Module 5 of 16

Prototypes and classes

The prototype chain and delegation; constructor functions and the four steps of new; class syntax with fields, private members, accessors and statics; extends and super; property descriptors and the toString/valueOf/toJSON protocols.

JavaScript's object model is one idea — objects delegate to other objects along a prototype chain — dressed in two syntaxes: the constructor functions everything was built with until 2015, and the `class` keyword that made the same machinery readable and added the parts it lacked (real private members, fields, static blocks). Knowing the chain is what turns `class` from magic into a notation you can predict: why methods are shared, what `instanceof` really tests, why a detached method loses `this`, and what the four steps of `new` are when an interviewer asks you to implement it.

The module opens with the chain itself — `Object.create`, lookup and shadowing, the built-in chains, prototype pollution — then the constructor protocol and its factory-function rival. The class lesson covers the whole surface on Node 16, including `#private` as a brand and field-initialisation order. Inheritance follows: `extends` sets up two chains, `super()` must come first, base constructors must not call overridables, and extending `Error` / `Map` / `Array` each has its rules. The last lesson goes under property access — descriptors, freeze/seal, and the protocols (`toString`, `valueOf`, `Symbol.toPrimitive`, `toJSON`, `Symbol.toStringTag`) that let your class plug into template literals, arithmetic, sorting and JSON.

The exercises build the machinery — a chain resolver, `new` by hand, a bank account with private state, a shape hierarchy, a descriptor report, a `Money` class that sorts and serialises — and end with an inventory hierarchy, an event emitter and a linked list class.

CONTENTS

1	The prototype chain — how JavaScript objects inherit	13 min
2	Constructor functions and what `new` really does	12 min
3	Class syntax — fields, methods, statics, private and accessors	14 min
4	Inheritance — extends, super, overriding and extending built-ins	13 min
5	Property descriptors and the object protocols — toString, valueOf, toJSON, Symbol.toPrimitive	13 min
✓	Checkpoint — Prototypes and classes	24 min

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

LESSON 1 · 13 MIN

The prototype chain — how JavaScript objects inherit

JavaScript has no classes in the Java sense — `class` is syntax over something older and simpler. Every object has a hidden link to another object, its **prototype**; when you read a property the object does not have, the engine follows the link and looks there, then at *that* object's prototype, and so on until it hits `null`. That walk is the whole inheritance system. Methods live once on a shared prototype rather than being copied into every instance, `instanceof` is a question about that chain, and half the "why does `toString` exist on my empty object" surprises are answered by drawing it. This lesson makes the chain explicit: how to read and set it, how lookup and shadowing work, and the built-in chains you are already standing on.

Every object has a prototype

```
const animal = { eats: true, describe() { return `${this.name} eats: ${this.eats
const rabbit = Object.create(animal);          // rabbit's prototype is animal
rabbit.name = "Peter";

rabbit.eats;                    // true — not on rabbit; found on animal
rabbit.describe();              // "Peter eats: true" — this is rabbit, the obje
Object.getPrototypeOf(rabbit) === animal;      // true
Object.getPrototypeOf(animal) === Object.prototype;   // true — literals inherit
Object.getPrototypeOf(Object.prototype);       // null — the end of every chain
```

`Object.create(proto)` makes an empty object whose prototype is `proto`. `Object.getPrototypeOf(obj)` reads the link; `Object.setPrototypeOf(obj, proto)` rewrites it (slow — engines deoptimise objects whose prototype changes after creation; set it at creation instead). The `__proto__` accessor (`rabbit.__proto__`) is the legacy spelling of the same link; read it in the console, do not write it in code.

Lookup and shadowing

Reading `obj.x` checks `obj` 's **own** properties first, then walks up. Writing `obj.x = v` always creates or updates an own property on `obj` — it never modifies the prototype — so a write on an instance **shadows** the inherited value:

```
rabbit.eats = false;     // own property now; animal.eats is still true
delete rabbit.eats;      // the shadow goes; rabbit.eats reads true again from an
```

That asymmetry is why `Object.hasOwn(obj, key)` and `in` give different answers: `"eats" in rabbit` is `true` (found on the chain), `Object.hasOwn(rabbit, "eats")` is `false` until it is shadowed. `for…in` walks inherited *enumerable* properties too — one more reason `Object.keys` (own, enumerable) is the loop you want.

Two subtleties. Getters and setters on the prototype run with `this` set to the receiver — a prototype setter does not create a shadow; it runs. And a method found up the chain is called with `this` being the object you started from, which is exactly what makes shared methods work on per-instance data.

The chains you already use

```
Object.getPrototypeOf([]) === Array.prototype;             // then Object.protot
Object.getPrototypeOf(function () {}) === Function.prototype;
Object.getPrototypeOf("s") === String.prototype;           // primitives are box
Object.getPrototypeOf(Object.create(null));                // null — an object w
```

`[1, 2].map` is found on `Array.prototype`; `"abc".toUpperCase` on `String.prototype` after a temporary box; `{}.toString` on `Object.prototype`. `Object.create(null)` produces a true dictionary with nothing inherited — no `toString`, no `constructor`, no `__proto__` hazard — which is why it appears in code that maps untrusted keys.

`instanceof` is a chain question

`x instanceof F` asks: does `F.prototype` appear anywhere in `x`'s prototype chain? Nothing about "type", nothing about how `x` was made:

```
rabbit instanceof Object;   // true — Object.prototype is on the chain
[] instanceof Array;        // true
Object.create(null) instanceof Object;   // false — empty chain
```

The next lesson connects this to constructor functions (`F.prototype` is the object `new F()` installs as the prototype). `Symbol.hasInstance` lets a class customise the check — rare, but it is why `instanceof` can lie.

Modifying built-in prototypes

`Array.prototype.last = function () { return this[this.length - 1]; }` works — every array gains `last`. It is also how libraries have broken each other for twenty years: two of them define `last` differently, a future standard adds a real `last` with other semantics, `for…in` over arrays starts yielding `"last"`. Polyfills (adding a *standard* method that an old engine lacks) are the accepted exception; anything else is a hazard. Write a function that takes the array.

Prototype pollution

When object keys come from user input — `settings[userKey] = value`, a deep merge of a parsed JSON body — a key of `"__proto__"` can write to `Object.prototype` itself, and every object in the process suddenly has that property. Guards: `Object.create(null)` for dictionaries, `Map` for anything keyed by input, and merge functions that skip `__proto__`, `constructor` and `prototype`. Several real CVEs in popular libraries were exactly this.

Common mistakes

- Expecting `obj.x = 1` to change the prototype's `x` — it shadows on the instance.
- Using `for…in` on objects that have (or might gain) inherited enumerable properties.
- `Object.setPrototypeOf` on hot objects — set the prototype at creation.
- Extending `Object.prototype` or `Array.prototype` with helpers.
- Thinking `instanceof` checks a class name; it checks a prototype object, which fails across two copies of a library.

Interview angle

- *"How does inheritance work in JavaScript?"* Delegation: every object links to a prototype; missing properties are looked up along the chain to `null`.
- *"Difference between `__proto__` and `prototype`?"* `__proto__` is an object's link to its prototype; `prototype` is a property of functions — the object `new F()` will link to. (Next lesson.)
- *"What does `instanceof` check?"* Whether `F.prototype` is on the object's chain.
- *"How do you create an object with no prototype, and why?"* `Object.create(null)` — a clean dictionary immune to inherited names and pollution.

"What is prototype pollution?" Writing through `__proto__` from untrusted keys so `Object.prototype` gains properties.

Key takeaways

Every object has a prototype link; reads walk it, writes never do — they shadow on the instance.

`Object.create(proto)`, `Object.getPrototypeOf`, `Object.hasOwn` versus `in`, `Object.keys` for own enumerable.

Built-ins: arrays → `Array.prototype` → `Object.prototype` → `null`; primitives box for the lookup.

`instanceof` = "is `F.prototype` on the chain"; `Object.create(null)` has an empty chain.

Never extend built-in prototypes; guard user-supplied keys against `__proto__`.

EXERCISE 1

A prototype-chain resolver

Build objects from commands and resolve lookups the way the engine does. `def <name> <parent⟼[key value]...` creates an object with `Object.create` (parent `-` means a prototype of `null`) and assigns the pairs as own properties. `get <name> <key>` prints `<name>.<key> = <value> (own)` when the property is the object's own, `(from <owner>)` naming the prototype that has it, or `<name>.<key> = undefined` when nothing on the chain does. `set <name> <key> <value>` assigns on that object. `chain <name>` prints the chain's names joined by `→`.

Example:

```
def animal - eats yes legs 4
def rabbit animal name peter
def baby rabbit
get baby name
get baby eats
set baby eats no
get baby eats
get rabbit eats
get baby wings
chain baby
```

→

```
baby.name = peter (from rabbit)
baby.eats = yes (from animal)
baby.eats = no (own)
rabbit.eats = yes (from animal)
baby.wings = undefined
baby → rabbit → animal
```

The `set` shadows on `baby`; `rabbit` and `animal` are untouched.

INPUT 1

```
def animal - eats yes legs 4
def rabbit animal name peter
def baby rabbit
get baby name
get baby eats
set baby eats no
get baby eats
get rabbit eats
get baby wings
chain baby
```

OUTPUT 1

```
baby.name = peter (from rabbit)
baby.eats = yes (from animal)
baby.eats = no (own)
rabbit.eats = yes (from animal)
baby.wings = undefined
baby → rabbit → animal
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim()
const objects = new Map();   // name → object
const nameOf = new Map();    // object → name
for (const line of lines) {
  const [cmd, name, ...rest] = line.trim().split(/\s+/);
  if (cmd === "def") {
    const obj = Object.create(rest[0] === "-" ? null : objects.get(rest[0]));
    for (let i = 1; i + 1 < rest.length; i += 2) obj[rest[i]] = rest[i + 1];
    objects.set(name, obj);
    nameOf.set(obj, name);
  } else if (cmd === "get") {
    const obj = objects.get(name);
    let owner = obj;
    while (owner !== null && !Object.hasOwn(owner, rest[0])) owner = Object.getPrototyp
    if (owner === null) console.log(`${name}.${rest[0]} = undefined`);
    else console.log(`${name}.${rest[0]} = ${obj[rest[0]]} (${owner === obj ? "own" : '
  } else if (cmd === "set") {
    objects.get(name)[rest[0]] = rest[1];   // a write always lands on the object itsel
  } else if (cmd === "chain") {
    const parts = [];
    for (let o = objects.get(name); o !== null; o = Object.getPrototypeOf(o)) parts.pus
    console.log(parts.join(" → "));
  }
}
```

EXERCISE 2

Own versus inherited

The first line is a JSON prototype, the second a JSON child; the starter creates the child with `Object.create(proto)` and copies its own properties on. Print `keys=<Object.keys(child)>`, then `forIn=<every key a for…in loop visits>`, then for each key in the union (proto keys first, then child keys, no repeats) `<key>: in=<key in child> own=<Object.hasOwn> value=<child[key]>`. Finally delete every own property of the child that also exists on the prototype and print `unshadowed: <key>=<value now read>, ...` (or `unshadowed: -` when nothing was shadowing).

