher-order functions — composition, currying and callbacks

Once functions are values, a whole style of programming opens: build small functions, then combine them with other functions. `map`, `filter` and `reduce` are the everyday examples; `compose`, `pipe`, `curry`, `debounce` and `once` are the ones you write yourself. This lesson is that style — how to think in functions that take and return functions, the idioms that recur across every codebase, and the callback convention that predates promises and still underlies Node.

The array trio, as higher-order functions

```
const prices = [5, 12, 30];
prices.map((p) ⇒ p * 1.2);                    // transform each: [6, 14.4, 3
prices.filter((p) ⇒ p > 10);                  // keep some: [12, 30]
prices.reduce((sum, p) ⇒ sum + p, 0);         // fold to one value: 47
prices.some((p) ⇒ p > 20); prices.every((p) ⇒ p > 0);
prices.find((p) ⇒ p > 10); prices.findIndex((p) ⇒ p > 10);
```

Each takes a function and applies it for you — the loop is inside the method. The callback receives `(element, index, array)`; using only the first argument is normal. Pass **named functions** for anything longer than one expression: `orders.filter(isPaid).map(toSummary)` reads as a sentence. Module 4 covers the toolkit in depth; here the point is the shape: *a function that takes a function*.

Functions that return functions

```
const greaterThan = (n) ⇒ (x) ⇒ x > n;
[1, 5, 10].filter(greaterThan(4));            // [5, 10]

const pluck = (key) ⇒ (obj) ⇒ obj[key];
users.map(pluck("email"));

const once = (f) ⇒ { let done = false, result; return (...a) ⇒ done ? result
const init = once(() ⇒ expensiveSetup());     // runs at most once
```

A function returning a configured function is the closure pattern from two lessons ago, used as a **factory of behaviour**. It turns "a comparison" or "a property access" into a value you can hand to `filter` or `map`.

Composition: `compose` and `pipe`

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

const trim = (s) ⇒ s.trim();
const lower = (s) ⇒ s.toLowerCase();
const slug = (s) ⇒ s.replace(/\s+/g, "-");
const slugify = pipe(trim, lower, slug);
slugify("  Hello World ");                    // "hello-world"
```

`pipe(f, g, h)(x)` is `h(g(f(x)))`. Each step is a small pure function you can test alone; the pipeline names the whole transformation. Prefer `pipe` in application code — people read left to right.

Currying and partial application

```
const add = (a) ⇒ (b) ⇒ a + b;              // curried by hand
add(2)(3);                                  // 5
const add2 = add(2);

const curry = (f) ⇒ function curried(...args) {
  return args.length ⩾ f.length ? f(...args) : (...more) ⇒ curried(...args,
};
const volume = curry((l, w, h) ⇒ l * w * h);
volume(2)(3)(4); volume(2, 3)(4); volume(2, 3, 4);   // 24 each

const partial = (f, ...fixed) ⇒ (...rest) ⇒ f(...fixed, ...rest);
const greet = (greeting, name) ⇒ `${greeting}, ${name}`;
const hello = partial(greet, "Hello");          // hello("Ada")
```

Currying turns an n-argument function into n one-argument functions; **partial application** fixes some arguments now. Both let you create specialised functions from general ones

without writing wrappers by hand — and both depend on `f.length`, which counts declared parameters (defaults and rest excluded).

Timing wrappers: `debounce` and `throttle`

```
const debounce = (f, ms) => {
  let timer;
  return (...args) => { clearTimeout(timer); timer = setTimeout(() => f(...arg
};
const throttle = (f, ms) => {
  let last = 0;
  return (...args) => { const now = Date.now(); if (now - last >= ms) { last =
};
```

`debounce` runs `f` only after calls stop for `ms` (search-as-you-type); `throttle` runs at most once per `ms` (scroll handlers). Both are closures over a timer or a timestamp — higher-order functions solving an asynchronous problem with no library.

The callback convention

Before promises, Node expressed "later" with callbacks and one rule: **error first**.

