Module 6 of 16

Collections and iteration

Map and Set with their idioms and hand-written algebra; WeakMap, WeakSet and WeakRef; the iteration protocol behind for…of and spread; generators and lazy pipelines; symbols as unique keys and the language's hooks.

Arrays and objects carry most programs, but the moment a key is not a string, a value must be unique, or a sequence should be computed on demand, JavaScript has better tools — and one small protocol that ties them together. `Map` and `Set` are real keyed collections with any key type, guaranteed order and O(1) operations; `WeakMap` attaches data to objects without keeping them alive; and everything that can be looped with `for…of`, spread or destructuring follows the iteration protocol: a `[Symbol.iterator]` method returning an object with `next()`. Implement it once and every piece of iteration syntax works on your class.

The module opens with Map and Set — when they beat objects and arrays, how keys compare, the frequency-count and group-by idioms, and the set algebra you must write yourself on Node 16. Weak collections follow, with the three patterns they exist for and why they cannot be iterated. The protocol lesson states the contract exactly and covers laziness and early exit; the generator lesson makes it easy — `yield`, `yield*`, sending values in, and the `take` / `map` / `filter` pipeline that touches only what is consumed. Symbols close the module: unique keys hidden from enumeration, the global registry, and the well-known symbols that let objects plug into `instanceof`, string conversion and concatenation.

The exercises count words two ways to expose the `__proto__` hazard, write set algebra, tag objects through a WeakMap, build a `Range` iterator by hand and a lazy pipeline with generators, and finish with an LRU cache on Map order, a graph explorer with a breadth-first generator, and an iterator toolkit.

CONTENTS

1	Map and Set — real keyed collections	12 min
2	WeakMap, WeakSet and WeakRef — data that does not keep objects alive	11 min
3	The iteration protocol — what for…of, spread and destructuring actually call	13 min
4	Generators — functions that pause, and lazy pipelines	14 min

5	Symbols — unique keys and the language's extension points	11 min
✓	Checkpoint — Collections and iteration	24 min

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

LESSON 1 · 12 MIN

Map and Set — real keyed collections

For twenty years JavaScript had one collection, the array, and one dictionary, the object — and the object was a poor dictionary: keys became strings, `"constructor"` was already "there", `__proto__` was a security hole, and getting the size meant `Object.keys(o).length`. `Map` and `Set` (2015) are the real thing: any value as a key, insertion order guaranteed, a `size`, O(1) operations, and iteration built in. This lesson is when to use each, the operations, the key-equality rule, the set algebra you have to write yourself on Node 16, and the frequency-count and group-by idioms that make up most of their use.

Map

```
const m = new Map();
m.set("a", 1).set("b", 2);              // set returns the map — chainable
m.get("a");          // 1
m.get("zzz");        // undefined
m.has("b");          // true
m.delete("a");       // true if it was there
m.size;              // 1  (a property, not a method — not length, not size()
m.clear();

const fromPairs = new Map([["x", 1], ["y", 2]]);
const fromObject = new Map(Object.entries({ x: 1, y: 2 }));
Object.fromEntries(fromPairs);          // back to an object — only for stri
```

Keys can be **anything**: objects, functions, `NaN`, `null`. `m.set(someDomNode, data)` attaches data to an object without touching it; `m.set(NaN, 1); m.get(NaN)` works because Map uses **SameValueZero**: `===` except that `NaN` equals `NaN` (and `+0` equals `-0`). Objects are keys by identity — two different `{}` are two keys, and a key you cannot recreate (a fresh literal) can only be found via the reference you kept.

Iteration: `for (const [k, v] of m)`, `m.keys()`, `m.values()`, `m.entries()` (the default), `m.forEach((v, k) ⇒ …)` — note value first. Order is **insertion order**, always, for every key type. `[...m]` gives the pairs; `[...m.keys()]` the keys.

Set

```
const s = new Set([1, 2, 2, 3]);          // {1, 2, 3}
s.add(4).add(1);                           // add ignores duplicates, returns t
s.has(2); s.delete(2); s.size;
[...new Set(arr)]                          // the dedupe idiom
const seen = new Set(); items.filter((x) => !seen.has(x.id) && seen.add(x.id))
```

Same equality rule (`NaN` is one member), same insertion order, same iteration (`for…of` , `values()` , `forEach`). A Set of objects holds them by identity — `new Set([{}, {}]).size` is 2.

Set algebra, by hand

Node 22 gained `union` , `intersection` , `difference` , `symmetricDifference` , `isSubsetOf` and friends; Node 16 has none, so know the one-liners:

```
const union = new Set([...a, ...b]);
const intersection = new Set([...a].filter((x) => b.has(x)));
const difference = new Set([...a].filter((x) => !b.has(x)));
const symmetric = new Set([...[...a].filter((x) => !b.has(x)), ...[...b].filte
const isSubset = [...a].every((x) => b.has(x));
```

`b.has` is O(1), so each is O(n); the equivalent with arrays and `includes` is O(n²) and the classic "why is this slow on 50k items" bug.

Map versus object

	`Map`	plain object
Key types	any value	strings and symbols (others coerced)
Prototype hazards	none	`constructor`, `toString`, `__proto__`
Size	`.size`	`Object.keys(o).length` — O(n)
Order	insertion, always	integer-like keys first, then insertion
Iteration	direct	via `Object.entries`
JSON	not directly (`Object.fromEntries` first)	native
Literal syntax	no	yes
Hot-path perf	fast for churn (adds/deletes)	fast for fixed shapes (hidden classes)

Rule of thumb: a **record** with known fields is an object; a **dictionary** keyed by data — especially user data — is a Map. Objects still win where JSON in and out is the whole job.

The idioms

```
// frequency count
const counts = new Map();
for (const w of words) counts.set(w, (counts.get(w) ?? 0) + 1);

// group by
const groups = new Map();
for (const u of users) {
  if (!groups.has(u.city)) groups.set(u.city, []);
  groups.get(u.city).push(u);
}

// sort a map's entries
[...counts].sort(([, a], [, b]) => b - a);      // by value desc
new Map([...m].sort(([a], [b]) => a.localeCompare(b)));   // a sorted copy

// index by id
const byId = new Map(users.map((u) => [u.id, u]));

// cache
const cache = new Map();
function slow(x) { if (!cache.has(x)) cache.set(x, compute(x)); return cache.
```

A Map also solves "delete while iterating": deleting the current entry inside `for…of` over a Map is safe (the iterator skips it), where mutating an array during iteration is not.

Performance and memory

Maps are hash tables: `get` / `set` / `has` / `delete` are O(1) amortised; iteration is O(n) in insertion order. A `Map` with a million entries is fine; a `Map` used for a five-field record is noise. Memory per entry is a few dozen bytes more than an object property. `Set` membership beats `Array.includes` from a few dozen elements up.

Common mistakes

`m.size()` (it is a property), `m.length` (undefined), `m[key]` (sets a property on the Map object, not an entry — silently wrong).

Expecting `m.get({ id: 1 })` to find an entry set with a *different* `{ id: 1 }` object — identity, not structure.

`JSON.stringify(map)` → `{}`; convert first.

Using `Object.keys(map)` (empty) instead of `[...map.keys()]`.

Arrays + `includes` inside a loop where a `Set` was the right structure.

Interview angle

"Map versus object?" Any key type, no prototype collisions, `size`, insertion order, faster churn — a dictionary; objects for records and JSON.

"How does Map compare keys?" SameValueZero: `===` plus `NaN` equals `NaN`; objects by identity.

"Count word frequencies." `counts.set(w, (counts.get(w) ?? 0) + 1)`.

"Intersection of two arrays efficiently?" Put one in a Set, filter the other with `has` — O(n) instead of O(n²).

"Why is `Set` iteration order defined?" The spec requires insertion order for both Map and Set.

Key takeaways

`Map`: any key, `.size`, insertion order, `get/set/has/delete`, iterate entries directly; `Set`: unique values, same rules.

Keys compare with SameValueZero; objects by identity.

Set algebra is hand-written on Node 16 — always via `has`, never `includes` in a loop.

Idioms: frequency count, group by, index by id, cache, dedupe.

Records are objects; dictionaries keyed by data are Maps; convert for JSON.

EXERCISE 1

Word frequencies, two ways

Count the words on the input line with a `Map` and — for comparison — with a plain object dictionary (`dict[w] = (dict[w] ?? 0) + 1`). Print `top=<the three most frequent as word=count, by count descending then word ascending>`, then `unique=<map.size> total=<word count>`, then `objectKeys=<Object.keys(dict).length> mapHasProto=<map.has("__proto__")> objectHasOwnProto=<Object.hasOwn(dict, "__proto__")>`.

Example: `the cat and the dog and the bird` →

```
top=the=3 and=2 bird=1
unique=5 total=8
objectKeys=5 mapHasProto=false objectHasOwnProto=false
```

With `__proto__ x __proto__ y` the Map reports 3 unique words and the object only 2 keys — the string `__proto__` is never an ordinary key on a plain object.

