Module 4 of 16

Objects, arrays and destructuring

Object literals and JSON; references, shallow and deep copies, Object.freeze and immutable updates; arrays and the sort trap; the map/filter/reduce toolbox; destructuring and spread in every position.

Objects and arrays are the two shapes every JavaScript program is made of, and both hide a handful of rules that separate code that works from code that seems to work. Objects: keys are strings, iteration order is integers-then-insertion, `in` sees the prototype, JSON drops `undefined` and functions and turns Dates into strings. Arrays: `new Array(3)` is three holes, `sort` compares as strings, `shift` is O(n), and half the methods mutate while the other half copy.

The module opens with the literal and its shorthands, moves to the one idea behind most "supernatural" bugs — copying is a decision (alias, shallow, deep) — and the immutable-update idiom that modern code prefers instead. Two lessons cover arrays: the mutating core and `sort` first, then the method toolbox with the `reduce` patterns (sum, frequency map, group by, index by key) that replace most loops. Destructuring and spread close the module: object and array patterns, defaults, rest, named parameters, and the config-merge idiom.

The exercises make you produce the exact outputs the rules predict — what `JSON.stringify` kept, which copy saw the mutation, what the default sort did to your numbers — and finish with three programs a working developer writes weekly: a grouped report, an immutable path-update engine, and a records processor that destructures in every parameter list.

CONTENTS

1	Object literals — properties, keys, iteration and JSON	13 min
2	References, shallow and deep copies, and freezing	13 min
3	Arrays — dynamic, sparse, and sorted lexically by default	13 min
4	The array methods toolbox — map, filter, reduce and friends	14 min
5	Destructuring and spread — taking things apart and putting them together	13 min
✓	Checkpoint — Objects, arrays and destructuring	24 min

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

LESSON 1 · 13 MIN

Object literals — properties, keys, iteration and JSON

An object in JavaScript is a bag of properties: string (or symbol) keys mapped to values, created most often with a literal `{ }`. There is no class needed, no schema, and any property can be added or removed at any time — which makes objects the language's universal record, dictionary and namespace. This lesson covers the literal syntax and its shorthands, the two ways to access a property and when each is required, the `in` / `hasOwn` /optional-chaining tests, the iteration order rules, and JSON as the wire format objects travel in.

The literal and its shorthands

```
const name = "Ada", born = 1815;
const key = "role";
const user = {
  name,                       // property shorthand: name: name
  born,
  "full name": "Ada Lovelace",   // any string as a key, quoted when not an id
  [key]: "mathematician",     // computed key: the value of `key`
  [`is${"Adult"}`]: true,     // computed from any expression
  greet() { return `Hi, ${this.name}`; },   // method shorthand
  get age() { return 2024 - this.born; },   // getter: read as a property, com
  set age(v) { this.born = 2024 - v; },     // setter
  nested: { city: "London" },
  tags: ["math", "engine"],
};
```

Keys are always strings (or symbols) — `{ 1: "a" }` has the key `"1"`, and `obj[1]` and `obj["1"]` are the same property. Trailing commas are fine. Duplicate keys take the last value silently.

Access: dot versus bracket

```
user.name             // dot: the key is a literal identifier
user["full name"]     // bracket: the key has a space, or…
user[key]             // …the key is a variable, or…
user[1 + 1]           // …an expression. Brackets always take a string (after c
user.missing          // undefined — not an error
user.missing.deep     // TypeError: Cannot read properties of undefined
user.missing?.deep    // undefined — optional chaining stops at the first nulli
```

A missing property is `undefined`; reading a property *of* `undefined` throws. That asymmetry is the source of the most common runtime error, and `?.` is its cure.

Adding, changing, deleting

```
user.email = "ada@example.com";   // add
user.born = 1816;                 // change
delete user.tags;                 // remove the property entirely (returns tru
user.tags = undefined;            // the key still exists with value undefined
```

`delete` is slow-ish and rare in application code; setting to `undefined` or building a new object without the key is more common. Objects are mutable by default; `Object.freeze` (next lesson) stops changes.

Testing for a property

```
"name" in user                    // true — own or inherited
Object.hasOwn(user, "name")       // true — own only (2022; the old spell
user.name !== undefined           // false negative if the value IS undef
user.name ?? "anonymous"          // default for missing or nullish
```

Use `Object.hasOwn` for "does this record have this field". `in` also sees prototype properties (`"toString" in {}` is `true`), which is almost never what you mean for data.