```
fs.readFile("data.txt", "utf8", (err, text) => {
  if (err) return handle(err);
  use(text);
});
```

The callback is the last argument; its first parameter is an error or `null`; you check it first and return. Nested callbacks produced the "pyramid of doom" that promises (module 8) fixed, but the convention survives in Node's APIs, event emitters and many libraries — `util.promisify` converts an error-first function into a promise-returning one. Two rules when writing your own: call the callback **exactly once**, and call it **asynchronously** even if the result is ready, so callers cannot observe two different orders.

Higher-order pitfalls

Passing methods: `arr.map(obj.method)` loses `this` (previous lesson); pass `(x) ⇒ obj.method(x)`.

`parseInt` in `map`: `["1", "2", "3"].map(parseInt)` gives `[1, NaN, NaN]` because `map` passes the index as the radix. `.map(Number)` or `.map((s) ⇒ parseInt(s, 10))`.

Side effects in `map`: `map` that ignores its result should be `forEach`; `forEach` that builds a result should be `map` or `reduce`. Match the method to the intent.

Over-abstraction: a `pipe` of eight one-liners can be less readable than a ten-line function. Compose when the pieces have names that mean something on their own.

Interview angle

"What is a higher-order function?" One that takes a function as an argument or returns one — `map`, `filter`, `compose`, `debounce`.

"Currying versus partial application?" Currying: one argument at a time, returning functions until all are supplied. Partial: fix some arguments now, take the rest later.

"Implement `compose` / `pipe`." `reduceRight` / `reduce` over the functions, starting from the input value.

"Why does `[\"1\",\"2\"].map(parseInt)` misbehave?" `map` passes `(element, index)`; `parseInt` reads the index as the radix.

"What is the error-first callback convention?" Callback last; first parameter an error or `null`; check it and return before using the result.

Key takeaways

Functions in, functions out: `map` / `filter` / `reduce` for data; factories (`greaterThan(n)`) for behaviour.

`pipe` (left to right) and `compose` (right to left) chain unary functions; name the steps.

Curry for one-at-a-time specialisation, partial for fixing leading arguments, `once` / `debounce` / `throttle` for call control — all closures.

Error-first callbacks: last argument, `(err, result)`, check `err` first, call exactly once and asynchronously.

`.map(parseInt)` and passing unbound methods are the two classic traps.

EXERCISE 1

pipe and compose

Implement `pipe(...fns)` (left to right, with `reduce`) and `compose(...fns)` (right to left, with `reduceRight`) over unary functions. Read `k` step names from `inc`, `double`, `square`, `neg`, then `m` numbers; for each number print `pipe(<steps>)(<x>)=<…> compose(<steps>)(<x>)=<…>` with the steps comma-joined.

Example: `2 inc double` then `3` then `1 5 -2` →

```
pipe(inc,double)(1)=4 compose(inc,double)(1)=3
pipe(inc,double)(5)=12 compose(inc,double)(5)=11
pipe(inc,double)(-2)=-2 compose(inc,double)(-2)=-3
```

INPUT 1

```
2 inc double
3
1 5 -2
```

OUTPUT 1

```
pipe(inc,double)(1)=4
compose(inc,double)(1)=3
pipe(inc,double)(5)=12
compose(inc,double)(5)=11
pipe(inc,double)(-2)=-2
compose(inc,double)(-2)=-3
```

INPUT 2

```
1 square
1
-3
```

OUTPUT 2

```
pipe(square)(-3)=9 compose(square)(-3)=9
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const steps = { inc: (x) ⇒ x + 1, double: (x) ⇒ x * 2, square: (x) ⇒ x * x, neg:
const pipe = (...fns) ⇒ (x) ⇒ fns.reduce((acc, f) ⇒ f(acc), x);
const compose = (...fns) ⇒ (x) ⇒ fns.reduceRight((acc, f) ⇒ f(acc), x);
let pos = 0;
const next = () ⇒ tokens[pos++];
const k = Number(next());
const names = [];
for (let i = 0; i < k; i++) names.push(next());
const fns = names.map((s) ⇒ steps[s]);
const m = Number(next());
for (let i = 0; i < m; i++) {
  const x = Number(next());
  console.log(`pipe(${names.join(",")})(${x})=${pipe(...fns)(x)} compose(${names.joi
}
```

EXERCISE 2

curry and partial

Implement a generic `curry(f)` that keeps collecting arguments until `f.length` are present, then calls `f` — so `volume(2)(3)(4)`, `volume(2, 3)(4)` and `volume(2, 3, 4)` all work. `partial(f, ...fixed)` is given. Read `n` lines `l w h name` and print `volume=<a>,<b>,<c> same=<all equal> <hello(name)>` where `hello = partial(greet, "Hello")`.

Example: `2` then `2 3 4 Ada`, `1 1 1 Bo` →