INPUT 1

```
the cat and the dog and the bird
```

OUTPUT 1

```
top=the=3 and=2 bird=1
unique=5 total=8
objectKeys=5 mapHasProto=false
objectHasOwnProto=false
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const counts = new Map();
const dict = {};
for (const w of tokens) {
  counts.set(w, (counts.get(w) ?? 0) + 1);
  dict[w] = (dict[w] ?? 0) + 1;      // the object version: __proto__ never becomes
}
const top = [...counts].sort(([wa, a], [wb, b]) => b - a || wa.localeCompare(wb)).sl
console.log(`top=${top.map(([w, n]) => `${w}=${n}`).join(" ")}`);
console.log(`unique=${counts.size} total=${tokens.length}`);
console.log(`objectKeys=${Object.keys(dict).length} mapHasProto=${counts.has("__prot
```

EXERCISE 2

Set algebra by hand

Two lines of numbers (the token `NaN` becomes `NaN`) form sets `a` and `b`. Using only `Set` operations and `has`, compute the union, intersection, difference `a - b`, symmetric difference and whether `a` is a subset of `b`. The starter's `show` prints a set sorted numerically with `NaN` last. Print `a=<> b=<>`, then one line each for `union`, `intersection`, `difference`, `symmetric`, then `subset=<bool>` `sizes=<a.size>,<b.size>`.

Example: `1 2 3 4 NaN` / `3 4 5 NaN NaN` →

```
a=[1,2,3,4,"NaN"] b=[3,4,5,"NaN"]
union=[1,2,3,4,5,"NaN"]
intersection=[3,4,"NaN"]
difference=[1,2]
symmetric=[1,2,5]
subset=false sizes=5,4
```

`NaN` is one member and matches itself — Set uses SameValueZero, not `===`.

INPUT 1

```
1 2 3 4 NaN
3 4 5 NaN NaN
```

OUTPUT 1

```
a=[1,2,3,4,"NaN"] b=[3,4,5,"NaN"]
union=[1,2,3,4,5,"NaN"]
intersection=[3,4,"NaN"]
difference=[1,2]
symmetric=[1,2,5]
subset=false sizes=5,4
```

INPUT 2

```
1 2
1 2 3
```

OUTPUT 2

```
a=[1,2] b=[1,2,3]
union=[1,2,3]
intersection=[1,2]
difference=[]
symmetric=[3]
subset=true sizes=2,3
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const a = new Set(lines[0].trim().split(/\s+/).map(Number));
const b = new Set(lines[1].trim().split(/\s+/).map(Number));
const show = (s) => JSON.stringify([...s].sort((x, y) => Number.isNaN(x) - Number.is
const union = new Set([...a, ...b]);
const intersection = new Set([...a].filter((x) => b.has(x)));
const difference = new Set([...a].filter((x) => !b.has(x)));
const symmetric = new Set([...[...a].filter((x) => !b.has(x)), ...[...b].filter((x)
const subset = [...a].every((x) => b.has(x));
console.log(`a=${show(a)} b=${show(b)}`);
console.log(`union=${show(union)}`);
console.log(`intersection=${show(intersection)}`);
console.log(`difference=${show(difference)}`);
console.log(`symmetric=${show(symmetric)}`);
console.log(`subset=${subset} sizes=${a.size},${b.size}`);
```

QUIZ

1. `const m = new Map(); m.set({ id: 1 }, "a"); m.get({ id: 1 })` is…

A. `"a"`

B. `undefined` — object keys compare by identity, and this is a different object

C. `null`

D. A `TypeError`

2. How many entries does `new Set([NaN, NaN, 0, -0])` have?

A. 4

B. 2 — SameValueZero treats `NaN` as equal to `NaN` and `0` equal to `-0`

C. 3

D. 1

3. `map.size()` …

A. Returns the count

B. Throws — `size` is a property, not a method

C. Returns `undefined`

D. Returns the capacity

4. The efficient intersection of two large arrays is…

A. `a.filter((x) ⇒ b.includes(x))`

B. Put `b` in a `Set` and `a.filter((x) ⇒ setB.has(x))` — O(n) instead of O(n²)

C. A nested `for` loop

D. `a.concat(b)`

5. Iteration order of a `Map`:

A. Sorted by key

B. Insertion order, for every key type

C. Integer keys first, then strings

D. Unspecified

ANSWER KEY · MAP AND SET — REAL KEYED COLLECTIONS

1. B Keep the reference you used as the key, or key by a primitive id.

2. B Unlike `===`, where `NaN !== NaN`.

3. B `map.length` is `undefined` and `map[key] = v` sets a property on the Map object, not an entry.

4. B `includes` scans the array on every call.

5. B Plain objects put integer-like keys first; Maps never reorder.

LESSON 2 · 11 MIN

WeakMap, WeakSet and WeakRef — data that does not keep objects alive

A `Map` holds its keys strongly: as long as the map exists, every key object stays in memory even if nothing else refers to it. That is what you want for a cache keyed by ids and exactly what you do not want for "extra data about *this* object" — a DOM node's handlers, a parsed result for a request object, a per-instance secret. `WeakMap` and `WeakSet` hold keys **weakly**: when the key object becomes unreachable elsewhere, the entry disappears with it, and no leak accumulates. `WeakRef` and `FinalizationRegistry` (2021) expose the same idea for single references. This lesson covers what the weak collections can and cannot do, the three patterns they exist for, and why you will rarely reach for `WeakRef`.

WeakMap

```
const meta = new WeakMap();
function tag(obj, info) { meta.set(obj, info); }
function infoOf(obj) { return meta.get(obj); }

let node = { id: 1 };
tag(node, { visits: 3 });
infoOf(node);          // { visits: 3 } — node itself is untouched: no new pro
node = null;           // nothing else holds the object → the entry is collect
```

Rules that follow from "weak":

Keys must be objects (or non-registered symbols since 2023; on Node 16, objects only). `meta.set("str", 1)` throws `TypeError: Invalid value used as weak map key` — a primitive cannot be "unreachable", so weak semantics make no sense for it.

Not iterable, no `size`, no `clear`, no `keys()`. If you could list the keys, the collector could never remove one without the program observing it; the API is `get` / `set` / `has` / `delete` and nothing else.

Values are held strongly *while the key lives*. A value that references its own key keeps nothing alive beyond the key's own lifetime — the engine handles ephemeron cycles.

The three patterns

1. Metadata on objects you do not own. Framework state for DOM nodes, parsed bodies for request objects, memoised results keyed by the input object — anywhere a `Map` would pin objects forever and a property on the object would be intrusive or impossible (frozen objects, host objects).

2. Private data before `#fields` — and still, for data private *across* classes or modules:

```
const secrets = new WeakMap();
class User {
  constructor(name, hash) { this.name = name; secrets.set(this, hash); }
  check(hash) { return secrets.get(this) === hash; }
}
```

Only code with access to `secrets` can read the hash. `#fields` (module 5) cover the in-class case more directly; the WeakMap form still wins when several modules cooperate or when you must attach private data to instances of a class you did not write.

3. Memoisation keyed by object identity.

```
const cache = new WeakMap();
function expensive(obj) {
  if (!cache.has(obj)) cache.set(obj, compute(obj));
  return cache.get(obj);
}
```

Same object → same result, computed once; when the caller drops the object, the cached result goes too. A `Map` here is a slow leak in any long-running process.

WeakSet

A set of objects with `add` / `has` / `delete` only. Used for "have I seen/processed this object": marking visited nodes in a traversal over objects you do not own, tracking instances that passed a check (`brandChecks.add(this)` in a constructor, `has(obj)` later), preventing double initialisation. No iteration, no size — if you need to *list* them, you need a `Set` and you accept the pinning.

WeakRef and FinalizationRegistry

```
let big = loadHuge();
const ref = new WeakRef(big);
big = null;
ref.deref();          // the object — or undefined once it has been collected
```

`WeakRef` lets you hold something *without* keeping it alive and check later whether it is still there. `FinalizationRegistry` lets you register a callback that runs *some time after* an object is collected (`registry.register(obj, heldValue)`). Both are deliberately non-deterministic: when collection happens is up to the engine, `deref()` results are stable only within one synchronous job, and a finaliser may run late or never (process exit). The spec's own advice is to avoid them where possible. Legitimate uses are rare and infrastructural: caches of large recomputable objects (images, parsed ASTs) that may be dropped under memory pressure, and releasing *external* resources (a WASM handle, a file descriptor in a native binding) when the JS wrapper is gone — as a backstop, never as the primary cleanup.

Why you cannot observe collection

Garbage collection is not part of the language's observable semantics: no `gc()` call, no event, no "collected" flag. Tests that try to prove a WeakMap entry vanished need `node --expose-gc` and are flaky by nature; the correct test is that the object graph *allows* collection (nothing strong refers to the key). Reason about reachability, not timing.

Common mistakes

- Using a `Map` keyed by objects for per-object data in a long-lived process — a leak that grows for the life of the server.
- Trying to iterate a WeakMap or read its size, then reaching for a Map plus manual deletes (which is a leak again if a delete is missed).
- Primitive keys in a WeakMap.
- `WeakRef` for ordinary caching because it "sounds efficient" — behaviour becomes GC-dependent and hard to test; an LRU `Map` with a size cap is predictable.
- Relying on `FinalizationRegistry` for cleanup that must happen (closing a connection).

Interview angle

"WeakMap versus Map?" Weak keys (objects only) that do not prevent collection; no iteration or size; for metadata, private data and identity-keyed memoisation without leaks.

"Why can't you iterate a WeakMap?" Listing keys would make collection observable; the design forbids it.