Example: `{"eats":true,"legs":4}` then `{"name":"peter","legs":3}` →

```
keys=name,legs
forIn=name,legs,eats
eats: in=true own=false value=true
legs: in=true own=true value=3
name: in=true own=true value=peter
unshadowed: legs=4
```

INPUT 1

```
{"eats":true,"legs":4}
{"name":"peter","legs":3}
```

OUTPUT 1

```
keys=name,legs
forIn=name,legs,eats
eats: in=true own=false value=true
legs: in=true own=true value=3
name: in=true own=true value=peter
unshadowed: legs=4
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim()
const proto = JSON.parse(lines[0]);
const child = Object.assign(Object.create(proto), JSON.parse(lines[1]));
console.log(`keys=${Object.keys(child).join(",")}`);
const forIn = [];
for (const k in child) forIn.push(k);
console.log(`forIn=${forIn.join(",")}`);
for (const k of new Set([...Object.keys(proto), ...Object.keys(child)])) {
  console.log(`${k}: in=${k in child} own=${Object.hasOwn(child, k)} value=${child[k]}`
}
const shadows = Object.keys(child).filter((k) => Object.hasOwn(proto, k));
for (const k of shadows) delete child[k];
console.log(`unshadowed: ${shadows.map((k) => `${k}=${child[k]}`).join(", ") || "-"}`);
```

QUIZ

1. `const child = Object.create(parent); child.x = 1;` — what changed?

A. `parent.x` is now 1

B. Only `child` gained an own property `x`; writes never travel up the chain

C. Both objects

D. It throws — `child` is empty

2. `"toString" in {}` is true because…

A. Every object has an own `toString`

B. The lookup reaches `Object.prototype`, which has `toString`

C. `in` always returns true

D. `{}` is a string

3. `x instanceof F` checks…

A. `x.constructor === F`

B. Whether `F.prototype` appears in `x`'s prototype chain

C. `typeof x === F.name`

D. Whether `x` was created by `new F`

4. Why is `Object.create(null)` used for dictionaries keyed by user input?

A. It is faster

B. It has no prototype: no inherited `constructor` / `toString` collisions and no `__proto__` pollution route

C. It allows non-string keys

D. It is frozen

5. Adding `Array.prototype.last = …` in application code is discouraged because…

A. It is a syntax error

B. Every array changes globally: collisions with libraries and future standards, and `for…in` starts yielding it

C. Arrays are frozen

D. It only affects new arrays

ANSWER KEY · THE PROTOTYPE CHAIN — HOW JAVASCRIPT OBJECTS INHERIT

1. B Reads walk the chain; writes create or update an own property on the receiver.

2. B `Object.hasOwn({}, "toString")` is false; `Object.create(null)` has no `toString` at all.

3. B Nothing about how `x` was made; `Object.create(F.prototype) instanceof F` is true.

4. B `Map` is the other answer; a plain `{}` inherits from `Object.prototype`.

5. B Polyfilling a standard method is the accepted exception.

LESSON 2 · 12 MIN

Constructor functions and what `new` really does

Before `class` (2015) every "class" in JavaScript was a plain function called with `new`, and that machinery is still what `class` compiles down to. Understanding it explains the odd vocabulary — why functions have a `.prototype` property, what `constructor` is, why forgetting `new` used to silently corrupt the global object — and it is the direct answer to a favourite interview question: *implement `new` yourself*. This lesson does exactly that, then compares constructors with the factory-function alternative so you can argue for either.

A function used as a constructor

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
Point.prototype.dist = function () { return Math.hypot(this.x, this.y); };

const p = new Point(3, 4);
p.dist();                                  // 5
Object.getPrototypeOf(p) === Point.prototype;   // true
p.constructor === Point;                   // true — inherited from Point.proto
```

Nothing marks `Point` as special; capitalising it is a convention that says "call me with `new`". Every function created with `function` gets a `.prototype` property automatically — a plain object holding one property, `constructor`, pointing back at the function. Methods go on that object once and are shared by every instance through the chain. Arrow functions and methods have no `.prototype` and cannot be `new`ed.

The four steps of `new`

`new F(args)` does, precisely:

1. Create a fresh object whose prototype is `F.prototype`.
2. Call `F` with `this` bound to that object and the arguments passed.
3. If `F` returns an **object**, that object is the result.
4. Otherwise (it returned `undefined` or a primitive) the fresh object is the result.

Written out:

```
function construct(F, ...args) {
  const obj = Object.create(F.prototype);        // step 1
  const result = F.apply(obj, args);             // step 2
  return result !== null && (typeof result === "object" || typeof result === "fu
}
```

Step 3 is the trapdoor: a constructor that `return`s an object hands back *that* object — the instance is discarded. Used rarely and deliberately (returning a cached instance, a proxy), it is otherwise a bug. A returned primitive is ignored. Modern code would write `Reflect.construct(F, args)` instead of the hand version, but the hand version is what the interviewer wants to see.

`new.target`, and the forgotten `new`

Call `Point(1, 2)` without `new` and `this` is `undefined` in strict mode — `this.x = x` throws. In sloppy mode `this` was the global object, so `x` and `y` quietly became globals: the classic pre-2015 bug. `new.target` inside a function is the constructor being `new`ed, or `undefined` when called plainly — the guard, if you want one:

```
function Point(x, y) {
  if (!new.target) return new Point(x, y);      // tolerate a missing new
  this.x = x; this.y = y;
}
```

Classes refuse outright: calling a class without `new` throws `TypeError: Class constructor Point cannot be invoked without 'new'`.

The `constructor` property

`Point.prototype.constructor === Point` by default, and instances inherit it — `p.constructor.name` is `"Point"`, and `new p.constructor(0, 0)` makes another of "whatever `p` is". It is only a convention: replacing the whole prototype object (`Point.prototype = { dist() {} }`) loses it unless you put it back, and nothing stops anyone assigning `p.constructor = Array`. Do not use it for type checks; use `instanceof`, or better, duck typing.

Factory functions — the other way

```
function makePoint(x, y) {
  return { x, y, dist() { return Math.hypot(x, y); } };
}
const q = makePoint(3, 4);
```

No `new`, no `this`, no prototype — the object is built and returned. Trade-offs:

	constructor / class	factory
Methods	shared on the prototype — one copy	recreated per object (closures) — memory per instance
Privacy	`#fields` (classes)	free — closure variables are private
`instanceof`	works	no (duck type instead)
Forgetting `new`	throws (class) or corrupts (sloppy function)	not an issue
`this` pitfalls	detached methods lose `this`	none — closures capture what they need

Factories are excellent for small objects and for anything passed around as callbacks; classes win when many instances share behaviour and identity matters. Both are idiomatic; know why you picked one.

Prototype gotchas that survive into classes

Shared mutable state on the prototype (`Point.prototype.tags = []`) is shared by *every* instance — one `push` shows everywhere. Per-instance data goes in the constructor.

Adding a method to `Point.prototype` after instances exist works — instances look up live.

Replacing `Point.prototype` after instances exist does **not** re-link the old instances.

`Point.prototype` is not the prototype *of* `Point`; `Object.getPrototypeOf(Point)` is `Function.prototype`. Two different links.

Common mistakes

- Putting methods inside the constructor body (`this.dist = function…`) — a copy per instance for no reason (unless you need the closure).
- Checking type with `x.constructor === Foo` instead of `instanceof` or a duck-typed check.
- Returning `this` explicitly (harmless) — or accidentally returning an object literal (replaces the instance).
- Believing `new` is slow or "not real JavaScript" — it is a small, well-optimised protocol.

Interview angle

- *"What does `new` do?"* Creates an object linked to `F.prototype`, calls `F` with it as `this`, returns the object unless `F` returned an object.
- *"Implement `new` without using it."* `Object.create(F.prototype)` + `F.apply` + the object-return check.
- *"Constructor versus factory?"* Shared prototype methods and `instanceof` versus closure privacy and no `this`.
- *"What is `new.target`?"* The constructor invoked with `new`, or `undefined` for a plain call.
- *"Why is `Function.prototype.prototype` `undefined`?"* Only ordinary `function`s and classes get a `.prototype` object; arrows and methods do not.

Key takeaways

- A constructor is a plain function; `F.prototype` is the object its instances link to; `constructor` points back.
- `new`: create linked object → call with `this` → return it unless an object was returned.
- Strict mode/classes make a forgotten `new` throw; `new.target` detects it.
- Methods on the prototype are shared; per-instance state belongs in the constructor; never share mutable defaults on the prototype.
- Factories trade `instanceof` and shared methods for closure privacy and no `this`.

EXERCISE 1

new, by hand

Implement `construct(F, ...args)` without using `new`: create an object linked to `F.prototype`, call `F` with it as `this`, and return the object the constructor returned when that is an object (or function), otherwise the created one. The starter has three constructors — `Point` (normal), `Box` (returns a plain

object, which must replace the instance) and `Prim` (returns a primitive, which must be ignored) — and prints a comparison against the real `new` for each input line `<Ctor> <numbers>`.

Example: `Point 3 4`, `Box 5`, `Prim 7` →