```
volume=24,24,24 same=true Hello, Ada
volume=1,1,1 same=true Hello, Bo
```

INPUT 1

```
2
2 3 4 Ada
1 1 1 Bo
```

OUTPUT 1

```
volume=24,24,24 same=true Hello, Ada
volume=1,1,1 same=true Hello, Bo
```

INPUT 2

```
1
0 9 9 Cy
```

OUTPUT 2

```
volume=0,0,0 same=true Hello, Cy
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const curry = (f) => function curried(...args) {
  return args.length >= f.length ? f(...args) : (...more) => curried(...args, ...mor
};
const partial = (f, ...fixed) => (...rest) => f(...fixed, ...rest);
const volume = curry((l, w, h) => l * w * h);
const greet = (greeting, name) => `${greeting}, ${name}`;
const hello = partial(greet, "Hello");
const n = Number(tokens[0]);
for (let i = 0; i < n; i++) {
  const l = Number(tokens[1 + 4 * i]), w = Number(tokens[2 + 4 * i]), h = Number(tok
  const a = volume(l)(w)(h), b = volume(l, w)(h), c = volume(l, w, h);
  console.log(`volume=${a},${b},${c} same=${a === b && b === c} ${hello(name)}`);
}
```

QUIZ

1. A higher-order function…

A. Is asynchronous

B. Takes a function as an argument or returns one

C. Has more than two parameters

D. Is recursive

2. `pipe(f, g, h)(x)` equals…

A. `f(g(h(x)))`

B. `h(g(f(x)))`

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

D. `[f, g, h]`

3. `["1", "2", "3"].map(parseInt)` gives…

A. `[1, 2, 3]`

B. `[1, NaN, NaN]` — `map` passes the index as the radix

C. `["1", "2", "3"]`

D. Throws

4. Currying versus partial application:

A. Same thing

B. Currying: one argument per call until all arrive; partial: fix some arguments now, take the rest later

C. Currying needs `bind`

D. Partial is recursive

5. In Node's error-first callback convention…

A. The callback is the first argument

B. The callback is last and receives `(err, result)`; check `err` first and return

C. Errors are thrown

D. Results come first

ANSWER KEY · HIGHER-ORDER FUNCTIONS — COMPOSITION, CURRYING AND CALLBACKS

1. B `map`, `filter`, `compose`, `debounce` — and your own factories.

2. B Left to right; `compose` is the reverse.

3. B `.map(Number)` or `.map((s) ⇒ parseInt(s, 10))`.

4. B Both are closures over the arguments supplied so far.

5. B Call it exactly once and asynchronously; `util.promisify` converts to a promise.

CHECKPOINT · 24 MIN

Checkpoint — Functions, scope and closures

This checkpoint covers the three ways to define a function and how they hoist, parameters with defaults, rest and spread, lexical scope and the scope chain, closures and their patterns and pitfalls, the five rules for `this` with `call` / `apply` / `bind`, and higher-order functions — composition, currying and the callback convention.

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:

- Why a function declaration can be called above its line and a `const` arrow cannot.
- When a default parameter applies, and the difference between rest and spread.
- What a closure captures (variables, not values) and why `for (var …)` callbacks all see the last value.
- The five `this` rules in precedence order, and the three ways to fix a detached method.
- What `bind` returns and why it cannot rebind an arrow.
- How `pipe` and `compose` differ, and why `["1","2"].map(parseInt)` fails.

The programs are a counter factory with private state and an audit log, a `this`-binding tracer that prints which rule decided each call, and a small function-pipeline interpreter built from `pipe`, `curry` and `once`.

EXERCISE 1

A counter with an audit log

Write `makeCounter()` with private `value` and `history` and methods `inc(by)`, `dec(by)`, `undo()` (reverts the last change, returning `undid <entry>` or `nothing to undo`), `get()` and `log()` (the history entries like `+5 -2` space-joined, or `(empty)`). Read `n` commands `inc <k>`, `dec <k>`, `undo`, `get` (prints `value=<v>`), `log` (prints `log=<entries>`).

Example: `7` then `inc 5`, `inc 2`, `get`, `undo`, `get`, `log`, `undo` →