"How would you attach data to a DOM node without a property?" A WeakMap keyed by the node.

"When is `WeakRef` appropriate?" Rarely: droppable caches of large recomputable values, backstop cleanup of external resources; never for correctness.

"Can a WeakMap key be a string?" No — `TypeError`; primitives are never unreachable.

Key takeaways

WeakMap/WeakSet hold object keys weakly: entries vanish with their keys; no iteration, no size; objects only.

Use for metadata on foreign objects, cross-module private data, and identity-keyed memoisation.

`WeakRef.deref()` may return `undefined`; `FinalizationRegistry` callbacks are late or never — never depend on them.

Reason about reachability; GC timing is not observable and not testable deterministically.

A `Map` keyed by objects in a long-running process is a leak unless entries are explicitly removed.

EXERCISE 1

Metadata without touching the object

Keep objects in a strong `Map` registry by name and their tags in a `WeakMap` keyed by the object. Commands: `make <name>` creates `{ name }`; `tag <name> <tag>` appends to that object's tag list; `untag <name>` deletes the entry and prints `untag <name> → <bool>`; `info <name>` prints `<name>: tags=<comma list or none> keys=<Object.keys of the object>`; `tagprim <value> <tag>` tries to use the number as a WeakMap key. Print `error: <message>` when a command throws.

Example: `make a`, `tag a red`, `tag a big`, `info a`, `untag a`, `untag a`, `info a`, `tagprim 5 oops` →

```
a: tags=red,big keys=name
untag a → true
untag a → false
a: tags=none keys=name
error: Invalid value used as weak map key
```

The object never gains a property, and a primitive can never be a weak key.

INPUT 1

```
make a
make b
tag a red
tag a big
tag b blue
info a
info b
untag a
untag a
info a
tagprim 5 oops
```

OUTPUT 1

```
a: tags=red,big keys=name
b: tags=blue keys=name
untag a → true
untag a → false
a: tags=none keys=name
error: Invalid value used as weak map
key
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const registry = new Map();     // name → object (strong: the program owns these)
const meta = new WeakMap();     // object → tags (weak: goes away with the object)
for (const line of lines) {
  const [cmd, a, b] = line.trim().split(/\s+/);
  try {
    if (cmd === "make") registry.set(a, { name: a });
    else if (cmd === "tag") {
      const obj = registry.get(a);
      if (!meta.has(obj)) meta.set(obj, []);
      meta.get(obj).push(b);
    } else if (cmd === "untag") console.log(`untag ${a} → ${meta.delete(registry.ge
    else if (cmd === "info") {
      const obj = registry.get(a);
      console.log(`${a}: tags=${meta.has(obj) ? meta.get(obj).join(",") : "none"} ke
    } else if (cmd === "tagprim") meta.set(Number(a), [b]);
  } catch (e) {
    console.log(`error: ${e.message}`);
  }
}
```

EXERCISE 2

Memoise by identity

`sumOfSquares(obj)` sums the squares of `obj.values`; cache its result in a `WeakMap` keyed by the object and count real computations. Commands: `obj <id> <numbers>` creates `{ id, values }`; `clone <id> <newId>` makes a **new** object with the same values; `calc <id>` prints `<id>: <result> (computed|cached)`. Print `computations=<n>` at the end.

Example: `obj a 1 2 3`, `calc a`, `calc a`, `clone a b`, `calc b`, `calc a` →

```
a: 14 (computed)
a: 14 (cached)
b: 14 (computed)
a: 14 (cached)
computations=2
```

The clone has equal contents but a different identity, so it is computed again — a WeakMap keys by reference.

INPUT 1

```
obj a 1 2 3
calc a
calc a
clone a b
calc b
calc a
```

OUTPUT 1

```
a: 14 (computed)
a: 14 (cached)
b: 14 (computed)
a: 14 (cached)
computations=2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
let computations = 0;
const cache = new WeakMap();
function sumOfSquares(obj) {
  if (cache.has(obj)) return { value: cache.get(obj), from: "cached" };
  computations++;
  const value = obj.values.reduce((s, x) => s + x * x, 0);
  cache.set(obj, value);
  return { value, from: "computed" };
}
const objects = new Map();
for (const line of lines) {
  const [cmd, a, ...rest] = line.trim().split(/\s+/);
  if (cmd === "obj") objects.set(a, { id: a, values: rest.map(Number) });
  else if (cmd === "clone") objects.set(rest[0], { ...objects.get(a), id: rest[0] })
  else if (cmd === "calc") {
    const r = sumOfSquares(objects.get(a));
    console.log(`${a}: ${r.value} (${r.from})`);
  }
}
console.log(`computations=${computations}`);
```

QUIZ

1. Why does `WeakMap` have no `size`, `keys()` or iteration?

A. An oversight

B. Listing keys would make garbage collection observable; the API is deliberately `get/set/has/delete` only

C. Performance

D. It has them since Node 16

2. `new WeakMap().set("key", 1)` …

A. Works

B. Throws a `TypeError` — keys must be objects; a primitive can never become unreachable

C. Stores it weakly

D. Converts the string to an object

3. A long-running server stores per-request data in a `Map` keyed by the request object. The consequence is…

A. Nothing — requests are small

B. A memory leak: the Map keeps every request object alive forever unless entries are deleted

C. A `TypeError`

D. Faster lookups

4. `WeakRef#deref()` may return…

A. Always the object

B. The object, or `undefined` once it has been collected — timing is up to the engine

C. A copy

D. `null`

5. Which is **not** a good WeakMap use?

A. Metadata on DOM nodes

B. Cross-module private data for class instances

C. A list of all live sessions that must be enumerated for a dashboard

D. Memoisation keyed by an input object

ANSWER KEY · WEAKMAP, WEAKSET AND WEAKREF — DATA THAT DOES NOT KEEP OBJECTS ALIVE

1. B If you need to enumerate, you need a `Map` — and you accept that it pins its keys.

2. B Weak semantics only make sense for values that can be collected.

3. B A `WeakMap` releases each entry when the request object is dropped.

4. B Never depend on it for correctness; it suits droppable caches and backstop cleanup only.

5. C Enumeration needs a `Set` / `Map`; a WeakSet cannot be listed.

LESSON 3 · 13 MIN

The iteration protocol — what for…of, spread and destructuring actually call

`for…of`, `[...x]`, `const [a, b] = x`, `Array.from(x)`, `new Map(x)`, `Promise.all(x)`, `yield*` — every one of these works on *any* object that follows one small contract, the **iteration protocol**. Arrays, strings, Maps, Sets, `arguments`, typed arrays, DOM lists and generator objects follow it; plain objects do not, which is why `for (const x of {a: 1})` throws. Implement the contract on your own class and all of that syntax works on it for free. This lesson states the protocol precisely, shows a hand-written iterator, and covers the details that matter in practice: laziness, early exit, and what is and is not iterable.

Two interfaces

An object is **iterable** if it has a method under the key `Symbol.iterator` that returns an **iterator**. An iterator is any object with a `next()` method returning `{ value, done }`:

```
const iterable = {
  from: 1, to: 3,
  [Symbol.iterator]() {                    // called once per loop
    let current = this.from, last = this.to;
    return {                               // the iterator — fresh state per
      next() {
        return current ⩽ last ? { value: current++, done: false } : { value:
      },
    };
  },
};
[...iterable];                 // [1, 2, 3]
for (const n of iterable) { }  // 1, 2, 3
const [first] = iterable;      // 1
Math.max(...iterable);         // 3
```

`for…of` does exactly: call `x[Symbol.iterator]()`, then call `next()` repeatedly, binding `value` until `done` is `true`. The `value` on the final `done: true` result is ignored by `for…of` and spread (it is the *return value*, visible only to `yield*` and manual callers). Because the iterator is created per loop, you can iterate the same iterable many times and each loop starts fresh — as long as `[Symbol.iterator]` returns a **new** iterator each call.

Iterators are often iterable too

The iterator objects the built-ins hand back (`arr.values()`, `map.entries()`, generator objects) also have a `[Symbol.iterator]` that returns `this`, so you can pass them straight to `for…of` — but they are **single-use**: once exhausted, a second loop sees nothing. `const it = m.keys(); [...it]; [...it]` gives the keys and then `[]`. An *iterable* (the Map) restarts; an *iterator* (the result of `keys()`) does not.

Manual iteration

```
const it = "hi"[Symbol.iterator]();
it.next();   // { value: "h", done: false }
it.next();   // { value: "i", done: false }
it.next();   // { value: undefined, done: true }
```

Manual `next()` is how you pull *some* of a sequence — the first item and then the rest through a different path, or two iterators zipped in lockstep, or a parser that reads one token ahead. Everything higher-level is built from this.

What is iterable

Iterable: `Array`, `String` (by **code point**, so emoji are one item — module 10), `Map`, `Set`, `arguments`, typed arrays, `NodeList` / `HTMLCollection` in browsers, generator objects, the results of `keys()/values()/entries()`. **Not** iterable: plain objects (use `Object.entries(obj)` — an array of pairs, which is), numbers, `null` / `undefined` (spreading either into an array throws; spreading into an *object* does not). `Array.from(x)` accepts iterables **and** array-likes (objects with `length` and indices), which is why it works on `{ length: 3 }` where spread does not.