Iterating

```
Object.keys(user)      // ["name", "born", "full name", …] own enumerable str
Object.values(user)
Object.entries(user)   // [["name", "Ada"], …] — the one to use with for…of a
for (const [k, v] of Object.entries(user)) console.log(k, v);
for (const k in user) { }   // legacy: includes inherited enumerable props; a
Object.fromEntries(entries) // the inverse — build an object from pairs (hand
```

Order is specified since 2015: integer-like keys first in ascending numeric order, then string keys in insertion order, then symbols. So `{ b: 1, 2: 1, a: 1, 1: 1 }` iterates as `1, 2, b, a`. Rely on insertion order for string keys; never on it for numeric keys — and if you need a real ordered map with arbitrary keys, `Map` (module 6).

Objects as dictionaries — and why `Map` exists

A plain object works as a string-keyed map, with three caveats: keys are coerced to strings (`obj[1]` is `obj["1"]`, `obj[{}]` is `obj["[object Object]"]`); inherited names like `"constructor"` and `"toString"` are already "present" via the prototype (`obj["constructor"]` is a function — use `Object.hasOwn` or `Object.create(null)`); and `__proto__` as a key is a security hazard when keys come from user input (**prototype pollution**). For anything with non-string keys, hostile input, or frequent adds/removes, use `Map`.

JSON

```
const text = JSON.stringify(user);            // '{"name":"Ada",…}' — strings,
JSON.stringify(user, null, 2);                // pretty-printed with 2-space i
JSON.stringify({ a: undefined, f() {}, d: new Date(0), n: NaN });   // '{"d":"
const back = JSON.parse(text);                // a plain object again — metho
JSON.parse("{bad json}");                     // SyntaxError — always try/cat
```

JSON is a *subset* of the literal syntax: keys must be double-quoted strings; no functions, `undefined`, `NaN` / `Infinity` (they become `null`), comments, or trailing commas; `Date`s become ISO strings and do not come back as `Date`s. `stringify` throws on cycles and on `BigInt`. The second argument of both is a **replacer**/**reviver** function for custom handling. `JSON.parse(JSON.stringify(x))` is the old deep-clone trick, with all of those losses.

`Object` static helpers you will use

`Object.assign(target, ...sources)` (copy properties; shallow), `Object.freeze` / `isFrozen`, `Object.keys/values/entries/fromEntries`, `Object.hasOwn`, `Object.create(proto)` (module 5), `Object.getOwnPropertyNames` (includes non-enumerable), `Object.defineProperty` (fine-grained control: writable, enumerable, configurable — rarely in application code).

Interview angle

"Dot versus bracket access?" Dot for identifier-like literal keys; brackets for computed keys, variables, or keys with spaces — brackets convert to string.

"How do you check whether an object has a property?" `Object.hasOwn(obj, key)` for own properties; `in` includes the prototype chain.

"What is the iteration order of object keys?" Integer-like keys ascending, then strings in insertion order, then symbols.

"What does `JSON.stringify` drop?" `undefined`, functions and symbols (omitted); `NaN` / `Infinity` become `null`; `Date`s become strings; `BigInt` and cycles throw.

"Why `Map` instead of an object as a dictionary?" Any key type, no prototype collisions or pollution, size, insertion order, better for churn.

Key takeaways

Literal shorthands: `{ name }`, `[computed]`, `method() {}`, `get` / `set`. Keys are strings (or symbols).

`a.b` for literal keys, `a[k]` for computed; missing → `undefined`, missing-of-missing → throw; `?.` guards.

`Object.hasOwn` for own properties; `Object.entries` + `for…of` to iterate; order: integers, then insertion.

Plain objects as dictionaries have string-only keys and prototype hazards — `Map` for real maps.

JSON: double-quoted keys, no functions/undefined/Dates round-trip; `parse` untrusted input in `try` / `catch`.

EXERCISE 1

Build an object from input, then inspect it

Read `n` followed by `n` key/value pairs and build an object with **bracket access** (the keys are only known at runtime). A value that parses as a finite number is stored as a number, otherwise as a string. Then print four lines: `keys=<Object.keys joined by ,>`, `hasOwn(toString)=<Object.hasOwn(obj,"toString")> in(toString)=<"toString" in obj>`, `json=<JSON.stringify(obj)>`, and `entries=<key:typeof value>` for every entry, space-separated.

Example: `3 name Ada born 1815 lang js` →

```
keys=name,born,lang
hasOwn(toString)=false in(toString)=true
json={"name":"Ada","born":1815,"lang":"js"}
entries=name:string born:number lang:string
```

The `in` line is the point: `toString` is *in* every plain object through its prototype, but it is not an *own* property unless you put it there.