```
Point(3,4): {"x":3,"y":4} proto=Point instanceof=true matchesNew=true
Box(5): {"side":5,"kind":"box"} proto=Object instanceof=false matchesNew=true
Prim(7): {"n":7} proto=Prim instanceof=true matchesNew=true
```

INPUT 1

```
Point 3 4
Box 5
Prim 7
```

OUTPUT 1

```
Point(3,4): {"x":3,"y":4} proto=Point
instanceof=true matchesNew=true
Box(5): {"side":5,"kind":"box"}
proto=Object instanceof=false
matchesNew=true
Prim(7): {"n":7} proto=Prim
instanceof=true matchesNew=true
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim()
function Point(x, y) { this.x = x; this.y = y; }
Point.prototype.dist = function () { return Math.hypot(this.x, this.y); };
function Box(side) { this.side = side; return { side, kind: "box" }; }   // returns an
function Prim(n) { this.n = n; return n * 2; }                            // returns a pr
const ctors = { Point, Box, Prim };
function construct(F, ...args) {
  const obj = Object.create(F.prototype);          // 1. a fresh object linked to F.pro
  const result = F.apply(obj, args);               // 2. run F with this = obj
  const isObject = result !== null && (typeof result === "object" || typeof result ===
  return isObject ? result : obj;                  // 3./4. an object result wins, anyt
}
for (const line of lines) {
  const [name, ...args] = line.trim().split(/\s+/);
  const F = ctors[name];
  const made = construct(F, ...args.map(Number));
  const real = new F(...args.map(Number));
  const proto = Object.getPrototypeOf(made) === F.prototype ? name : Object.getPrototyp
  console.log(`${name}(${args.join(",")}): ${JSON.stringify(made)} proto=${proto} insta
}
```

EXERCISE 2

Factory versus constructor, measured

The starter builds `n` points twice — with `new Point(x, y)` (method on the prototype) and with `makePoint(x, y)` (method in a closure). Print for each index `#i: ctor=<dist 2dp> factory=<dist 2dp> equal=<===>`; then `instanceof: ctor=<every one instanceof Point> factory=<any instanceof Point>`; then `keys: ctor=<Object.keys of the first> factory=<same>`; then `methodShared: ctor=<all dist functions identical> factory=<same>` (print `n/a` when fewer than two points); then `constructor: ctor=<constructor.name> factory=<constructor.name>` (`-` when empty).

Example: `2 3 4 6 8` →

```
#0: ctor=5.00 factory=5.00 equal=true
#1: ctor=10.00 factory=10.00 equal=true
instanceof: ctor=true factory=false
keys: ctor=x,y factory=x,y,dist
methodShared: ctor=true factory=false
constructor: ctor=Point factory=Object
```

INPUT 1

```
2 3 4 6 8
```

OUTPUT 1

```
#0: ctor=5.00 factory=5.00 equal=true
#1: ctor=10.00 factory=10.00 equal=true
instanceof: ctor=true factory=false
keys: ctor=x,y factory=x,y,dist
methodShared: ctor=true factory=false
constructor: ctor=Point factory=Object
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
function Point(x, y) { this.x = x; this.y = y; }
Point.prototype.dist = function () { return Math.hypot(this.x, this.y); };
function makePoint(x, y) { return { x, y, dist() { return Math.hypot(x, y); } }; }
const n = Number(tokens[0]);
const built = [], made = [];
for (let i = 0; i < n; i++) {
  const x = Number(tokens[1 + 2 * i]), y = Number(tokens[2 + 2 * i]);
  built.push(new Point(x, y));
  made.push(makePoint(x, y));
}
for (let i = 0; i < n; i++) console.log(`#${i}: ctor=${built[i].dist().toFixed(2)} fact
console.log(`instanceof: ctor=${built.every((p) => p instanceof Point)} factory=${made.
console.log(`keys: ctor=${Object.keys(built[0] ?? {}).join(",")} factory=${Object.keys(
const shared = (list) => list.length < 2 ? "n/a" : String(list.every((p) => p.dist ===
console.log(`methodShared: ctor=${shared(built)} factory=${shared(made)}`);
console.log(`constructor: ctor=${built[0] ? built[0].constructor.name : "-"} factory=${
```

QUIZ

1. The four steps of `new F(a)`:

A. Copy `F`, call it, return the copy

B. Create an object linked to `F.prototype`; call `F` with it as `this`; return `F`'s result if it is an object; otherwise return the created object

C. Call `F`, wrap the result in an object

D. Create an object, copy `F.prototype`'s properties into it, call `F`

2. `function F() { return 5; } new F()` gives…

A. `5`

B. The new instance — a primitive return value is ignored

C. `undefined`

D. A `TypeError`

3. `F.prototype` is…

A. The prototype of the function `F`

B. The object that instances made with `new F` link to as their prototype

C. A copy of `Object.prototype`

D. Only present on classes

4. Calling a `class` constructor without `new`…

A. Works like a function

B. Throws a `TypeError`

C. Returns `undefined`

D. Creates the instance anyway

5. A factory function's advantage over a constructor:

A. Methods are shared on a prototype

B. `instanceof` works

C. Closure privacy and no `this` to lose; no `new` to forget

D. It is always faster

ANSWER KEY · CONSTRUCTOR FUNCTIONS AND WHAT `NEW` REALLY DOES

1. B Step three is why a constructor returning an object discards the instance.

2. B Only an object (or function) return replaces the instance.

3. B The prototype of `F` itself is `Function.prototype`; two different links.

4. B A sloppy-mode function constructor would instead set properties on the global object.

5. C The trade: per-object method copies and no `instanceof`.

LESSON 3 · 14 MIN

Class syntax — fields, methods, statics, private and accessors

`class` gives the constructor-and-prototype machinery a shape that reads like every other language, plus things the old syntax could not express: real private members, class fields with initialisers, static blocks. It is still prototypes underneath — `typeof Point` is `"function"`, methods land on `Point.prototype` — but the syntax also changes a few semantics (strict mode, no hoisting, throw without `new`). This lesson covers the complete surface as it runs on Node 16, with the corners that trip people: field initialisation order, `this` in methods passed as callbacks, and what `#private` actually guarantees.

The full surface

```
class Account {
  static count = 0;                       // static field — on the class, not in
  static #registry = new Map();           // private static
  #balance = 0;                           // private instance field, initialised
  owner;                                  // public field, undefined until assig
  history = [];                           // public field with initialiser — a N

  constructor(owner, opening = 0) {
    this.owner = owner;
    this.#balance = opening;
    Account.count++;
    Account.#registry.set(owner, this);
  }

  deposit(amount) {                       // method — lives on Account.prototype
    if (amount ⩽ 0) throw new RangeError("amount must be positive");
    this.#balance += amount;
    this.history.push(["deposit", amount]);
    return this;                          // fluent
  }

  get balance() { return this.#balance; }          // accessor — read like a pro
  set nickname(v) { this._nick = String(v).trim(); }

  #audit(kind) { /* private method */ }

  static open(owner) { return new Account(owner); }   // static method — factory
  static lookup(owner) { return Account.#registry.get(owner); }

  toString() { return `${this.owner}: ${this.#balance}`; }
}
```

What the syntax changes

Strict mode inside the class body, always.

Not hoisted like a function declaration: `new Account()` before the `class` line is a `ReferenceError` (temporal dead zone, like `let`).

Must be called with `new` — a plain call throws.

Methods are non-enumerable: `Object.keys(new Account("a"))` lists fields, not methods; `for…in` skips them.

No commas between members; semicolons after fields are optional but conventional.

Fields and initialisation order

Public fields (`history = []`) are assigned on the instance **before** the constructor body runs (for a base class) — each instance gets its own value from re-evaluating the initialiser, which is why `history = []` is safe where `Account.prototype.history = []` would be shared. In a subclass, fields initialise right after `super()` returns (next lesson). Field initialisers can reference `this` and earlier fields. Arrow-function fields (`handle = () ⇒ {…}`) capture `this` per instance — the standard fix for methods handed to event listeners, at the cost of one closure per instance instead of one shared method.

Private members

`#balance` is private in the strong sense: not a naming convention, not `_balance` , but a name the language checks at parse time. Outside the class body `acct.#balance` is a **SyntaxError**, not `undefined` ; `Object.keys` , `JSON.stringify` , `for…in` , `Reflect.ownKeys` and Proxies cannot see it; a subclass cannot reach a parent's `#field` . Two classes with a `#x` each have different `#x` s. `#x in obj` (Node 16.4+) tests whether an object has the brand — the safe way to check "is this really one of ours" inside a static method. Because privates are per-class brands, an object created by another copy of the same class (two versions of a library) fails the check — the same failure mode `instanceof` has.

Accessors

`get balance()` defines a read-only computed property; add `set balance(v)` for writes. Accessors sit on the prototype like methods. Use them for derived values and validated writes, not for wrapping every field Java-style — a public field is an honest public field. Accessors run code on every read; do not put anything expensive there.

Statics

`static` members belong to the class object: `Account.count` , `Account.open("x")` . Inside a static method `this` is the class, which matters for subclasses: `static create() { return new this(); }` makes a `SavingsAccount` when called as `SavingsAccount.create()` . `static { … }` blocks (Node 16.11+) run once when the class is evaluated — for setup that needs several statements.

Class expressions and names