Laziness and early exit

Values are produced **on demand**: `next()` computes the next one only when asked. An iterator over a file, a database cursor, or an infinite sequence is fine as long as consumers stop. `for…of` stops when you `break`, `return` or `throw` — and then calls the iterator's optional `return()` method so it can release resources (close the file, clear a timer).

Destructuring a prefix (`const [a, b] = it`) also calls `return()`. Spread and `Array.from` consume everything — never spread an infinite iterator.

```
function* watched() { try { yield 1; yield 2; yield 3; } finally { console.log
for (const v of watched()) { if (v === 2) break; }   // logs "cleanup" — break
```

Iterables in your own classes

```
class Playlist {
  #tracks = [];
  add(t) { this.#tracks.push(t); return this; }
  [Symbol.iterator]() { return this.#tracks[Symbol.iterator](); }    // delega
  *shuffled() { /* a generator method — next lesson */ }
}
for (const t of new Playlist().add("a").add("b")) { }
```

Delegating to an inner array's iterator is the one-liner; when the sequence is computed (a tree walk, a range, a filtered view) write the iterator by hand or, far more readably, as a generator — the whole next lesson. Exposing iteration rather than the underlying array keeps the array private and lets you change the storage later.

Consumers you already use

`Array.from(it, mapFn)`, `new Set(it)`, `new Map(pairs)`, `Object.fromEntries(pairs)`, `Promise.all(it)`, `Math.max(...it)`, `fn(...it)`, `[...it]`, `const [a, ...rest] = it`, `yield* it`, `for await (const x of asyncIt)`. Node 22's iterator helpers (`it.map().filter().take()`) do not exist on Node 16 — write generators for the same effect.

Common mistakes

- Returning `this` from `[Symbol.iterator]` with state stored on the iterable — the second loop finds it exhausted.
- Spreading a plain object into an array, or a Map into `Object.fromEntries` without checking the keys are strings.

Forgetting that `String` iterates by code point while `.length` and indexing are by UTF-16 unit.

Spreading a possibly-infinite or huge iterator.

Holding an iterator across `break` without letting `return()` run (only a problem when you call `next()` by hand — then call `it.return?.()` yourself).

Interview angle

"What makes an object iterable?" A `[Symbol.iterator]()` method returning an object with `next()` → `{ value, done }`.

"Iterable versus iterator?" Iterable produces iterators (restartable); an iterator is the cursor (single-use); built-in iterators are also iterable by returning themselves.

"Why can't you `for…of` an object?" No `Symbol.iterator`; use `Object.entries`.

"What happens on `break` inside `for…of`?" The iterator's `return()` is called so it can clean up.

"Make a class work with spread and `for…of`." Implement `[Symbol.iterator]`, usually by delegating or with a generator.

Key takeaways

Iterable = has `[Symbol.iterator]()`; iterator = has `next()` returning `{ value, done }`; the protocol is what `for…of`, spread, destructuring and `Array.from` call.

Return a fresh iterator per call so loops restart; built-in iterators are single-use.

Strings iterate by code point; plain objects are not iterable; `Array.from` also takes array-likes.

Iteration is lazy; `break` calls `return()`; never spread the infinite.

Own classes: delegate to an inner iterator or write a generator.

EXERCISE 1

A Range that speaks the protocol

Give `Range(from, to, step)` a hand-written `[Symbol.iterator]()` returning an iterator object with `next()` (values `from`, `from+step`, … while ≤ `to`) and `return()`, which increments the global `returnCalls`. Each call must return a **fresh** iterator so the range can be looped repeatedly. Line 1 is

`from to step`, line 2 a value to break at. Print `spread=<[...r]>`, then `sum=<via for…of> again=<[...r] a second time>`, then `first=<> second=<> squares=<Array.from(r, x ⇒ x*x)>` (destructure the first two), then run `for…of` breaking at the given value and print `breakAt=<v> returnCalled=<how many times return() ran during that loop>`.

Example: `1 10 3` then `4` →

```
spread=[1,4,7,10]
sum=22 again=[1,4,7,10]
first=1 second=4 squares=[1,16,49,100]
breakAt=4 returnCalled=1
```

INPUT 1

```
1 10 3
4
```

OUTPUT 1

```
spread=[1,4,7,10]
sum=22 again=[1,4,7,10]
first=1 second=4 squares=[1,16,49,100]
breakAt=4 returnCalled=1
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) ⇒ l.trim
let returnCalls = 0;
class Range {
  constructor(from, to, step = 1) { this.from = from; this.to = to; this.step = step
  [Symbol.iterator]() {
    let current = this.from;
    const { to, step } = this;
    return {
      next: () ⇒ (current ⩽ to ? { value: (current += step) - step, done: false }
      return: () ⇒ { returnCalls++; return { value: undefined, done: true }; },
    };
  }
}
const [from, to, step] = lines[0].trim().split(/\s+/).map(Number);
const breakAt = Number(lines[1]);
const r = new Range(from, to, step);
console.log(`spread=${JSON.stringify([...r])}`);
let sum = 0;
for (const x of r) sum += x;
console.log(`sum=${sum} again=${JSON.stringify([...r])}`);
const [first, second] = r;
console.log(`first=${first} second=${second} squares=${JSON.stringify(Array.from(r,
const before = returnCalls;
for (const x of r) if (x ⩵ breakAt) break;
console.log(`breakAt=${breakAt} returnCalled=${returnCalls - before}`);
```

EXERCISE 2

What is iterable?

Each line is a JSON value. Decide whether it is iterable (`typeof value[Symbol.iterator] === "function"` , guarding `null`). For an iterable print `<line>: iterable items=<[...value].length> first=<JSON of the first item>` ; for a non-iterable object print `<line>: not iterable entries=<Object.entries(value).length>` ; for anything else `<line>: not iterable (<null or typeof>)` .

Example:

```
[1,2,3]
"abc"
{"a":1,"b":2}
42
null
```

→

```
[1,2,3]: iterable items=3 first=1
"abc": iterable items=3 first="a"
{"a":1,"b":2}: not iterable entries=2
42: not iterable (number)
null: not iterable (null)
```

INPUT 1

```
[1,2,3]
"abc"
{"a":1,"b":2}
42
null
true
[]
```

OUTPUT 1

```
[1,2,3]: iterable items=3 first=1
"abc": iterable items=3 first="a"
{"a":1,"b":2}: not iterable entries=2
42: not iterable (number)
null: not iterable (null)
true: not iterable (boolean)
[]: iterable items=0 first=undefined
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
for (const line of lines) {
  const value = JSON.parse(line);
  const iterable = value != null && value != undefined && typeof value[Symbol.iter
  if (iterable) {
    const items = [...value];
    console.log(`${line.trim()}: iterable items=${items.length} first=${JSON.stringi
  } else if (value != null && typeof value == "object") {
    console.log(`${line.trim()}: not iterable entries=${Object.entries(value).length
  } else {
    console.log(`${line.trim()}: not iterable (${value == null ? "null" : typeof va
  }
}
```

QUIZ

1. An object is iterable when…

A. It has a `length`

B. It has a `[Symbol.iterator]()` method returning an object with `next()` that yields `{ value, done }`

C. It is an array

D. It has a `forEach` method

2. `const it = map.keys(); [...it]; [...it]` — the second spread gives…

A. The keys again

B. `[]` — an iterator is single-use; the Map itself is the restartable iterable

C. An error

D. The values

3. `for (const x of { a: 1 })` …

A. Iterates the values

B. Throws — plain objects are not iterable; use `Object.entries`

C. Iterates the keys

D. Runs zero times

4. `break` inside `for…of` over a custom iterator…

A. Leaves the iterator hanging

B. Calls the iterator's `return()` method, if any, so it can clean up

C. Calls `next()` once more

D. Throws

5. `Array.from({ length: 3 })` works but `[...{ length: 3 }]` throws because…

A. Spread needs numbers

B. `Array.from` accepts array-likes as well as iterables; spread requires an iterable

C. `length` is reserved

D. Spread only works on arrays

ANSWER KEY · THE ITERATION PROTOCOL — WHAT FOR…OF, SPREAD AND DESTRUCTURING ACTUALLY CALL

1. B That single contract powers `for…of`, spread, destructuring and `Array.from`.

2. B Built-in iterators return `this` from `[Symbol.iterator]`, so they can be looped once.

3. B `for…in` walks keys; `for…of` needs the protocol.

4. B Destructuring a prefix does the same; manual `next()` callers must call `return()` themselves.

5. B `Array.from({ length: n }, fn)` is the "n of something" idiom for that reason.

LESSON 4 · 14 MIN

Generators — functions that pause, and lazy pipelines

A generator is a function that can **pause** at `yield`, hand a value out, and later resume exactly where it stopped with all its local variables intact. Call it and nothing runs; you get a generator object that is both an iterator and an iterable. That makes generators the shortest way to implement the iteration protocol — a range, a tree walk, an infinite sequence in three lines — and the natural way to build **lazy pipelines**: `filter`, `map` and `take` that touch only the items actually consumed. They also run backwards: `next(value)` sends data *into* the paused function, which is how coroutines, state machines and (historically) `async` / `await` were built. This lesson covers the syntax, the protocol, delegation with `yield*`, the pipeline toolkit, and where generators earn their keep.