INPUT 1

```
3 name Ada born 1815 lang js
```

OUTPUT 1

```
keys=name,born,lang
hasOwn(toString)=false in(toString)=true
json=
{"name":"Ada","born":1815,"lang":"js"}
entries=name:string born:number
lang:string
```

INPUT 2

```
1 x 5
```

OUTPUT 2

```
keys=x
hasOwn(toString)=false in(toString)=true
json={"x":5}
entries=x:number
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);
const obj = {};
for (let i = 0; i < n; i++) {
  const key = tokens[1 + 2 * i], raw = tokens[2 + 2 * i];
  obj[key] = Number.isFinite(Number(raw)) ? Number(raw) : raw;   // computed key vid
}
console.log(`keys=${Object.keys(obj).join(",")}`);
console.log(`hasOwn(toString)=${Object.hasOwn(obj, "toString")} in(toString)=${"toSt
console.log(`json=${JSON.stringify(obj)}`);
console.log(`entries=${Object.entries(obj).map(([k, v]) => `${k}:${typeof v}`).join(
```

EXERCISE 2

What JSON loses

The starter parses one JSON object from stdin and adds four properties JSON cannot carry faithfully: `u: undefined`, `f` (a function), `d: new Date(0)` and `n: NaN`. Stringify it, parse the text back, and report: `json=<the text>`, `before=<keys of the original>`, `after=<keys of the parsed copy>`, and `dateType=<typeof back.d> nanBecame=<back.n>`.

Example: `{"a":1,"b":"two"}` →

```
json={"a":1,"b":"two","d":"1970-01-01T00:00:00.000Z","n":null}
before=a,b,u,f,d,n
after=a,b,d,n
dateType=string nanBecame=null
```

`undefined` and functions vanish, a Date becomes an ISO string, and `NaN` becomes `null` — none of them come back.

INPUT 1

```
{"a":1,"b":"two"}
```

OUTPUT 1

```
json={"a":1,"b":"two","d":"1970-01-
01T00:00:00.000Z","n":null}
before=a,b,u,f,d,n
after=a,b,d,n
dateType=string nanBecame=null
```

REFERENCE SOLUTION

```
"use strict";
const obj = JSON.parse(require("fs").readFileSync(0, "utf8"));
obj.u = undefined;
obj.f = () => 1;
obj.d = new Date(0);
obj.n = NaN;
const text = JSON.stringify(obj);
console.log(`json=${text}`);
const back = JSON.parse(text);
console.log(`before=${Object.keys(obj).join(",")}`);
console.log(`after=${Object.keys(back).join(",")}`);
console.log(`dateType=${typeof back.d} nanBecame=${back.n}`);
```

QUIZ

1. `const key = "age"; const o = { key: 30 };` — what is `o.age`?

A. `30`

B. `undefined` — the property is literally named `key`; `{ [key]: 30 }` computes it

C. `"age"`

D. A syntax error

2. `"toString" in {}` and `Object.hasOwn({}, "toString")` are…

A. Both `true`

B. `true` and `false` — `in` walks the prototype chain, `hasOwn` checks own properties only

C. Both `false`

D. `false` and `true`

3. `Object.keys({ b: 1, 2: 1, a: 1, 1: 1 })` is…

A. `["b", "2", "a", "1"]` — insertion order

B. `["1", "2", "b", "a"]` — integer-like keys first ascending, then strings in insertion order

C. `["a", "b", "1", "2"]`

D. Unspecified

4. `JSON.parse(JSON.stringify({ a: undefined, b: () ⇒ 1, c: new Date(0) }))` gives…

A. The same object back

B. `{ c: "1970-01-01T00:00:00.000Z" }` — `undefined` and functions are dropped, Dates become strings

C. `{ a: null, b: null, c: null }`

D. A `TypeError`

5. Why prefer a `Map` over a plain object as a dictionary keyed by user input?

A. Objects are slower in every case

B. Keys are coerced to strings, `"constructor"` / `"toString"` already resolve through the prototype, and `__proto__` as a key is a pollution hazard

C. `Map` supports JSON directly

D. Objects cannot be iterated

ANSWER KEY · OBJECT LITERALS — PROPERTIES, KEYS, ITERATION AND JSON

1. B Unquoted names in a literal are literal; brackets evaluate the expression.

2. B For "does this record have this field" use `Object.hasOwn`.

3. B Never rely on insertion order for numeric keys; use a `Map` for an arbitrary-keyed ordered collection.

4. B JSON is data only; anything that is not a string, finite number, boolean, null, array or plain object is lost or converted.