```
value=7
undid +2
value=5
log=+5
undid +5
```

INPUT 1

```
7
inc 5
inc 2
get
undo
get
log
undo
```

OUTPUT 1

```
value=7
undid +2
value=5
log=+5
undid +5
```

INPUT 2

```
2
undo
log
```

OUTPUT 2

```
nothing to undo
log=(empty)
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
function makeCounter() {
  let value = 0;
  const history = [];
  return {
    inc: (by = 1) => { value += by; history.push(`+${by}`); },
    dec: (by = 1) => { value -= by; history.push(`-${by}`); },
    undo: () => {
      const last = history.pop();
      if (last === undefined) return "nothing to undo";
      value -= Number(last);
      return `undid ${last}`;
    },
    get: () => value,
    log: () => (history.length ? history.join(" ") : "(empty)"),
  };
}
let pos = 0;
const next = () => tokens[pos++];
const n = Number(next());
const c = makeCounter();
for (let i = 0; i < n; i++) {
  const cmd = next();
  if (cmd === "inc") c.inc(Number(next()));
  else if (cmd === "dec") c.dec(Number(next()));
  else if (cmd === "undo") console.log(c.undo());
  else if (cmd === "get") console.log(`value=${c.get()}`);
  else console.log(`log=${c.log()}`);
}
```

EXERCISE 2

Binding precedence

Read three names and make objects `a`, `b`, `c` with those names. With `function who() { return this === undefined ? "undefined" : this.name; }`, print four lines: `bindTwice=<who.bind(a).bind(b)()>` (the first bind wins), `callOnBound=<who.bind(a).call(c)>` (call cannot override), `arrowIgnoresBind=<an arrow returning this.name, bound to a, then called>` — at the top level of a CommonJS file `this` is `module.exports`, whose `name` is `undefined` — and `newOverridesBind=<new (Named.bind(b))(c.name).name>` where `function Named(name) { this.name = name; }`.

Example: `3 Ada Bo Cy` →

```
bindTwice=Ada
callOnBound=Ada
arrowIgnoresBind=undefined
newOverridesBind=Cy
```

INPUT 1

```
3 Ada Bo Cy
```

OUTPUT 1

```
bindTwice=Ada
callOnBound=Ada
arrowIgnoresBind=undefined
newOverridesBind=Cy
```

INPUT 2

```
3 x y z
```

OUTPUT 2

```
bindTwice=x
callOnBound=x
arrowIgnoresBind=undefined
newOverridesBind=z
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const [a, b, c] = tokens.slice(1, 4).map((name) => ({ name }));
function who() { return this === undefined ? "undefined" : this.name; }
const boundA = who.bind(a);
console.log(`bindTwice=${boundA.bind(b)()}`);            // the first bind wins
console.log(`callOnBound=${boundA.call(c)}`);            // call cannot override a b
const arrow = () => (this === undefined ? "undefined" : this.name);
console.log(`arrowIgnoresBind=${arrow.bind(a)()}`);      // arrows have no this of t
function Named(name) { this.name = name; }
const BoundNamed = Named.bind(b);
console.log(`newOverridesBind=${new BoundNamed(c.name).name}`);   // new beats bind:
```

EXERCISE 3

A pipeline interpreter

The first line names steps, each `name` or `name:arg`, from a registry of step **factories**: `inc`, `double`, `add:k`, `mul:k`, and `tap` (passes the value through and increments a closure-held counter). The second line holds numbers. Build each step by calling its factory with the argument, `pipe` them, apply the pipeline to every number printing `<x> → <result>`, then print `tapped=<how many times tap ran>`.

Example: `inc tap double add:3 tap` / `1 2 3` →