The syntax and the protocol

```
function* countdown(n) {
  while (n > 0) yield n--;
  return "done";                          // the value of the final { done: tr
}
const g = countdown(3);                   // nothing has run yet
g.next();     // { value: 3, done: false }
g.next();     // { value: 2, done: false }
g.next();     // { value: 1, done: false }
g.next();     // { value: "done", done: true }
g.next();     // { value: undefined, done: true } — forever after

[...countdown(3)];                        // [3, 2, 1] — return value is not p
for (const n of countdown(3)) { }
```

`function*` (or `*method()` in a class/object, or `async function*`) marks a generator; arrows cannot be generators. Each `next()` runs the body until the next `yield`, whose operand becomes `value`. A `return` (or falling off the end) sets `done: true`. Generator objects are single-use, like every iterator.

Infinite sequences and `take`

```
function* naturals() { let n = 1; while (true) yield n++; }
function* take(n, it) { for (const x of it) { if (n-- <= 0) return; yield x; }
[...take(5, naturals())];                  // [1, 2, 3, 4, 5]
```

Infinite loops are fine inside a generator because the loop only advances when someone calls `next()`. The consumer decides how much to compute — `take` stops pulling, and `for…of` in `take` calls the source's `return()` on the way out.

Lazy pipelines

```
function* map(fn, it) { for (const x of it) yield fn(x); }
function* filter(pred, it) { for (const x of it) if (pred(x)) yield x; }

const pipeline = take(3, map((x) => x * x, filter((x) => x % 2, naturals())));
[...pipeline];                             // [1, 9, 25] — only 5 naturals were
```

Each stage pulls from the one below it only when pulled itself. Compared with `arr.filter().map().slice(0, 3)`: no intermediate arrays, works on infinite or streaming sources, and stops at the first three results instead of processing everything. For a small in-memory array, the array methods are simpler and faster; for large, lazy or infinite data, generators win. Node 22's iterator helpers (`it.filter().map().take()`) are exactly these functions as methods; on Node 16 you write them once.

`yield*` — delegation

```
function* flatten(tree) {
  for (const node of tree) {
    if (Array.isArray(node)) yield* flatten(node);      // hand control to the
    else yield node;
  }
}
[...flatten([1, [2, [3, [4]]], 5])];      // [1, 2, 3, 4, 5]

function* chain(...its) { for (const it of its) yield* it; }     // works on A
[...chain("ab", [1, 2], new Set([3]))];   // ["a", "b", 1, 2, 3]
```

`yield* inner` yields every value of `inner` as if written inline, forwards `next(value)` / `return()` / `throw()` into it, and evaluates to `inner`'s return value. Recursive tree traversals become a few lines; without `yield*` you would loop and re-yield by hand.

Sending values in: `next(value)`

```
function* average() {
  let sum = 0, count = 0;
  while (true) {
    const x = yield count ? sum / count : 0;     // yield gives out the curre
    sum += x; count++;
  }
}
const avg = average();
avg.next();        // prime: runs to the first yield, value 0 (the argument t
avg.next(10).value;   // 10
avg.next(20).value;   // 15
```

The argument to `next(v)` becomes the value of the `yield` expression the generator is paused on. The first `next()` has no paused `yield` to deliver to, so its argument is dropped — hence the priming call. `gen.return(v)` finishes the generator early (running `finally` blocks) and `gen.throw(err)` raises inside it at the paused `yield` — the two channels that make generators full coroutines, and the foundation of the pre-2017 `async` libraries (`co`, redux-saga).

Generators as class iterables

```
class Tree {
  constructor(value, children = []) { this.value = value; this.children = chi
  *[Symbol.iterator]() {                          // depth-first, pre-order
    yield this.value;
    for (const c of this.children) yield* c;
  }
  *breadthFirst() {
    const queue = [this];
    while (queue.length) { const n = queue.shift(); yield n.value; queue.push(
  }
}
```

A generator method is the idiomatic way to make a class iterable when the sequence is computed rather than stored — and to offer several orders (`breadthFirst()`) as plain methods.

`try` / `finally` and cleanup

Code after a `yield` may never run if the consumer stops early — put cleanup in `finally`; `for…of`'s `break` calls `return()`, which runs it. A generator holding a file handle or a lock should always be structured `try { yield … } finally { release() }`.

Async generators, briefly

`async function*` yields promises' resolved values and is consumed with `for await (const x of gen())` — the pattern for paginated APIs, streams and line-by-line file reading. Module 8 covers them with promises; the mechanics are the ones here plus `await`.

Common mistakes

- Expecting the body to run at call time; it runs at the first `next()`.
- Reusing a generator object for a second loop (single-use); call the generator function again.
- Forgetting the priming `next()` when sending values in.
- Spreading an infinite generator.
- Returning a value and expecting it in `[...gen]` — return values are not yielded.
- Arrow functions with `*` (syntax error) or `yield` inside a nested normal function (it is not the generator).

Interview angle

- *"What is a generator?"* A function that pauses at `yield` and resumes on `next()`, producing an iterator lazily.
- *"Implement `take(n, iterable)`."* A generator with a counter that `return`s when the count runs out.

"Why lazy?" Compute only what is consumed; works on infinite and streaming sources; no intermediate arrays.

"What does `yield*` do?" Delegates to another iterable, yielding its values and forwarding `next/return/throw`.

"How do you send a value into a generator?" `gen.next(v)` — it becomes the value of the paused `yield`; prime first.

Key takeaways

`function*` + `yield`: the body runs on `next()`, pausing at each `yield`; `return` ends it; generator objects are single-use iterators and iterables.

Infinite sequences are fine; `take` / `map` / `filter` generators make lazy pipelines that compute only what is consumed.

`yield*` delegates (recursion, chaining any iterables) and forwards the protocol.

`next(v)` sends values in (prime first); `return` / `throw` finish or raise inside; `finally` for cleanup.

Generator methods are the idiomatic class iterable; async generators pair with `for await`.

EXERCISE 1

A lazy pipeline that proves its laziness

The starter's `naturals()` generator counts how many values it has produced in `pulled`. Write generators `take(n, it)` (yields at most `n` items and stops pulling immediately after the last), `map(fn, it)`, `filter(pred, it)`, and `flatten(list)` (recursive with `yield*`). Line 1 is `k m`: print `result=<first k squares of naturals divisible by m> pulled=<how many naturals were generated>`. Line 2 is a nested JSON array: print `flatten=<JSON>`. Finally drive `take(2, naturals())` by hand and print `manual=<three JSON.stringify(gen.next()) results separated by spaces>`.

Example: `3 4` then `[1,[2,[3,[4]]],5]` →

```
result=[16,64,144] pulled=12
flatten=[1,2,3,4,5]
manual={"value":1,"done":false} {"value":2,"done":false} {"done":true}
```

Only twelve naturals were ever produced for three results — nothing past the twelfth was touched.

INPUT 1

```
3 4
[1,[2,[3,[4]]],5]
```

OUTPUT 1

```
result=[16,64,144] pulled=12
flatten=[1,2,3,4,5]
manual={"value":1,"done":false}
{"value":2,"done":false} {"done":true}
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
let pulled = 0;
function* naturals() { let n = 1; while (true) { pulled++; yield n++; } }
function* take(n, it) { if (n <= 0) return; for (const x of it) { yield x; if (--n =
function* map(fn, it) { for (const x of it) yield fn(x); }
function* filter(pred, it) { for (const x of it) if (pred(x)) yield x; }
function* flatten(list) { for (const x of list) { if (Array.isArray(x)) yield* flatt
const [k, m] = lines[0].trim().split(/\s+/).map(Number);
const nested = JSON.parse(lines[1]);
const result = [...take(k, map((x) => x * x, filter((x) => x % m === 0, naturals())))
console.log(`result=${JSON.stringify(result)} pulled=${pulled}`);
console.log(`flatten=${JSON.stringify([...flatten(nested)])}`);
const g = take(2, naturals());
console.log(`manual=${JSON.stringify(g.next())} ${JSON.stringify(g.next())} ${JSON.s
```

EXERCISE 2

Sending values in

Write a generator `stats()` that receives numbers through `next(x)` and, after each, yields `{ count, sum, min, max, avg }`. The starter primes it with one `next()`. Commands: `add <x>` prints `count=<> sum=<> min=<> max=<> avg=<2dp>` from the yielded object (or `add <x> ignored: generator finished` if it has finished); `done` calls `gen.return()` and prints `done=<r.done> final=<count> values, avg <2dp> afterwards=<JSON.stringify(gen.next(1))>` (`final=nothing` when no values were added).