5. B `Object.create(null)` removes the prototype hazards; `Map` removes them and allows any key type.

LESSON 2 · 13 MIN

References, shallow and deep copies, and freezing

Objects are shared by reference, so "copying" an object is a decision, not an operation: copy the reference (an alias), copy one level of properties (shallow), or copy everything reachable (deep). Getting it wrong is the source of bugs that look supernatural — a change in one place appears in another, a default config mutated by its first user, a React state update that did not re-render. This lesson lays out the three levels, the tools for each (`{...obj}`, `Object.assign`, `structuredClone`, the JSON trick), `Object.freeze` and what it does not do, and the immutable-update idioms that modern code prefers.

Level 0: aliasing

```
const a = { n: 1, tags: ["x"] };
const b = a;              // same object, two names
b.n = 2;                  // a.n is 2
```

Assignment, passing as an argument, storing in an array — all copy the **reference**. Function parameters are the classic trap: a function that mutates its argument mutates the caller's object.

Level 1: shallow copy

```
const c = { ...a };                 // spread: own enumerable props, one level
const d = Object.assign({}, a);     // same result, older syntax
c.n = 3;                            // a.n unchanged
c.tags.push("y");                   // a.tags is ["x", "y"] — the array was sh
```

Spread copies the *properties*; any property that is itself an object is still the same nested object. Shallow copies are exactly right when the nested objects are treated as immutable (the common case in well-designed code) and exactly wrong when you then mutate a nested value. Arrays: `[...arr]`, `arr.slice()`, `Array.from(arr)` — all shallow.

Level 2: deep copy

```
const e = structuredClone(a);              // Node 17+ / modern browsers: deep,
const f = JSON.parse(JSON.stringify(a)); // the old trick: deep, but drops und
function deepClone(v) {                    // by hand, for plain data (objects,
  if (Array.isArray(v)) return v.map(deepClone);
  if (v && typeof v === "object") return Object.fromEntries(Object.entries(v)
  return v;
}
```

`structuredClone` is the right answer where it exists (Node 17+; this plan's Node 16 lacks it — write the recursive version or use the JSON trick for plain data). Deep copies cost time proportional to the whole graph; if you find yourself deep-cloning on every update, the design wants immutable updates instead.

`Object.freeze` — shallow, but honest

```
const cfg = Object.freeze({ retries: 3, hosts: ["a"] });
cfg.retries = 5;          // ignored silently in sloppy mode, TypeError in str
cfg.hosts.push("b");      // works! freeze is shallow — the array is a separat
Object.isFrozen(cfg);     // true
```

Freezing forbids adding, removing and changing own properties of *that* object. For a deep freeze, recurse (`Object.freeze` each nested object); libraries do this for constants. `Object.seal` (no add/remove, but changes allowed) and `Object.preventExtensions` (no add) are its lighter siblings, rarely used. `const` freezes nothing — it only stops rebinding the variable.

Immutable updates: the idiom modern code prefers

Instead of mutating an object, produce a new one with the change:

```
const user = { name: "Ada", address: { city: "London", zip: "N1" }, tags: ["a

const renamed = { ...user, name: "Ada L." };
const moved = { ...user, address: { ...user.address, city: "Paris" } };
const tagged = { ...user, tags: [...user.tags, "b"] };
const { tags, ...withoutTags } = user;
const list2 = list.map((u) => (u.id === id ? { ...u, active: true } : u));
const list3 = list.filter((u) => u.id !== id);
```

Each result shares the untouched parts with the original — cheap — and equality of references now means "nothing changed", which is what React, Redux and every memoisation scheme rely on. The rules: spread every level on the path to the change; never call a mutating array method (`push`, `sort`, `splice`) on shared data; treat function arguments as read-only unless the function's name says otherwise (`sortInPlace`).

Mutating versus non-mutating array methods

Mutate the array	Return a new array
`push`, `pop`, `shift`, `unshift`	`concat`, `slice`, `[...a]`
`splice`	`filter`, `map`, `flat`, `flatMap`
`sort`, `reverse`	`toSorted`, `toReversed` (Node 20+); `[...a].sort()` on 16
`fill`, `copyWithin`	`Array.from`, `with` (Node 20+)

`sort` mutating in place is the one that bites: `const sorted = arr.sort()` sorts `arr` *and* aliases it. Copy first: `[...arr].sort(cmp)`.

Equality of objects

`===` on objects is identity. There is no built-in structural equality; write one (compare `Object.entries` recursively), use `JSON.stringify` for plain data with stable key order, or a library's `isEqual`. Two separately created objects with the same content are never `===`, which is the whole reason immutable updates make change detection cheap: a *changed* object is a *different* reference.