```
1 → 7
2 → 9
3 → 11
tapped=6
```

INPUT 1

```
inc tap double add:3 tap
1 2 3
```

OUTPUT 1

```
1 → 7
2 → 9
3 → 11
tapped=6
```

INPUT 2

```
mul:10
0 5
```

OUTPUT 2

```
0 → 0
5 → 50
tapped=0
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const stepNames = lines[0].trim().split(/\s+/);
const values = lines[1].trim().split(/\s+/).map(Number);
let tapped = 0;
const registry = {
  inc: () => (x) => x + 1,
  double: () => (x) => x * 2,
  add: (k) => (x) => x + Number(k),
  mul: (k) => (x) => x * Number(k),
  tap: () => (x) => { tapped++; return x; },
};
const pipe = (...fns) => (x) => fns.reduce((acc, f) => f(acc), x);
const fns = stepNames.map((s) => {
  const [name, arg] = s.split(":");
  return registry[name](arg);
});
const run = pipe(...fns);
for (const v of values) console.log(`${v} → ${run(v)}`);
console.log(`tapped=${tapped}`);
```

CHECKPOINT QUESTIONS

1. Which form is hoisted with its body?

A. Arrow function

B. Function expression

C. Function declaration

D. Method shorthand

2. A default parameter is used when the argument is…

A. Falsy

B. `undefined`

C. `null`

D. Missing only, not explicit `undefined`

3. Spread versus rest:

A. Same syntax, same meaning

B. Rest gathers remaining parameters in a signature; spread expands an iterable at a call or literal

C. Rest is for arrays only

D. Spread is for objects only

4. Where does a name lookup end?

A. At the function

B. At the global scope — then `ReferenceError` on read

C. At the nearest block

D. At the module only

5. Closures capture…

A. Values

B. Live bindings

C. Types

D. Copies of objects

6. `for (let i …)` fixes the loop-closure bug because…

A. `let` is faster

B. Each iteration gets its own `i` binding

C. Closures cannot capture `let`

D. `let` copies the value

7. Highest precedence among the `this` rules:

A. Method call

B. `bind`

C. Arrow function (lexical `this`)

D. Plain call

8. `obj.method` passed to `setTimeout` runs with `this`…

A. `obj`

B. `undefined` in strict mode — fix with an arrow, `bind`, or an arrow field

C. The timer

D. `globalThis` always

9. `bind` returns…

A. The result of calling `f`

B. A new function with `this` (and optional leading arguments) fixed

C. `this`

D. A promise

10. `compose(f, g)(x)` is…

A. `g(f(x))`

B. `f(g(x))`

C. `f(x)(g)`

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

11. Memoisation stores…

A. The function's source

B. Previous results keyed by argument, in a closure-held cache

C. The call stack

D. Global variables

12. An error-first callback should be called…

A. Twice, once with the error and once with the result

B. Exactly once, asynchronously, with `(err, result)`

C. Synchronously when possible

D. Only on success

ANSWER KEY · CHECKPOINT — FUNCTIONS, SCOPE AND CLOSURES

1. C The others are values available only after their line.

2. B `null` is passed through.

3. B Same three dots, opposite directions.

4. B Block → function → module → global, in that order.

5. B Which is why a shared `var` loop variable shows its final value.

6. B One binding per iteration by specification.

7. C Arrow, then `new`, then explicit binding, then the receiver, then `undefined`.

8. B The dot is lost when the method is detached.

9. B `call` / `apply` invoke immediately.

10. B Right to left, like function composition in mathematics.

11. B Use `Map.has` so a falsy cached value still counts as a hit.

12. B Two calls or a sometimes-sync callback are classic bugs.