Example: `add 10`, `add 20`, `add 3`, `done` →

```
count=1 sum=10 min=10 max=10 avg=10.00
count=2 sum=30 min=10 max=20 avg=15.00
count=3 sum=33 min=3 max=20 avg=11.00
done=true final=3 values, avg 11.00 afterwards={"done":true}
```

INPUT 1

```
add 10
add 20
add 3
done
```

OUTPUT 1

```
count=1 sum=10 min=10 max=10 avg=10.00
count=2 sum=30 min=10 max=20 avg=15.00
count=3 sum=33 min=3 max=20 avg=11.00
done=true final=3 values, avg 11.00
afterwards={"done":true}
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
function* stats() {
  let count = 0, sum = 0, min = Infinity, max = -Infinity;
  let x = yield null;                          // the priming next() stops here; its ar
  while (true) {
    count++; sum += x; min = Math.min(min, x); max = Math.max(max, x);
    x = yield { count, sum, min, max, avg: sum / count };
  }
}
const gen = stats();
gen.next();   // prime: run to the first yield
let last = null;
for (const line of lines) {
  const [cmd, value] = line.trim().split(/\s+/);
  if (cmd === "add") {
    const r = gen.next(Number(value));
    if (r.done) { console.log(`add ${value} ignored: generator finished`); continue;
    last = r.value;
    console.log(`count=${last.count} sum=${last.sum} min=${last.min} max=${last.max}
  } else if (cmd === "done") {
    const r = gen.return(last);
    console.log(`done=${r.done} final=${last ? `${last.count} values, avg ${last.avg
  }
}
```

QUIZ

1. Calling a generator function `gen()` …

A. Runs its body to the end

B. Runs nothing and returns a generator object; the body starts on the first `next()`

C. Returns the first yielded value

D. Returns an array

2. `[...(function* () { yield 1; return 2; })()]` is…

A. `[1, 2]`

B. `[1]` — the return value is the `done: true` result and is not part of the sequence

C. `[2]`

D. `[1, undefined]`

3. `yield* inner` …

A. Yields the inner iterable as one value

B. Yields each value of `inner` in turn and forwards `next/return/throw` to it; evaluates to `inner`'s return value

C. Creates a new generator

D. Is a syntax error outside classes

4. Why does sending values into a generator need a priming `next()`?

A. It allocates memory

B. The first `next()` runs to the first `yield`; there is no paused `yield` yet to receive its argument, so that argument is dropped

C. Generators start paused at `return`

D. It does not — the first argument is delivered

5. A pipeline `take(3, map(f, filter(p, infinite())))` terminates because…

A. `infinite()` is not really infinite

B. Each stage pulls one item at a time only when asked; `take` stops asking after three, so nothing beyond is computed

C. `map` buffers everything

D. JavaScript limits loops

ANSWER KEY · GENERATORS — FUNCTIONS THAT PAUSE, AND LAZY PIPELINES

1. B Generator objects are single-use iterators and iterables.

2. B `yield*` and manual callers can see the return value; spread and `for…of` ignore it.

3. B Recursive tree walks and `chain(...iterables)` are the typical uses.

4. B Every later `next(v)` makes `v` the value of the `yield` expression the generator is paused on.

5. B Spreading `infinite()` directly would hang — laziness lives in the consumer's restraint.

LESSON 5 · 11 MIN

Symbols — unique keys and the language's extension points

A symbol is a primitive whose only job is to be **unique**: `Symbol("id") !== Symbol("id")`. Used as a property key it cannot collide with any string key or any other symbol, it is skipped by `Object.keys`, `for…in` and `JSON.stringify`, and it lets two pieces of code attach data to the same object without knowing about each other. The language itself uses a fixed set of **well-known symbols** as hook names — `Symbol.iterator` is the one you have been using all module — so that your objects can plug into `for…of`, `instanceof`, string conversion and more without the language reserving any string names. This lesson covers the primitive, the global registry, symbols as keys, and the well-known symbols worth knowing by name.

The primitive

```
const id = Symbol("user id");          // the string is a description for deb
typeof id;                             // "symbol"
id.description;                        // "user id"
id.toString();                         // "Symbol(user id)"
String(id);                            // same; `${id}` and id + "" THROW — r
Symbol("x") === Symbol("x");           // false — every call is a new symbol
```

Symbols cannot be `new`ed (`new Symbol()` throws), are never coerced to strings implicitly (a guard against accidentally turning a key into `"Symbol(x)"`), and are compared by identity. The seventh primitive type, after `string`, `number`, `bigint`, `boolean`, `undefined`, `null`.

Symbols as property keys

```
const HIDDEN = Symbol("hidden");
const obj = { visible: 1, [HIDDEN]: "secret" };   // computed key syntax
obj[HIDDEN];                                      // "secret" — must use bracket

Object.keys(obj);                       // ["visible"]
JSON.stringify(obj);                    // {"visible":1}
for (const k in obj) { }                // visible only
Object.getOwnPropertySymbols(obj);      // [Symbol(hidden)]
Reflect.ownKeys(obj);                   // ["visible", Symbol(hidden)] — everyth
{ ...obj }                              // copies symbol keys too (own enumerabl
Object.assign({}, obj)                  // same
```

Symbol-keyed properties are **not** private — anyone with `Object.getOwnPropertySymbols` or the symbol itself can read them. They are *collision-free* and *invisible to the common enumeration paths*, which is a different, useful guarantee: a library can stamp `obj[Symbol("cached")]` on your objects without ever clashing with your fields or leaking into your JSON. For true privacy, `#fields` or a WeakMap.

The global registry: `Symbol.for`

```
Symbol.for("app.id") === Symbol.for("app.id");   // true — looked up (or creat
Symbol.keyFor(Symbol.for("app.id"));             // "app.id"
Symbol.keyFor(Symbol("local"));                  // undefined — not registered
```

`Symbol.for(key)` returns the same symbol for the same string everywhere in the realm — across modules, across bundles, across iframes. Use it when two independently loaded pieces of code must agree on a key (Node's own `Symbol.for("nodejs.util.inspect.custom")` is the canonical example). Use plain `Symbol()` when uniqueness is the point.

Well-known symbols

Built-in symbols stored as static properties of `Symbol`; the language looks them up on your objects at specific moments:

Symbol	The language calls it when…	You implement
`Symbol.iterator`	`for…of`, spread, destructuring, `Array.from`	a method returning an iterator (usually a generator)
`Symbol.asyncIterator`	`for await…of`	an async generator or async iterator
`Symbol.toPrimitive`	conversion to a primitive (`+`, `${}`, `==`)	`(hint) ⇒ primitive` — overrides `valueOf` / `toString`
`Symbol.toStringTag`	`Object.prototype.toString.call(x)`	a getter returning the tag, giving `[object Tag]`
`Symbol.hasInstance`	`x instanceof C`	`static [Symbol.hasInstance](x)` — custom `instanceof`
`Symbol.species`	array/promise methods creating derived objects	a static getter returning the constructor to use
`Symbol.isConcatSpreadable`	`arr.concat(x)`	`true` to spread an array-like, `false` to keep an array whole
`Symbol.match/replace/search/split`	string methods given a non-RegExp	objects that act like patterns
`Symbol.unscopables`	`with` statements	ignore

```
class Even {
  static [Symbol.hasInstance](n) { return Number.isInteger(n) && n % 2 === 0;
}
4 instanceof Even;       // true — no Even object was ever created

const arrayLike = { length: 2, 0: "a", 1: "b", [Symbol.isConcatSpreadable]: tr
[1].concat(arrayLike);  // [1, "a", "b"]
```

You will implement `Symbol.iterator` often, `Symbol.toPrimitive` / `toStringTag` sometimes, `hasInstance` and `species` almost never — but recognising them explains behaviour that otherwise looks magical (why `Stack extends Array` methods return `Stack`s: `species`).

Enums and sentinels

Symbols make excellent **sentinel values** — `const MISSING = Symbol("missing")` can never be confused with a real value the way `null` or `-1` can — and readable enums (`const Color = Object.freeze({ Red: Symbol("red"), Green: Symbol("green") })`) whose members cannot be forged from strings. The cost: symbols do not serialise, so anything that crosses JSON needs a string mapping.

Common mistakes

`obj.SYM` instead of `obj[SYM]` (creates a string-keyed property).

Expecting `JSON.stringify` or `Object.keys` to show symbol keys — or, the reverse, expecting symbol keys to be private.

`` `${sym}` `` — throws; use `sym.description` or `String(sym)`.

Using `Symbol()` where two modules need the *same* key (use `Symbol.for`), or `Symbol.for` where uniqueness matters.

Storing symbols in something that will be serialised.

Interview angle

"What is a symbol for?" A unique, collision-free property key hidden from ordinary enumeration; and the language's hook names (well-known symbols).

"`Symbol()` versus `Symbol.for()`?" Fresh and unique every call versus one shared symbol per string in a global registry.

"Are symbol properties private?" No — hidden from `keys`/JSON/`for…in`, visible via `getOwnPropertySymbols` / `Reflect.ownKeys`.

"Name three well-known symbols." `iterator`, `toPrimitive`, `hasInstance` (or `toStringTag`, `asyncIterator`, `species`).

"How would you customise `instanceof`?" `static [Symbol.hasInstance](x)` on the class.

Key takeaways

`Symbol(desc)` is unique and never implicitly stringified; `typeof` is `"symbol"`.

As keys: brackets to access; skipped by `keys` / `for…in` / JSON; found by `getOwnPropertySymbols` / `Reflect.ownKeys`; copied by spread.

`Symbol.for` shares one symbol per string across the realm; `Symbol.keyFor` reads it back.

Well-known symbols are hooks: `iterator`, `asyncIterator`, `toPrimitive`, `toStringTag`, `hasInstance`, `species`, `isConcatSpreadable`.

Symbols are ideal sentinels and enum members, and do not survive JSON.