```
const Temp = class Temperature { };     // named expression: Temp.name is "Tempe
const Anon = class { };                 // Anon.name is "Anon" (inferred)
const mixin = (Base) ⇒ class extends Base { };   // classes are values — pass a
```

Methods as callbacks — `this` again

`button.on("click", acct.deposit)` detaches the method; inside, `this` is `undefined` (strict) and `this.#balance` throws. Fixes, in order of preference: an arrow at the call site (`() ⇒ acct.deposit(10)`), `acct.deposit.bind(acct)` , or an arrow-function field on the class when the method is *always* used as a callback. Module 3's five rules apply unchanged.

Classes versus objects and closures

A class earns its keep when there will be many instances sharing behaviour, when identity (`instanceof` , brand checks) matters, or when a framework expects one. A module that exports functions over a closed-over state is often simpler than a singleton class; a plain object literal beats a class with one instance and no methods. "Prefer composition" is not anti-class — it is anti-*hierarchy*.

Common mistakes

- `this.history = []` inside a method meant to append — reassigning instead of `push` .
- Forgetting that fields with initialisers make a new value per instance, and putting the shared cache in a field instead of a `static` .
- Reading a private of another object of the same class as `other.#x` — this *is* allowed (same class body); reading it outside the class is not.
- Passing `obj.method` as a callback and losing `this` .
- Defining a getter and then trying to assign to it without a setter — silently ignored in sloppy mode, `TypeError` in strict (which classes are).

Interview angle

"Are classes just syntactic sugar?" Mostly — prototypes underneath — but with real additions: private members, strict mode, no hoisting, `new` required, non-enumerable methods.

"`#private` versus `_private`?" Language-enforced (SyntaxError outside, invisible to reflection) versus a naming convention.

"Where do fields versus methods live?" Fields on the instance, methods on the prototype, statics on the class.

"Why arrow-function fields?" Per-instance `this` capture for callbacks; cost: one closure per instance.

"When is `this` the class?" In static methods — so `new this()` builds the subclass that was called.

Key takeaways

The surface: fields (public/private/static), methods, accessors, statics, static blocks, private methods.

Classes are strict, not hoisted, throw without `new`, and have non-enumerable methods.

Field initialisers run per instance before the constructor body; `history = []` is safe, prototype arrays are shared.

`#names` are enforced privacy and per-class brands; `#x in obj` tests them.

Methods passed as callbacks lose `this` — arrow at the call site, `bind`, or an arrow field.

EXERCISE 1

A bank account with private state

Complete `Account`: a `#balance` private field set from the opening amount, a `static count` incremented per instance, a public `history` array, `deposit(amount)` / `withdraw(amount)` that push `["deposit", amount]` / `["withdraw", amount]` and throw `RangeError("amount must be positive")` for non-positive amounts and `RangeError("insufficient funds in <name>")` when overdrawn, a `get balance()` accessor, and `toString()` returning `<name>: <balance> (<n> transactions)`. Then run the commands `open <name> <amount>`, `deposit`, `withdraw`, `show <name>` (prints `String(account)`) and `count` (prints `accounts=<Account.count>`); print `error: <message>` for a thrown error.

Example:

```
open ada 100
open bob 20
deposit ada 50
withdraw bob 30
withdraw ada 70
deposit bob -5
show ada
show bob
count
```

→

```
error: insufficient funds in bob
error: amount must be positive
ada: 80 (2 transactions)
bob: 20 (0 transactions)
accounts=2
```

INPUT 1

```
open ada 100
open bob 20
deposit ada 50
withdraw bob 30
withdraw ada 70
deposit bob -5
show ada
show bob
count
```

OUTPUT 1

```
error: insufficient funds in bob
error: amount must be positive
ada: 80 (2 transactions)
bob: 20 (0 transactions)
accounts=2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim()
class Account {
  static count = 0;
  #balance = 0;
  history = [];
  constructor(name, opening) {
    this.name = name;
    this.#balance = opening;
    Account.count++;
  }
  deposit(amount) {
    if (amount <= 0) throw new RangeError("amount must be positive");
    this.#balance += amount;
    this.history.push(["deposit", amount]);
    return this;
  }
  withdraw(amount) {
    if (amount <= 0) throw new RangeError("amount must be positive");
    if (amount > this.#balance) throw new RangeError(`insufficient funds in ${this.name
    this.#balance -= amount;
    this.history.push(["withdraw", amount]);
    return this;
  }
  get balance() { return this.#balance; }
  toString() { return `${this.name}: ${this.#balance} (${this.history.length} transacti
}
const accounts = new Map();
for (const line of lines) {
  const [cmd, name, amount] = line.trim().split(/\s+/);
  try {
    if (cmd === "open") accounts.set(name, new Account(name, Number(amount)));
    else if (cmd === "deposit") accounts.get(name).deposit(Number(amount));
    else if (cmd === "withdraw") accounts.get(name).withdraw(Number(amount));
    else if (cmd === "show") console.log(String(accounts.get(name)));
    else if (cmd === "count") console.log(`accounts=${Account.count}`);
  } catch (e) {
    console.log(`error: ${e.message}`);
  }
}
```

EXERCISE 2

Accessors that convert and validate

Write `Temperature` with a private `#celsius`, `get` / `set celsius`, `get` / `set fahrenheit` (the setter converts and reuses the celsius setter), and `get kelvin`. Setting anything below −273.15 °C throws `RangeError("below absolute zero")` and leaves the value unchanged. Commands: `c <v>`, `f <v>`,

`show` → `C=<1dp> F=<1dp> K=<2dp>`, `inspect` → `keys=<JSON of Object.keys(t)> json=<JSON.stringify(t)> protoHasCelsius=<Object.hasOwn(Temperature.prototype, "celsius")>`. Print `error: <message>` on a rejected write.

Example: `c 100`, `show`, `f 32`, `show`, `c -300`, `show`, `inspect` →

```
C=100.0 F=212.0 K=373.15
C=0.0 F=32.0 K=273.15
error: below absolute zero
C=0.0 F=32.0 K=273.15
keys=[] json={} protoHasCelsius=true
```

The last line is the lesson: private fields are invisible to `Object.keys` and JSON, and accessors live on the prototype.

INPUT 1

```
c 100
show
f 32
show
c -300
show
inspect
```

OUTPUT 1

```
C=100.0 F=212.0 K=373.15
C=0.0 F=32.0 K=273.15
error: below absolute zero
C=0.0 F=32.0 K=273.15
keys=[] json={} protoHasCelsius=true
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim()
class Temperature {
  #celsius = 0;
  get celsius() { return this.#celsius; }
  set celsius(v) {
    if (v < -273.15) throw new RangeError("below absolute zero");
    this.#celsius = v;
  }
  get fahrenheit() { return this.#celsius * 9 / 5 + 32; }
  set fahrenheit(v) { this.celsius = (v - 32) * 5 / 9; }   // reuse the validating sett
  get kelvin() { return this.#celsius + 273.15; }
}
const t = new Temperature();
for (const line of lines) {
  const [cmd, value] = line.trim().split(/\s+/);
  try {
    if (cmd === "c") t.celsius = Number(value);
    else if (cmd === "f") t.fahrenheit = Number(value);
    else if (cmd === "show") console.log(`C=${t.celsius.toFixed(1)} F=${t.fahrenheit.to
    else if (cmd === "inspect") console.log(`keys=${JSON.stringify(Object.keys(t))} jso
  } catch (e) {
    console.log(`error: ${e.message}`);
  }
}
```

QUIZ

1. Where does a class method live, and where does a field with an initialiser live?

A. Both on the instance

B. Method on `C.prototype`, field on each instance (initialiser re-evaluated per instance)

C. Both on the prototype

D. Method on the class object, field on the prototype

2. `acct.#balance` written outside the class body is…

A. `undefined`

B. A `SyntaxError` — private names are checked at parse time

C. A `TypeError` at runtime

D. Allowed for subclasses

3. `Object.keys(new C())` for `class C { x = 1; m() {} }` is…

A. `["x", "m"]`

B. `["x"]` — class methods are non-enumerable and on the prototype

C. `["m"]`

D. `[]`

4. `new C()` written before the `class C {}` line…

A. Works — classes hoist like functions

B. Throws a `ReferenceError` — the class binding is in its temporal dead zone

C. Returns `undefined`

D. Creates an empty object

5. Inside `static create() { return new this(); }`, called as `Sub.create()`, `this` is…

A. The instance

B. `Sub` — in a static method `this` is the class it was called on

C. `undefined`

D. The base class

ANSWER KEY · CLASS SYNTAX — FIELDS, METHODS, STATICS, PRIVATE AND ACCESSORS

1. B That is why `history = []` is per-instance while a prototype array would be shared.

2. B Privates are invisible to reflection, JSON and subclasses too.

3. B Spread and `JSON.stringify` follow the same rule.

4. B Function declarations hoist with their body; classes behave like `let`.

5. B That is what makes static factories work for subclasses.

LESSON 4 · 13 MIN

Inheritance — extends, super, overriding and extending built-ins

`extends` links two prototype chains: the subclass's prototype inherits from the parent's prototype (so instances find parent methods), and the subclass constructor inherits from the parent constructor (so statics are found too). `super` is the keyword for reaching the parent from either place. The rules are few but strict — call `super()` before touching `this`, fields initialise after it, `super.method()` uses the *home object* not the receiver — and the lesson closes with the two things people actually do with inheritance in JavaScript: extend `Error`, `Map` and `Array`, and prefer composition for everything else.

The two chains `extends` creates

```
class Shape {
  constructor(name) { this.name = name; }
  area() { return 0; }
  describe() { return `${this.name} with area ${this.area().toFixed(2)}`; }
  static create(...args) { return new this(...args); }
}
class Circle extends Shape {
  constructor(r) {
    super("circle");        // must run before any use of this
    this.r = r;
  }
  area() { return Math.PI * this.r ** 2; }                        // override
  describe() { return super.describe() + ` (r=${this.r})`; }      // extend the
}

const c = new Circle(1);
Object.getPrototypeOf(Circle.prototype) === Shape.prototype;   // instance chain
Object.getPrototypeOf(Circle) === Shape;                        // static chain:
Circle.create(2) instanceof Circle;                             // true — static
c instanceof Shape;                                             // true
```

Overriding is just shadowing on the nearer prototype: `c.area` finds `Circle.prototype.area` first. `describe`, found on `Shape.prototype`, calls `this.area()` — and `this` is `c`, so it dispatches to the override. That late binding is the entire point of polymorphism, and it needs no annotations.

`super()` and construction order

In a derived class the `this` object is created by the **base** constructor — `super()` is the call that creates it — so:

- A derived constructor **must** call `super()` before reading or writing `this` (`ReferenceError: Must call super constructor in derived class before accessing 'this'`).
- A derived class with no constructor gets `constructor(...args) { super(...args); }` for free.
- The derived class's **fields initialise right after `super()` returns**, before the rest of the constructor body. So if a base constructor calls an overridden method, that method runs before the subclass's fields exist:

```
class Base { constructor() { this.init(); } init() {} }
class Derived extends Base {
  value = 42;
  init() { console.log(this.value); }     // logs undefined — field not yet init
}
new Derived();
```

The fix is a design rule, not a trick: base constructors do not call overridable methods.

`super.method()` and the home object

`super.describe()` looks up `describe` on the **prototype of the class where the method was written** (the *home object*), not on `this`'s prototype. That is what makes two levels of overriding work — each `super` call climbs exactly one level from where it was written, regardless of how deep the receiver's class is. It also means `super` in a method copied to another object still refers to the original parent. Static methods can use `super.staticMethod()` too, climbing the static chain.

Extending built-ins

```
class HttpError extends Error {
  constructor(status, message) {
    super(message);                          // sets message and captures the stack
    this.name = "HttpError";                 // otherwise the stack says "Error"
    this.status = status;
  }
}
try { throw new HttpError(404, "no such page"); }
catch (e) { e instanceof HttpError && e instanceof Error; }   // true

class DefaultMap extends Map {
  constructor(makeDefault) { super(); this.makeDefault = makeDefault; }
  get(key) { if (!this.has(key)) this.set(key, this.makeDefault()); return super
}

class Stack extends Array {
  peek() { return this[this.length - 1]; }
}
const s = Stack.from([1, 2, 3]);
s.map((x) => x * 2) instanceof Stack;     // true — array methods build the subc
Array.isArray(s);                         // true
```

Subclassing `Error` is the one inheritance every codebase needs (module 7 goes deep). `Map` / `Set` subclasses work cleanly. `Array` subclasses work but carry the `Symbol.species` machinery (methods return the subclass) and were impossible before 2015 — prefer a class that *holds* an array unless you truly want array behaviour. `Promise` subclasses are legal and almost never worth it.

Mixins: multiple inheritance by function

JavaScript has single inheritance; a mixin is a function from class to class:

```
const Serializable = (Base) => class extends Base {
  toJSON() { return { ...this, kind: this.constructor.name }; }
};
const Comparable = (Base) => class extends Base {
  compareTo(other) { return this.valueOf() - other.valueOf(); }
};
class Money extends Serializable(Comparable(Object)) { /* … */ }
```

Each application inserts one prototype into the chain. Use sparingly — three mixins deep and nobody knows where a method comes from.

Composition over inheritance

A class hierarchy is right when subclasses are genuinely *kinds of* the parent, share most behaviour, and the hierarchy is shallow (one or two levels). It is wrong when you reach for it to share a utility method, when the "is-a" is a stretch (`Button extends Rectangle`), or when the tree keeps growing sideways. The alternative is holding an instance of the thing and delegating (`this.logger.log(...)`), or passing behaviour in as functions. Most production JavaScript has few deep hierarchies and many small classes that *contain* other objects.

Common mistakes

- Using `this` before `super()` ; forgetting `super(...args)` when you add a constructor.
- Base constructors calling methods the subclass overrides (fields are not ready).
- Forgetting `this.name = "…"` in an `Error` subclass, so logs say `Error` .
- Overriding a method without calling `super.method()` when the parent's version did required work.
- Deep hierarchies for code reuse; `extends Array` when an array field would do.

Interview angle

- *"What does `extends` set up?"* Two prototype links: `Sub.prototype → Base.prototype` for instances and `Sub → Base` for statics.
- *"Why must `super()` be called first?"* The base constructor creates `this` ; before it there is no object.
- *"How does method overriding dispatch?"* Property lookup finds the nearest prototype's version; `this` is the receiver, so parent code calls child overrides.
- *"How do you make a custom error?"* `class X extends Error` , `super(message)` , set `name` , add fields.
- *"Inheritance versus composition?"* Extend for real is-a relationships, shallow; otherwise hold and delegate.

Key takeaways

- `extends` links instance and static chains; `instanceof` follows the instance chain.

`super()` first in derived constructors; fields initialise after it; base constructors should not call overridables.

`super.method()` climbs one level from where the method was written (home object).

Extend `Error` (set `name`), `Map` / `Set` freely; `Array` works via `Symbol.species` but usually hold an array instead.

Mixins are functions from class to class; prefer composition for sharing behaviour.

EXERCISE 1

A shape hierarchy

Given `Shape` (with `describe()` printing `<name>: area=<2dp> perimeter=<2dp>`), write `Circle(r)`, `Rect(w, h)` and `Square(s) extends Rect`. `Circle.describe` appends `r=<r>`, `Rect.describe` appends `<w>x<h>` — both by calling `super.describe()`. A square is a rect named `square` built with `super(s, s, "square")`. Read one shape per line (`circle r`, `rect w h`, `square s`), print each `describe()`, then `total area=<2dp> shapes=<n> rects=<count instanceof Rect> squares=<count instanceof Square>`, then `chain=<constructor names along a Square instance's prototype chain>`.

Example: `circle 1`, `rect 2 3`, `square 4` →

```
circle: area=3.14 perimeter=6.28 r=1
rect: area=6.00 perimeter=10.00 2x3
square: area=16.00 perimeter=16.00 4x4
total area=25.14 shapes=3 rects=2 squares=1
chain=Square → Rect → Shape → Object
```

INPUT 1

```
circle 1
rect 2 3
square 4
```

OUTPUT 1

```
circle: area=3.14 perimeter=6.28 r=1
rect: area=6.00 perimeter=10.00 2x3
square: area=16.00 perimeter=16.00 4x4
total area=25.14 shapes=3 rects=2
squares=1
chain=Square → Rect → Shape → Object
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim()
class Shape {
  constructor(name) { this.name = name; }
  area() { return 0; }
  perimeter() { return 0; }
  describe() { return `${this.name}: area=${this.area().toFixed(2)} perimeter=${this.pe
}
class Circle extends Shape {
  constructor(r) { super("circle"); this.r = r; }
  area() { return Math.PI * this.r ** 2; }
  perimeter() { return 2 * Math.PI * this.r; }
  describe() { return `${super.describe()} r=${this.r}`; }
}
class Rect extends Shape {
  constructor(w, h, name = "rect") { super(name); this.w = w; this.h = h; }
  area() { return this.w * this.h; }
  perimeter() { return 2 * (this.w + this.h); }
  describe() { return `${super.describe()} ${this.w}x${this.h}`; }
}
class Square extends Rect {
  constructor(s) { super(s, s, "square"); }
}
const shapes = [];
for (const line of lines) {
  const [kind, ...nums] = line.trim().split(/\s+/);
  const [a, b] = nums.map(Number);
  if (kind === "circle") shapes.push(new Circle(a));
  else if (kind === "rect") shapes.push(new Rect(a, b));
  else if (kind === "square") shapes.push(new Square(a));
}
for (const s of shapes) console.log(s.describe());
const total = shapes.reduce((sum, s) => sum + s.area(), 0);
console.log(`total area=${total.toFixed(2)} shapes=${shapes.length} rects=${shapes.filt
const chain = [];
for (let p = Object.getPrototypeOf(new Square(1)); p !== null; p = Object.getPrototypeO
console.log(`chain=${chain.join(" → ")}`);
```

EXERCISE 2

Extending Map, Array and Error

Write three built-in subclasses. `DefaultMap extends Map` takes a `makeDefault` function and its `get` creates the default for a missing key. `Stack extends Array` adds `peek()`. `HttpError extends Error` takes `(status, message)`, sets `name` to `"HttpError"` and stores `status`. Input: line 1 words, line 2 numbers, line 3 a status code and a message. Print `counts: <word=count sorted by word>` `instanceofMap=<>` using the DefaultMap, then `stack: peek=<> length=<> isArray=<Array.isArray>`

`mapKeepsClass=<stack.map(...) instanceof Stack>` `doubledPeek=<peek of the doubled stack>`, then throw the error and print `<name> <status>: <message>` `instanceofError=<>` `stackStartsWithName=<e.stack.startsWith("HttpError")>`.

Example: `to be or not to be` / `3 1 4 1 5` / `404 page not found` →

```
counts: be=2 not=1 or=1 to=2 instanceofMap=true
stack: peek=5 length=5 isArray=true mapKeepsClass=true doubledPeek=10
HttpError 404: page not found instanceofError=true stackStartsWithName=true
```

INPUT 1