EXERCISE 1

String keys and symbol keys, side by side

Line 1 lists string keys (assign each its index), line 2 lists names for `Symbol(name)` keys (assign `hidden—<i>`). Build one object with both. Print `keys=<Object.keys> json=<JSON.stringify>`; `symbols=<getOwnPropertySymbols count> descriptions=<their descriptions>`; `ownKeys=<Reflect.ownKeys length> spreadKeepsSymbols=<does { ...obj } keep them all>`; then for the first symbol name: `plainEqual=<Symbol(n) === Symbol(n)> forEqual=<Symbol.for(n) === Symbol.for(n)> keyFor=<Symbol.keyFor(Symbol.for(n))> keyForPlain=<Symbol.keyFor of your first plain symbol>`; finally try to put the first symbol in a template literal and print `template=<result or the error's constructor name> string=<String(sym)> typeof=<typeof sym>`.

Example: `name born` / `id secret` →

```
keys=name,born json={"name":0,"born":1}
symbols=2 descriptions=id,secret
ownKeys=4 spreadKeepsSymbols=true
plainEqual=false forEqual=true keyFor=id keyForPlain=undefined
template=TypeError string=Symbol(id) typeof=symbol
```

INPUT 1

```
name born
id secret
```

OUTPUT 1

```
keys=name,born json={"name":0,"born":1}
symbols=2 descriptions=id,secret
ownKeys=4 spreadKeepsSymbols=true
plainEqual=false forEqual=true keyFor=id
keyForPlain=undefined
template=TypeError string=Symbol(id)
typeof=symbol
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const stringKeys = lines[0].trim().split(/\s+/);
const symbolNames = lines[1].trim().split(/\s+/);
const obj = {};
stringKeys.forEach((k, i) => { obj[k] = i; });
const symbols = symbolNames.map((n) => Symbol(n));
symbols.forEach((s, i) => { obj[s] = `hidden-${i}`; });
console.log(`keys=${Object.keys(obj).join(",")} json=${JSON.stringify(obj)}`);
console.log(`symbols=${Object.getOwnPropertySymbols(obj).length} descriptions=${Obje
console.log(`ownKeys=${Reflect.ownKeys(obj).length} spreadKeepsSymbols=${Object.getO
const name = symbolNames[0];
console.log(`plainEqual=${Symbol(name) === Symbol(name)} forEqual=${Symbol.for(name)
let templated;
try { templated = `${symbols[0]}`; } catch (e) { templated = e.constructor.name; }
console.log(`template=${templated} string=${String(symbols[0])} typeof=${typeof symb
```

EXERCISE 2

Well-known symbols in action

Implement three hooks. `Even` gets `static [Symbol.hasInstance](n)` so `4 instanceof Even` is true for even integers. `Version(major, minor)` gets `[Symbol.toPrimitive](hint)` — `"number"` → `major + minor / 100`, otherwise `v<major>.<minor>` — and `get [Symbol.toStringTag]()` returning `"Version"`. Line 1 is numbers, line 2 a version like `1.4`. Print `even=<comma list> odd=<comma list>`; then `string=<template of v> number=<+v> default=<v + ""> compare=<v > 1> tag=<Object.prototype.toString.call(v)>`; then build `{ length: 2, 0: "x", 1: "y", [Symbol.isConcatSpreadable]: true }` and the same object without the symbol and print `concatSpreadable=<JSON of [1].concat(spreadable)> concatPlain=<JSON of [1].concat(plain)>`.

Example: `1 2 3 4 10 7` / `1.4` →

```
even=2,4,10 odd=1,3,7
string=v1.4 number=1.04 default=v1.4 compare=true tag=[object Version]
concatSpreadable=[1,"x","y"] concatPlain=[1,{"0":"x","1":"y","length":2}]
```

INPUT 1

```
1 2 3 4 10 7
1.4
```

OUTPUT 1

```
even=2,4,10 odd=1,3,7
string=v1.4 number=1.04 default=v1.4
compare=true tag=[object Version]
concatSpreadable=[1,"x","y"]
concatPlain=[1,
{"0":"x","1":"y","length":2}]
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
class Even {
  static [Symbol.hasInstance](n) { return Number.isInteger(n) && n % 2 === 0; }
}
class Version {
  constructor(major, minor) { this.major = major; this.minor = minor; }
  [Symbol.toPrimitive](hint) {
    if (hint === "number") return this.major + this.minor / 100;
    return `v${this.major}.${this.minor}`;      // string and default
  }
  get [Symbol.toStringTag]() { return "Version"; }
}
const numbers = lines[0].trim().split(/\s+/).map(Number);
const [major, minor] = lines[1].trim().split(".").map(Number);
console.log(`even=${numbers.filter((n) => n instanceof Even).join(",")} odd=${number
const v = new Version(major, minor);
console.log(`string=${v} number=${+v} default=${v + ""} compare=${v > 1} tag=${Objec
const spreadable = { length: 2, 0: "x", 1: "y", [Symbol.isConcatSpreadable]: true };
const notSpreadable = { length: 2, 0: "x", 1: "y" };
console.log(`concatSpreadable=${JSON.stringify([1].concat(spreadable))} concatPlain=
```

QUIZ

1. `Symbol("id") === Symbol("id")` is…

A. `true`

B. `false` — every `Symbol()` call creates a unique value; the string is only a description

C. `true` in strict mode

D. A `TypeError`

2. Symbol-keyed properties are skipped by…

A. Nothing

B. `Object.keys`, `for…in` and `JSON.stringify` — but found by `Object.getOwnPropertySymbols`, `Reflect.ownKeys` and copied by spread

C. Spread

D. `Reflect.ownKeys`

3. `` `${Symbol("x")}` `` …

A. Gives `"Symbol(x)"`

B. Throws a `TypeError` — symbols never convert to strings implicitly; use `String(sym)` or `.description`

C. Gives `"x"`

D. Gives `""`

4. Which well-known symbol does `for…of` look up?

A. `Symbol.toPrimitive`

B. `Symbol.iterator`

C. `Symbol.species`

D. `Symbol.hasInstance`

5. `Symbol.for` is the right choice when…

A. You need uniqueness above all

B. Two independently loaded pieces of code must agree on the same key

C. The symbol will be serialised to JSON

D. You want a private field

ANSWER KEY · SYMBOLS — UNIQUE KEYS AND THE LANGUAGE'S EXTENSION POINTS

1. B `Symbol.for("id") === Symbol.for("id")` is `true` — the global registry.

2. B Hidden from the common paths, not private.

3. B A guard against accidentally turning a key into the text `Symbol(x)`.

4. B `for await…of` uses `Symbol.asyncIterator`.

5. B Node's `Symbol.for("nodejs.util.inspect.custom")` is the canonical example.

CHECKPOINT · 24 MIN

Checkpoint — Collections and iteration

This checkpoint covers `Map` and `Set` (operations, SameValueZero, insertion order, the idioms and hand-written set algebra), the weak collections and why they cannot be iterated, the iteration protocol (`Symbol.iterator`, `next()`, `done` / `value`, laziness, `return()`), generators (`yield`, `yield*`, `next(value)`, lazy pipelines), and symbols as keys and as the language's hooks.

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:

- Map versus object; how Map compares keys; how to count, group and index with a Map.
- Why a WeakMap has no `size` and takes only object keys; the three patterns it exists for.
- The exact shape of an iterator; iterable versus iterator; what `break` triggers.
- What runs when a generator is called, what `yield*` does, why the first `next(v)` argument is dropped.
- `Symbol()` versus `Symbol.for()`; which enumeration paths skip symbol keys; three well-known symbols.

The programs are an LRU cache built on a Map's insertion order, a graph explorer with adjacency sets and a breadth-first generator, and a small iterator toolkit — `chunk`, `zip`, `enumerate`, `takeWhile` — driven by commands.

EXERCISE 1

An LRU cache on Map order

Implement `LRUCache(capacity)` over a private `Map` whose insertion order is the recency order (oldest first). `get(key)` returns the value (re-inserting the entry so it becomes newest) or `-1`; `put(key, value)` inserts or refreshes, then evicts the oldest entry when over capacity and returns the evicted key (else `null`); `get size`; `entries()` yields entries newest first. Line 1 is the capacity; then `put k v` (print `evict <k>` when something was evicted), `get k` (print `get <k> → <v|-1>`), `show` (print `[k=v, ...] size=<n>` newest first).

Example: capacity `2`, then `put a 1`, `put b 2`, `get a`, `put c 3`, `get b`, `show`, `put a 10`, `show` →

```
get a → 1
evict b
get b → -1
[c=3, a=1] size=2
[a=10, c=3] size=2
```

INPUT 1

```
2
put a 1
put b 2
get a
put c 3
get b
show
put a 10
show
```

OUTPUT 1

```
get a → 1
evict b
get b → -1
[c=3, a=1] size=2
[a=10, c=3] size=2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
class LRUCache {
  #map = new Map();     // insertion order = recency: oldest first
  constructor(capacity) { this.capacity = capacity; }
  get(key) {
    if (!this.#map.has(key)) return -1;
    const value = this.#map.get(key);
    this.#map.delete(key);
    this.#map.set(key, value);      // re-insert: now the newest
    return value;
  }
  put(key, value) {
    if (this.#map.has(key)) this.#map.delete(key);
    this.#map.set(key, value);
    if (this.#map.size > this.capacity) {
      const oldest = this.#map.keys().next().value;
      this.#map.delete(oldest);
      return oldest;
    }
    return null;
  }
  get size() { return this.#map.size; }
  *entries() { yield* [...this.#map].reverse(); }
}
const cache = new LRUCache(Number(lines[0]));
for (const line of lines.slice(1)) {
  const [cmd, k, v] = line.trim().split(/\s+/);
  if (cmd === "put") {
    const evicted = cache.put(k, Number(v));
    if (evicted !== null) console.log(`evict ${evicted}`);
  } else if (cmd === "get") console.log(`get ${k} → ${cache.get(k)}`);
  else if (cmd === "show") console.log(`[${[...cache.entries()].map(([key, val]) =>
}
```

EXERCISE 2

A graph explorer with a breadth-first generator

Store an undirected graph as a `Map` from node to a `Set` of neighbours. Write a generator `bfs(start)` yielding `[node, distance, parent]` in breadth-first order, visiting neighbours in sorted order and each node once. Commands: `edge a b`; `bfs a` prints `bfs a: <node@distance ...>`; `path a b` reconstructs the shortest path from the parents (stop pulling from the generator as soon as `b` appears) and prints `path a → b: <nodes> (<n> hops)` or `path a → b: none`; `degree a` prints `degree a = <neighbour count>` (0 for an unknown node).

Example: edges `a b`, `a c`, `b d`, `c d`, `d e`, `f g`, then `bfs a`, `path a e`, `path a g`, `degree d` →

```
bfs a: a@0 b@1 c@1 d@2 e@3
path a → e: a b d e (3 hops)
path a → g: none
degree d = 3
```

INPUT 1

```
edge a b
edge a c
edge b d
edge c d
edge d e
edge f g
bfs a
path a e
path a g
degree d
degree z
```

OUTPUT 1

```
bfs a: a@0 b@1 c@1 d@2 e@3
path a → e: a b d e (3 hops)
path a → g: none
degree d = 3
degree z = 0
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const adjacency = new Map();   // node -> Set of neighbours
function neighbours(n) {
  if (!adjacency.has(n)) adjacency.set(n, new Set());
  return adjacency.get(n);
}
function link(a, b) { neighbours(a).add(b); neighbours(b).add(a); }
function* bfs(start) {
  const seen = new Set([start]);
  const queue = [[start, 0, null]];
  while (queue.length) {
    const [node, dist, parent] = queue.shift();
    yield [node, dist, parent];
    for (const next of [...neighbours(node)].sort()) {
      if (!seen.has(next)) { seen.add(next); queue.push([next, dist + 1, node]); }
    }
  }
}
function path(a, b) {
  const parent = new Map();
  for (const [node, , from] of bfs(a)) {
    parent.set(node, from);
    if (node === b) break;                    // stop pulling: nothing past the tar
  }
  if (!parent.has(b)) return null;
  const route = [];
  for (let n = b; n !== null; n = parent.get(n)) route.unshift(n);
  return route;
}
for (const line of lines) {
  const [cmd, a, b] = line.trim().split(/\s+/);
  if (cmd === "edge") link(a, b);
  else if (cmd === "bfs") console.log(`bfs ${a}: ${[...bfs(a)].map(([n, d]) => `${n}
  else if (cmd === "path") {
    const p = path(a, b);
    console.log(p ? `path ${a} -> ${b}: ${p.join(" ")} (${p.length - 1} hops)` : `pa
  } else if (cmd === "degree") console.log(`degree ${a} = ${neighbours(a).size}`);
}
```

EXERCISE 3

An iterator toolkit

Write four generators that work on **any** iterable: `chunk(it, size)` (arrays of `size`, the last possibly shorter), `zip(...its)` (arrays of the i-th items, stopping at the shortest), `enumerate(it, start =`

`0)` (`[index, item]` pairs) and `takeWhile(pred, it)`. Input: line 1 numbers `a`, line 2 words `b`, line 3 a chunk size, line 4 a limit. Print `chunks=<chunk(a, size)>`, `zip=<zip(a, b)>`, `enumerate=<enumerate(b, 1)>`, `takeWhile=<takeWhile(x ⇒ x < limit, a)>` (all as JSON), then `zipString=<zip("abc", a) as c+n pairs joined by ,>`.

Example: `1 2 3 4 5 6 7` / `red green blue` / `3` / `5` →

```
chunks=[[1,2,3],[4,5,6],[7]]
zip=[[1,"red"],[2,"green"],[3,"blue"]]
enumerate=[[1,"red"],[2,"green"],[3,"blue"]]
takeWhile=[1,2,3,4]
zipString=a1,b2,c3
```

INPUT 1

```
1 2 3 4 5 6 7
red green blue
3
5
```

OUTPUT 1

```
chunks=[[1,2,3],[4,5,6],[7]]
zip=[[1,"red"],[2,"green"],[3,"blue"]]
enumerate=[[1,"red"],[2,"green"],
[3,"blue"]]
takeWhile=[1,2,3,4]
zipString=a1,b2,c3
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
function* chunk(it, size) {
  let buffer = [];
  for (const x of it) {
    buffer.push(x);
    if (buffer.length === size) { yield buffer; buffer = []; }
  }
  if (buffer.length) yield buffer;
}
function* zip(...its) {
  const iterators = its.map((it) => it[Symbol.iterator]());
  while (true) {
    const results = iterators.map((i) => i.next());
    if (results.some((r) => r.done)) return;
    yield results.map((r) => r.value);
  }
}
function* enumerate(it, start = 0) { let i = start; for (const x of it) yield [i++,
function* takeWhile(pred, it) { for (const x of it) { if (!pred(x)) return; yield x;
const a = lines[0].trim().split(/\s+/).map(Number);
const b = lines[1].trim().split(/\s+/);
const size = Number(lines[2]);
const limit = Number(lines[3]);
console.log(`chunks=${JSON.stringify([...chunk(a, size)])}`);
console.log(`zip=${JSON.stringify([...zip(a, b)])}`);
console.log(`enumerate=${JSON.stringify([...enumerate(b, 1)])}`);
console.log(`takeWhile=${JSON.stringify([...takeWhile((x) => x < limit, a)])}`);
console.log(`zipString=${[...zip("abc", a)].map(([c, n]) => c + n).join(",")}`);
```

CHECKPOINT QUESTIONS

1. `JSON.stringify(new Map([["a", 1]]))` is…

A. `{"a":1}`

B. `{}` — Maps have no enumerable own properties; convert with `Object.fromEntries` first

C. `[["a",1]]`

D. An error

2. Deleting the current entry while iterating a `Map` with `for…of`…

A. Throws

B. Is safe — Map iterators handle deletions; the array equivalent skips elements

C. Restarts the loop

D. Deletes the next entry too

3. `const s = new Set(); items.filter((x) ⇒ !s.has(x.id) && s.add(x.id))` deduplicates by id because…

A. `filter` removes duplicates itself

B. `Set#add` returns the Set (truthy), so an unseen id passes and is recorded in one expression

C. `has` adds the id

D. It sorts the items

4. Which can be a `WeakSet` member?

A. A string

B. A number

C. An object

D. A symbol created with `Symbol.for`

5. An iterator's `next()` must return…

A. The next value

B. An object `{ value, done }`

C. `null` at the end

D. An array

6. `const [a, b] = gen()` where `gen` yields five values…

A. Consumes all five

B. Pulls two values and then calls the generator's `return()`, running its `finally` blocks

C. Throws — generators cannot be destructured

D. Binds arrays

7. Inside a generator, `const x = yield 5;` — `x` is…

A. `5`

B. The argument passed to the **next** `next(v)` call

C. Always `undefined`

D. The return value

8. `gen.return(v)` on a paused generator…

A. Is ignored

B. Finishes it (running `finally` blocks) and yields `{ value: v, done: true }`

C. Restarts it

D. Throws inside it

9. Why are `"abc".length` and `[..."abc"].length` sometimes different for other strings?

A. Spread drops spaces

B. `length` counts UTF-16 code units while iteration is by code point — an emoji is two units, one item

C. Spread includes the terminator

D. They are always equal

10. `4 instanceof Even` can be `true` because…

A. Numbers are objects

B. `Even` defines `static [Symbol.hasInstance](n)`, which `instanceof` calls instead of walking the chain

C. `Even.prototype` is `Number.prototype`

D. It cannot

11. `Reflect.ownKeys(obj)` returns…

A. Enumerable string keys

B. Every own key — string keys (including non-enumerable) then symbol keys

C. Only symbols

D. Inherited keys too

12. The best structure for "have I processed this object before" over objects you do not own, in a long-lived process, is…

A. An array with `includes`

B. A `WeakSet` — identity membership without keeping the objects alive

C. A `Map` of object to `true`

D. A property on each object

ANSWER KEY · CHECKPOINT — COLLECTIONS AND ITERATION

1. B Same for `Set` — spread it into an array first.

2. B One more reason Maps suit churn.

3. B A seen id fails `!s.has`, so `add` never runs again for it.

4. C Weak collections hold objects only on Node 16.

5. B `done: true` signals the end; that result's `value` is the return value.

6. B Destructuring a prefix ends iteration early like `break` does.

7. B `yield` hands 5 out and later receives what the caller sends in.

8. B `gen.throw(err)` raises at the paused `yield` instead.

9. B Module 10 covers Unicode in depth.

10. B A reminder that `instanceof` can be customised — and can lie.

11. B `Object.keys` is enumerable strings only; `getOwnPropertySymbols` is symbols only.

12. B A `Set` / `Map` would pin every object; a property is intrusive and fails on frozen objects.