```
to be or not to be
3 1 4 1 5
404 page not found
```

OUTPUT 1

```
counts: be=2 not=1 or=1 to=2
instanceofMap=true
stack: peek=5 length=5 isArray=true
mapKeepsClass=true doubledPeek=10
HttpError 404: page not found
instanceofError=true
stackStartsWithName=true
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim()
class DefaultMap extends Map {
  constructor(makeDefault) { super(); this.makeDefault = makeDefault; }
  get(key) {
    if (!this.has(key)) this.set(key, this.makeDefault());
    return super.get(key);
  }
}
class Stack extends Array {
  peek() { return this[this.length - 1]; }
}
class HttpError extends Error {
  constructor(status, message) {
    super(message);
    this.name = "HttpError";
    this.status = status;
  }
}
const words = lines[0].trim().split(/\s+/);
const numbers = lines[1].trim().split(/\s+/).map(Number);
const [status, ...msg] = lines[2].trim().split(/\s+/);
const counts = new DefaultMap(() => 0);
for (const w of words) counts.set(w, counts.get(w) + 1);
console.log(`counts: ${[...counts].sort(([a], [b]) => a.localeCompare(b)).map(([w, n])
const stack = Stack.from(numbers);
const doubled = stack.map((x) => x * 2);
console.log(`stack: peek=${stack.peek()} length=${stack.length} isArray=${Array.isArray
try {
  throw new HttpError(Number(status), msg.join(" "));
} catch (e) {
  console.log(`${e.name} ${e.status}: ${e.message} instanceofError=${e instanceof Error
}
```

QUIZ

1. In a derived class constructor, using `this` before `super()` …

A. Is fine

B. Throws a `ReferenceError` — the base constructor is what creates `this`

C. Refers to the parent instance

D. Creates a second object

2. A base constructor calls `this.init()`, which the subclass overrides to read a subclass field. The field is…

A. Initialised

B. `undefined` — subclass fields initialise only after `super()` returns

C. Frozen

D. Inherited from the base

3. `super.describe()` looks up `describe` on…

A. `this`'s prototype

B. The prototype of the class where the method was written (the home object's parent)

C. The base of the whole hierarchy

D. The instance itself

4. In `class HttpError extends Error`, why set `this.name`?

A. Otherwise `instanceof` fails

B. Otherwise stack traces and `String(err)` say `Error` rather than `HttpError`

C. It is required by `super`

D. To make it enumerable

5. `extends` creates which links?

A. Only `Sub.prototype → Base.prototype`

B. `Sub.prototype → Base.prototype` for instance lookup **and** `Sub → Base` for static lookup

C. Only `Sub → Base`

D. A copy of the parent's methods

ANSWER KEY · INHERITANCE — EXTENDS, SUPER, OVERRIDING AND EXTENDING BUILT-INS

1. B A derived class without a constructor gets `constructor(...a) { super(...a); }` implicitly.

2. B Design rule: base constructors do not call overridable methods.

3. B Each `super` call climbs exactly one level from where it was written, so multi-level overrides compose.

4. B `super(message)` sets the message and captures the stack; the name is yours to set.

5. B `Sub.create()` finds an inherited static through the second link.

LESSON 5 · 13 MIN

Property descriptors and the object protocols — toString, valueOf, toJSON, Symbol.toPrimitive

Two more layers sit under ordinary property access. Every property has a **descriptor** — is it writable, does it show up in loops, can it be deleted or redefined — and `Object.defineProperty` lets you set those bits, which is how `Object.freeze`, non-enumerable class methods and read-only constants are built. And the language consults a handful of **well-known methods** when it needs to turn your object into something else: `toString` for strings, `valueOf` for numbers, `toJSON` for `JSON.stringify`, `Symbol.toPrimitive` to take over all of it, `Symbol.toStringTag` for the `[object X]` label. Implement them and your class plugs into template literals, `+`, sorting, JSON and logging without callers doing anything special.

Descriptors

```
const o = { a: 1 };
Object.getOwnPropertyDescriptor(o, "a");
// { value: 1, writable: true, enumerable: true, configurable: true }

Object.defineProperty(o, "id", { value: 42 });          // defaults are all FALS
Object.defineProperty(o, "double", { get() { return this.a * 2; }, enumerable: t
Object.keys(o);              // ["a", "double"] — id is non-enumerable
JSON.stringify(o);           // {"a":1,"double":2} — same rule
o.id = 7;                    // strict mode: TypeError; sloppy: silently ignored
delete o.id;                 // false / TypeError — non-configurable
```

Three flags on a data property: **writable** (can the value change), **enumerable** (does it appear in `for…in`, `Object.keys`, spread, `JSON.stringify`), **configurable** (can it be deleted or have its descriptor changed; a non-configurable property can still go from writable to non-writable — one-way). Accessor properties replace `value` / `writable` with `get` / `set`. Properties created by assignment or literals are all-true; `defineProperty` defaults to all-false, which is the surprise.

`Object.defineProperties(obj, {…})` sets several; `Object.getOwnPropertyDescriptors(obj)` reads all — the correct way to copy accessors, since `{...obj}` and `Object.assign` **invoke** getters and copy the resulting values.

The lock-down trio

`Object.preventExtensions(o)` — no new properties.

`Object.seal(o)` — plus every property non-configurable (no deletes, no redefinition); values still writable.

`Object.freeze(o)` — plus non-writable. Shallow, as module 4 said.

`Object.isFrozen/isSealed/isExtensible` read the state.

Class methods are defined non-enumerable, which is why they do not appear in `Object.keys(instance)` or in JSON — and why spreading an instance drops its methods.

`toString` and `valueOf`

When an object meets a string context (`${obj}`, `String(obj)`, `"" + obj` for the string case) or a numeric one (`+obj`, `obj * 2`, `obj < other`), the engine converts it via **ToPrimitive** with a hint — `"string"`, `"number"` or `"default"`. Without `Symbol.toPrimitive`, hint `"string"` tries `toString` then `valueOf`; hints `"number"` and `"default"` try `valueOf` then `toString`. `+` uses `"default"` (so `valueOf` first — that is why `date1 - date2` is a number of milliseconds while `${date}` is a readable string); `==` with a primitive uses `"default"`; `Date` is the one built-in whose `"default"` behaves like `"string"`.

```
class Money {
  constructor(cents, currency = "USD") { this.cents = cents; this.currency = cur
  valueOf() { return this.cents; }                        // numeric contexts
  toString() { return `${(this.cents / 100).toFixed(2)} ${this.currency}`; }
}
const m = new Money(1250);
`${m}`;          // "12.50 USD"  (hint string → toString)
m + 1;           // 1251         (hint default → valueOf)
m > new Money(1000);   // true    (relational → number → valueOf)
[new Money(300), new Money(100)].sort((a, b) => a - b);   // valueOf makes the c
```

`Symbol.toPrimitive` — one method, all hints

```
class Temperature {
  constructor(c) { this.c = c; }
  [Symbol.toPrimitive](hint) {
    if (hint === "number") return this.c;
    if (hint === "string") return `${this.c}°C`;
    return this.c;                    // "default": + and ==
  }
}
```

When present it wins over both `toString` and `valueOf`. Returning a non-primitive throws a `TypeError`. Use it when the two-method dance is not expressive enough (a default that differs from number), otherwise `toString` / `valueOf` read more plainly.

`toJSON`

`JSON.stringify` calls `toJSON()` if present and serialises **its return value** instead of the object — `Date` uses this to emit an ISO string. Return a plain object or a primitive; return `undefined` to omit the value entirely. This is also how you expose a class's private state to JSON deliberately (`#fields` are invisible otherwise), and how you drop derived or sensitive fields.

```
class User {
  #passwordHash;
  constructor(name, hash) { this.name = name; this.#passwordHash = hash; }
  toJSON() { return { name: this.name, kind: "user" }; }
}
JSON.stringify([new User("ada", "…")]);   // [{"name":"ada","kind":"user"}]
```

The second argument of `JSON.stringify` (a **replacer** function or an allow-list array) and of `JSON.parse` (a **reviver**) do the same job from the outside — `JSON.parse(text, (k, v) ⇒ k === "when" ? new Date(v) : v)` is how dates come back to life.

`Symbol.toStringTag` and inspection

`Object.prototype.toString.call(x)` returns `"[object Array]"`, `"[object Date]"`, `"[object Null]"` — the most reliable built-in type test there is, and it reads a class's `get [Symbol.toStringTag]() { return "Money"; }` to give `"[object Money]"`. Node's `console.log` and `util.inspect` also honour a custom `[Symbol.for("nodejs.util.inspect.custom")]` method for how an instance prints in the REPL. Neither replaces `instanceof`; they make logs readable.

Common mistakes

`defineProperty` without `enumerable: true` / `writable: true` when you wanted a normal property.

Copying an object with getters via spread and being surprised the copy has plain values.

`toString` returning a non-string, or `valueOf` returning an object (falls through to `toString` — confusing).

Relying on `+` calling `toString` (it calls `valueOf` first).

Assuming `JSON.stringify` will see private fields or methods.

Interview angle

"What are the property attributes?" `writable`, `enumerable`, `configurable` (plus `get` / `set` for accessors); `defineProperty` defaults them to false.

"`Object.freeze` versus `seal` versus `preventExtensions`?" No writes/no deletes/no adds versus no deletes/no adds versus no adds.

"How does `${obj}` decide what to print?" ToPrimitive with hint string: `Symbol.toPrimitive` if any, else `toString`, else `valueOf`.

"How do you customise `JSON.stringify` for a class?" `toJSON()` returning the shape to serialise; a replacer from outside.

"Why does `Object.keys(instance)` not list methods?" Class methods are defined non-enumerable on the prototype.

Key takeaways

Descriptors: `writable` / `enumerable` / `configurable`; literals default true, `defineProperty` defaults false; accessors have `get` / `set`.

`preventExtensions` ⊂ `seal` ⊂ `freeze`, all shallow; class methods are non-enumerable.

ToPrimitive: `Symbol.toPrimitive(hint)` wins; else string hint → `toString` first, number/default → `valueOf` first; `+` and `==` use default.

`toJSON` decides what `JSON.stringify` sees; replacers and revivers do it from outside.

`Symbol.toStringTag` labels `[object X]`; none of these replace `instanceof`.

EXERCISE 1

A descriptor report

Parse a JSON record. Define `id` with `Object.defineProperty` and only `{ value: "rec-1" }` (so it is read-only, hidden and permanent), and `size` as an **enumerable getter** returning the number of keys `Object.keys(this)` reports. Print `keys=`, `names=` (`Object.getOwnPropertyNames`), `json=`; then attempt `record.id = "other"` and `delete record.id` in `try` / `catch` and print `writeId=<changed|TypeError> deleteId=<deleted|TypeError> id=<record.id>`; then `descriptor(id)=<JSON of the descriptor>`; then freeze the record, try adding a property, and print `frozen=<> add=<added|TypeError> sizeStillComputed=<record.size>`.

Example: `{"name":"ada","born":1815}` →

```
keys=name,born,size
names=name,born,id,size
json={"name":"ada","born":1815,"size":3}
writeId=TypeError deleteId=TypeError id=rec-1
descriptor(id)={"value":"rec-1","writable":false,"enumerable":false,"configurable":fals
frozen=true add=TypeError sizeStillComputed=3
```

INPUT 1

```
{"name":"ada","born":1815}
```

OUTPUT 1

```
keys=name,born,size
names=name,born,id,size
json={"name":"ada","born":1815,"size":3}
writeId=TypeError deleteId=TypeError id=rec-1
descriptor(id)={"value":"rec-
1","writable":false,"enumerable":false,"configurable":false}
frozen=true add=TypeError sizeStillComputed=3
```

REFERENCE SOLUTION

```
"use strict";
const record = JSON.parse(require("fs").readFileSync(0, "utf8"));
Object.defineProperty(record, "id", { value: "rec-1" });                        // writa
Object.defineProperty(record, "size", { get() { return Object.keys(this).length; }, enu
console.log(`keys=${Object.keys(record).join(",")}`);
console.log(`names=${Object.getOwnPropertyNames(record).join(",")}`);
console.log(`json=${JSON.stringify(record)}`);
let write;
try { record.id = "other"; write = "changed"; } catch (e) { write = e.constructor.name;
let remove;
try { delete record.id; remove = "deleted"; } catch (e) { remove = e.constructor.name;
console.log(`writeId=${write} deleteId=${remove} id=${record.id}`);
const d = Object.getOwnPropertyDescriptor(record, "id");
console.log(`descriptor(id)=${JSON.stringify(d)}`);
Object.freeze(record);
let add;
try { record.extra = 1; add = "added"; } catch (e) { add = e.constructor.name; }
console.log(`frozen=${Object.isFrozen(record)} add=${add} sizeStillComputed=${record.si
```

EXERCISE 2

Money that sorts, adds and serialises

Give `Money(cents, currency)` a `valueOf()` returning the cents, a `toString()` returning `<amount 2dp> <currency>`, a `toJSON()` returning `{ amount, currency }` with amount in units, and `get [Symbol.toStringTag]()` returning `"Money"`. Read cents values, build the objects, and print: `template=<a inside a template literal> string=<String(b)>`; `plus=<a + 1> minus=<a - b> greater=<a > b>`; `sorted=<copies sorted by (x, y) ⇒ x - y, as strings joined by " | ">`; `json=<JSON.stringify(all)>`; `tag=<Object.prototype.toString.call(a)> total=<reduce((s, m) ⇒ s + m, 0)>`.

Example: `1250 999 100` →

```
template=12.50 USD string=9.99 USD
plus=1251 minus=251 greater=true
sorted=1.00 USD | 9.99 USD | 12.50 USD
json=[{"amount":12.5,"currency":"USD"},{"amount":9.99,"currency":"USD"},{"amount":1,"cu
tag=[object Money] total=2349
```

Template literals take the string hint (`toString`); `+`, `-` and `>` take `valueOf`.

INPUT 1

```
1250 999 100
```

OUTPUT 1

```
template=12.50 USD string=9.99 USD
plus=1251 minus=251 greater=true
sorted=1.00 USD | 9.99 USD | 12.50 USD
json=[{"amount":12.5,"currency":"USD"},
{"amount":9.99,"currency":"USD"},
{"amount":1,"currency":"USD"}]
tag=[object Money] total=2349
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
class Money {
  constructor(cents, currency = "USD") { this.cents = cents; this.currency = currency;
  valueOf() { return this.cents; }
  toString() { return `${(this.cents / 100).toFixed(2)} ${this.currency}`; }
  toJSON() { return { amount: this.cents / 100, currency: this.currency }; }
  get [Symbol.toStringTag]() { return "Money"; }
}
const amounts = tokens.map(Number).map((c) => new Money(c));
const [a, b] = amounts;
console.log(`template=${a} string=${String(b)}`);
console.log(`plus=${a + 1} minus=${a - b} greater=${a > b}`);
console.log(`sorted=${[...amounts].sort((x, y) => x - y).map(String).join(" | ")}`);
console.log(`json=${JSON.stringify(amounts)}`);
console.log(`tag=${Object.prototype.toString.call(a)} total=${amounts.reduce((s, m) =>
```

QUIZ

1. `Object.defineProperty(o, "k", { value: 1 })` creates a property that is…

A. Normal — writable, enumerable, configurable

B. Read-only, non-enumerable and non-configurable — every unspecified flag defaults to `false`

C. Writable but hidden

D. A getter

2. `${obj}` with no `Symbol.toPrimitive` calls…

A. `valueOf` then `toString`

B. `toString` first (string hint), then `valueOf`

C. `toJSON`

D. Neither — it prints `[object Object]` always

3. `JSON.stringify(x)` when `x` has a `toJSON()` method…

A. Ignores it

B. Serialises whatever `toJSON()` returns instead of `x`

C. Calls it only for Dates

D. Throws

4. `{ ...objWithGetter }` produces…

A. A copy with the same getter

B. A plain data property holding the getter's current value — spread invokes accessors

C. An error

D. An empty object

5. `Object.seal(o)` versus `Object.freeze(o)`:

A. Identical

B. Sealed: no add/delete but values writable; frozen: no add/delete/write

C. Frozen allows adding

D. Sealed is deep

ANSWER KEY · PROPERTY DESCRIPTORS AND THE OBJECT PROTOCOLS — TOSTRING, VALUEOF, TOJSON, SYMBOL.TOPRIMITIVE

1. B Properties created by assignment or literals default all three to `true`.

2. B `obj + 1` uses the default hint: `valueOf` first.

3. B That is how `Date` becomes an ISO string and how a class exposes private state on purpose.

4. B `Object.defineProperties(target, Object.getOwnPropertyDescriptors(src))` copies accessors intact.

5. B `preventExtensions` only stops additions; all three are shallow.

CHECKPOINT · 24 MIN

Checkpoint — Prototypes and classes

This checkpoint covers the prototype chain and lookup, `Object.create`, shadowing, `instanceof`; constructor functions and the four steps of `new`; class syntax with fields, private members, accessors and statics; `extends`, `super`, construction order and extending built-ins; property descriptors and the conversion protocols.

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:

- What a write to an inherited property does, and how `in`, `Object.hasOwn` and `Object.keys` differ.
- The four steps of `new`, and what happens when a constructor returns an object.
- Where fields, methods and statics live; what `#private` guarantees; why methods passed as callbacks lose `this`.
- Why `super()` must come first, when subclass fields initialise, and what `super.method()` looks up.
- The three descriptor flags and their defaults; which of `toString` / `valueOf` / `Symbol.toPrimitive` runs for `${x}` and for `x + 1`.

The programs are an inventory built from a small class hierarchy with private state and JSON output, an event emitter with `on` / `once` / `off` / `emit` semantics, and a linked list class with a `size` accessor, static `from`, and `toString`.

EXERCISE 1

An inventory hierarchy

Write `Item(name, price, qty)` with `get value()`, `isExpired(day)` (always false), `describe(day)` → `<name> x<qty> @ <price 2dp> = <value 2dp>` plus `[expired]` when expired, and `toJSON()` → `{ name, price, qty }`; `PerishableItem extends Item` adds `expiresDay`, `isExpired(day)` is `day > expiresDay`, and its `toJSON` spreads `super.toJSON()` plus `expiresDay`; `Inventory` keeps items in a private `Map` by name with `add`, `remove(name)` (returns the Map's boolean), `report(day)` (items sorted by name, then `total=<sum of non-expired values 2dp> items=<n> expired=<n>`) and `toJSON()` returning the items array. Commands: `item n p q`, `perishable n p q d`, `remove n` (prints `removed <n>=<bool>`), `report d`, `json` (prints `JSON.stringify(inv)`).

Example:

```
item pen 1.50 10
perishable milk 2.25 4 3
perishable bread 3.00 2 5
report 4
remove pen
remove ghost
json
```

→

```
bread x2 @ 3.00 = 6.00
milk x4 @ 2.25 = 9.00 [expired]
pen x10 @ 1.50 = 15.00
total=21.00 items=3 expired=1
removed pen=true
removed ghost=false
[{"name":"milk","price":2.25,"qty":4,"expiresDay":3},{"name":"bread","price":3,"qty":2,
```

INPUT 1

```
item pen 1.50 10
perishable milk 2.25 4 3
perishable bread 3.00 2 5
report 4
remove pen
remove ghost
json
```

OUTPUT 1

```
bread x2 @ 3.00 = 6.00
milk x4 @ 2.25 = 9.00 [expired]
pen x10 @ 1.50 = 15.00
total=21.00 items=3 expired=1
removed pen=true
removed ghost=false
[{"name":"milk","price":2.25,"qty":4,"expiresDay":3},
{"name":"bread","price":3,"qty":2,"expiresDay":5}]
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim()
class Item {
  constructor(name, price, qty) { this.name = name; this.price = price; this.qty = qty;
  get value() { return this.price * this.qty; }
  isExpired() { return false; }
  describe(day) { return `${this.name} x${this.qty} @ ${this.price.toFixed(2)} = ${this
  toJSON() { return { name: this.name, price: this.price, qty: this.qty }; }
}
class PerishableItem extends Item {
  constructor(name, price, qty, expiresDay) { super(name, price, qty); this.expiresDay
  isExpired(day) { return day > this.expiresDay; }
  toJSON() { return { ...super.toJSON(), expiresDay: this.expiresDay }; }
}
class Inventory {
  #items = new Map();
  add(item) { this.#items.set(item.name, item); return this; }
  remove(name) { return this.#items.delete(name); }
  get size() { return this.#items.size; }
  report(day) {
    const items = [...this.#items.values()].sort((a, b) => a.name.localeCompare(b.name)
    for (const it of items) console.log(it.describe(day));
    const total = items.filter((it) => !it.isExpired(day)).reduce((s, it) => s + it.val
    console.log(`total=${total.toFixed(2)} items=${items.length} expired=${items.filter
  }
  toJSON() { return [...this.#items.values()]; }
}
const inv = new Inventory();
for (const line of lines) {
  const [cmd, ...args] = line.trim().split(/\s+/);
  if (cmd === "item") inv.add(new Item(args[0], Number(args[1]), Number(args[2])));
  else if (cmd === "perishable") inv.add(new PerishableItem(args[0], Number(args[1]), N
  else if (cmd === "remove") console.log(`removed ${args[0]}=${inv.remove(args[0])}`);
  else if (cmd === "report") inv.report(Number(args[0]));
  else if (cmd === "json") console.log(JSON.stringify(inv));
}
```

EXERCISE 2

An event emitter

Implement `Emitter` with a private `Map` from event name to listener records `{ id, fn, once }`: `on(event, id, fn)` (returns an unsubscribe function), `once(event, id, fn)`, `off(event, id)` (returns whether something was removed), `emit(event, payload)` (calls every listener registered at the time of the call, in registration order, drops `once` listeners after their first call, and returns how many were called) and `count(event)`. Script: `on <event> <id>` / `once <event> <id>` register a listener that prints `<id>:<event>:<payload>`; `off <event> <id>` prints `off <event> <id> → <bool>`; `emit`

`<event> <payload>` prints `emit <event> → <n>` after the listeners ran; `count <event>` prints `count <event> = <n>`.

Example:

```
on save a
on save b
once save c
emit save first
emit save second
off save a
count save
```

→

```
a:save:first
b:save:first
c:save:first
emit save → 3
a:save:second
b:save:second
emit save → 2
off save a → true
count save = 1
```

INPUT 1

```
on save a
on save b
once save c
emit save first
emit save second
off save a
off save zzz
count save
emit save third
count load
```

OUTPUT 1

```
a:save:first
b:save:first
c:save:first
emit save → 3
a:save:second
b:save:second
emit save → 2
off save a → true
off save zzz → false
count save = 1
b:save:third
emit save → 1
count load = 0
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim()
class Emitter {
  #listeners = new Map();   // event → [{ id, fn, once }]
  #list(event) {
    if (!this.#listeners.has(event)) this.#listeners.set(event, []);
    return this.#listeners.get(event);
  }
  on(event, id, fn, once = false) {
    this.#list(event).push({ id, fn, once });
    return () => this.off(event, id);
  }
  once(event, id, fn) { return this.on(event, id, fn, true); }
  off(event, id) {
    const list = this.#list(event);
    const at = list.findIndex((l) => l.id === id);
    if (at !== -1) list.splice(at, 1);
    return at !== -1;
  }
  emit(event, payload) {
    const list = [...this.#list(event)];          // a copy: listeners may unsubscribe
    for (const l of list) {
      l.fn(payload);
      if (l.once) this.off(event, l.id);
    }
    return list.length;
  }
  count(event) { return this.#list(event).length; }
}
const em = new Emitter();
for (const line of lines) {
  const [cmd, event, arg] = line.trim().split(/\s+/);
  if (cmd === "on") em.on(event, arg, (p) => console.log(`${arg}:${event}:${p}`));
  else if (cmd === "once") em.once(event, arg, (p) => console.log(`${arg}:${event}:${p}
  else if (cmd === "off") console.log(`off ${event} ${arg} → ${em.off(event, arg)}`);
  else if (cmd === "emit") console.log(`emit ${event} → ${em.emit(event, arg)}`);
  else if (cmd === "count") console.log(`count ${event} = ${em.count(event)}`);
}
```

EXERCISE 3

A linked list class

Implement `LinkedList` over `Node { value, next }` with private `#head` / `#size`: `get size`, `static from(iterable)`, `push`, `unshift`, `insertAt(i, v)` and `removeAt(i)` (both throw `RangeError("index out of range")` for bad indices; `removeAt` returns the removed value), `indexOf(v)` (−1 when absent), `reverse()` in place, a generator `[Symbol.iterator]` yielding values, and `toString()` → `[v1 → v2 → …] size=<n>` (`[] size=0` when empty). Commands: `push v`, `unshift v`, `insert i v`, `remove i`

(prints `removed=<v>`), `find v` (prints `index=<i>`), `reverse`, `show` (prints `String(list)`), `sum` (prints `sum=<total via the iterator>`); print `error: <message>` for a thrown error.

Example: `push 1`, `push 2`, `push 3`, `unshift 0`, `show`, `insert 2 9`, `show`, `remove 0`, `find 9`, `reverse`, `show`, `sum`, `remove 10` →

```
[0 → 1 → 2 → 3] size=4
[0 → 1 → 9 → 2 → 3] size=5
removed=0
index=1
[3 → 2 → 9 → 1] size=4
sum=15
error: index out of range
```

INPUT 1

```
push 1
push 2
push 3
unshift 0
show
insert 2 9
show
remove 0
find 9
find 42
reverse
show
sum
remove 10
```

OUTPUT 1

```
[0 → 1 → 2 → 3] size=4
[0 → 1 → 9 → 2 → 3] size=5
removed=0
index=1
index=-1
[3 → 2 → 9 → 1] size=4
sum=15
error: index out of range
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim()
class Node { constructor(value, next = null) { this.value = value; this.next = next; }
class LinkedList {
  #head = null;
  #size = 0;
  get size() { return this.#size; }
  static from(iterable) {
    const list = new LinkedList();
    for (const v of iterable) list.push(v);
    return list;
  }
  #nodeAt(i) {
    let node = this.#head;
    for (let k = 0; k < i; k++) node = node.next;
    return node;
  }
  push(value) { return this.insertAt(this.#size, value); }
  unshift(value) { return this.insertAt(0, value); }
  insertAt(i, value) {
    if (i < 0 || i > this.#size) throw new RangeError("index out of range");
    if (i === 0) this.#head = new Node(value, this.#head);
    else { const prev = this.#nodeAt(i - 1); prev.next = new Node(value, prev.next); }
    this.#size++;
    return this;
  }
  removeAt(i) {
    if (i < 0 || i >= this.#size) throw new RangeError("index out of range");
    let removed;
    if (i === 0) { removed = this.#head; this.#head = removed.next; }
    else { const prev = this.#nodeAt(i - 1); removed = prev.next; prev.next = removed.r
    this.#size--;
    return removed.value;
  }
  indexOf(value) {
    let i = 0;
    for (const v of this) { if (v === value) return i; i++; }
    return -1;
  }
  reverse() {
    let prev = null, node = this.#head;
    while (node !== null) { const next = node.next; node.next = prev; prev = node; node
    this.#head = prev;
    return this;
  }
  *[Symbol.iterator]() { for (let n = this.#head; n !== null; n = n.next) yield n.value
  toString() { return `[${[...this].join(" → ")}] size=${this.#size}`; }
}
const list = new LinkedList();
for (const line of lines) {
```

```
  const [cmd, a, b] = line.trim().split(/\s+/);
  try {
    if (cmd === "push") list.push(Number(a));
    else if (cmd === "unshift") list.unshift(Number(a));
    else if (cmd === "insert") list.insertAt(Number(a), Number(b));
    else if (cmd === "remove") console.log(`removed=${list.removeAt(Number(a))}`);
    else if (cmd === "find") console.log(`index=${list.indexOf(Number(a))}`);
    else if (cmd === "reverse") list.reverse();
    else if (cmd === "show") console.log(String(list));
    else if (cmd === "sum") console.log(`sum=${[...list].reduce((s, v) => s + v, 0)}`);
  } catch (e) {
    console.log(`error: ${e.message}`);
  }
}
```

CHECKPOINT QUESTIONS

1. `Object.getPrototypeOf(Object.getPrototypeOf([]))` is…

A. `Array.prototype`

B. `Object.prototype`

C. `null`

D. `Function.prototype`

2. `const o = Object.create(null); o.toString()` …

A. Returns `"[object Object]"`

B. Throws — the chain is empty, so there is no `toString`

C. Returns `""`

D. Returns `undefined`

3. `function F() { return { a: 1 }; } new F() instanceof F` is…

A. `true`

B. `false` — the returned object replaced the instance and has a plain prototype

C. An error

D. `undefined`

4. `p.constructor === Point` is `true` because…

A. Every object stores its constructor

B. `p` inherits `constructor` from `Point.prototype`, which was created with it

C. `new` sets it on the instance

D. It is an own property of `p`

5. Arrow-function class fields (`handle = () => {…}`) are used to…

A. Make methods static

B. Bind `this` per instance so the method survives being passed as a callback — at one closure per instance

C. Make methods private

D. Speed up calls

6. `#x in obj` (Node 16.4+) tests…

A. Whether `obj` has any private field

B. Whether `obj` carries this class's `#x` brand — the safe "is this one of ours" check

C. Whether `x` is enumerable

D. Nothing — it is a syntax error

7. A subclass with no constructor of its own…

A. Cannot be instantiated

B. Gets `constructor(...args) { super(...args); }` implicitly

C. Skips the base constructor

D. Must define fields

8. `class S extends Array {}; S.from([1]).map(x ⇒ x) instanceof S` is…

A. `false`

B. `true` — array methods build results through `Symbol.species`, the subclass by default

C. An error

D. Only in strict mode

9. Which flag decides whether a property appears in `Object.keys`, spread and `JSON.stringify`?

A. `writable`

B. `enumerable`

C. `configurable`

D. `value`

10. `date1 - date2` gives milliseconds but `${date1}` gives readable text because…

A. Dates are strings

B. `-` takes the number hint (`valueOf`) and template literals the string hint (`toString`)

C. Subtraction calls `toJSON`

D. It is special-cased in the engine

11. When does `JSON.stringify` omit a property entirely?

A. When the value is `null`

B. When the value is `undefined`, a function, or `toJSON` returns `undefined` — and always for non-enumerable and private members

C. Never

D. When the key starts with `_`

12. Composition is preferred over inheritance when…

A. Never — inheritance is always cleaner

B. The relationship is not a genuine is-a, the hierarchy would grow deep, or you only want to reuse a utility method

C. The class has fields

D. The language forbids `extends`

ANSWER KEY · CHECKPOINT — PROTOTYPES AND CLASSES

1. B Array → `Array.prototype` → `Object.prototype` → `null`.

2. B `o.toString` is `undefined`; calling it is a `TypeError`.

3. B Step three of `new`.

4. B Replacing `Point.prototype` wholesale loses it unless restored; it is convention, not a type check.

5. B A prototype method is shared but loses `this` when detached.

6. B Reading `obj.#x` on a foreign object would throw instead.

7. B Arguments pass straight through to the parent.

8. B `Array.isArray` is also `true` for the subclass instance.

9. B Class methods are non-enumerable, which is why instances serialise to their fields only.

10. B `Date` is the one built-in whose default hint acts like string, so `date + 1` concatenates.

11. B `null` and `NaN` (as `null`) are kept.

12. B Extend for shallow, genuine is-a relationships; otherwise hold an instance and delegate.