Defensive copying at boundaries

A class that stores an array it was given, or returns its internal array, hands out aliases: callers can mutate its state from outside. Copy on the way in (`this.items = [...items]`) and on the way out (`return [...this.items]` or a frozen view). The same applies to default parameter objects — `function f(opts = DEFAULTS)` shares one `DEFAULTS` between all calls; spread it.

Interview angle

"Shallow versus deep copy?" Shallow copies one level (`{...o}`, `Object.assign`); nested objects are shared. Deep copies the whole graph (`structuredClone`, recursion).

"What does `Object.freeze` do?" Prevents adding/removing/changing the object's own properties — shallowly; nested objects stay mutable.

"Does `const` make an object immutable?" No — only the binding.

"How do you update a nested field immutably?" Spread every level on the path: `{ ...u, address: { ...u.address, city } }`.

"Why does `arr.sort()` cause bugs?" It sorts in place and returns the same array; copy first.

Key takeaways

Assignment aliases; spread/`assign`/`slice` copy one level; `structuredClone` or recursion copies deeply.

`Object.freeze` is shallow; `const` is not immutability at all.

Prefer immutable updates — spread the path, `map`/`filter` arrays — so a changed thing is a new reference.

Know which array methods mutate (`push`, `sort`, `splice`, `reverse`) and copy before them on shared data.

Copy at class and function boundaries; never share a default options object.

EXERCISE 1

Alias, shallow, deep — who saw the change?

The starter parses an object with a number `n`, a nested object `inner` (with `x`) and an array `list`, then makes three copies: an **alias**, a **shallow** copy with spread, and `deepClone(original)` — which you must implement recursively for arrays and plain objects (primitives return as-is). Then it mutates the original: `n = 99`, `inner.x = 99`, `list.push(99)`. Print one line per copy — `<label>: n=<> inner.x=<> list=<JSON>` — and finally `shallowSharesInner=<shallow.inner === original.inner> deepSharesInner=<deep.inner === original.inner>`.

Example: `{"n":1,"inner":{"x":1},"list":[1,2]}` →

```
alias: n=99 inner.x=99 list=[1,2,99]
shallow: n=1 inner.x=99 list=[1,2,99]
deep: n=1 inner.x=1 list=[1,2]
shallowSharesInner=true deepSharesInner=false
```

INPUT 1

```
{"n":1,"inner":{"x":1},"list":[1,2]}
```

OUTPUT 1

```
alias: n=99 inner.x=99 list=[1,2,99]
shallow: n=1 inner.x=99 list=[1,2,99]
deep: n=1 inner.x=1 list=[1,2]
shallowSharesInner=true
deepSharesInner=false
```

REFERENCE SOLUTION

```
"use strict";
const original = JSON.parse(require("fs").readFileSync(0, "utf8"));
const alias = original;
const shallow = { ...original };
function deepClone(v) {
  if (Array.isArray(v)) return v.map(deepClone);
  if (v !== null && typeof v === "object") return Object.fromEntries(Object.entries(
  return v;
}
const deep = deepClone(original);
original.n = 99;
original.inner.x = 99;
original.list.push(99);
for (const [label, o] of [["alias", alias], ["shallow", shallow], ["deep", deep]]) {
  console.log(`${label}: n=${o.n} inner.x=${o.inner.x} list=${JSON.stringify(o.list)
}
console.log(`shallowSharesInner=${shallow.inner === original.inner} deepSharesInner=
```

EXERCISE 2

Freeze is shallow

Parse a config `{ retries, hosts }` and freeze it with `Object.freeze`. In strict mode (the starter has `"use strict"`), try `cfg.retries = 99` inside `try` / `catch` and record either `changed` or the error's constructor name; then try `cfg.hosts.push("z")` and record `changed to <new length>` or the error name. Print `top=<> nested=<>`, then `frozen=<Object.isFrozen(cfg)> nestedFrozen=<Object.isFrozen(cfg.hosts)>`, then `retries=<> hosts=<JSON>`.

Example: `{"retries":3,"hosts":["a","b"]}` →

```
top=TypeError nested=changed to 3
frozen=true nestedFrozen=false
retries=3 hosts=["a","b","z"]
```

The frozen object refused the top-level write, but the array inside it is a separate, unfrozen object.

INPUT 1

```
{"retries":3,"hosts":["a","b"]}
```

OUTPUT 1

```
top=TypeError nested=changed to 3
frozen=true nestedFrozen=false
retries=3 hosts=["a","b","z"]
```

REFERENCE SOLUTION

```
"use strict";
const cfg = Object.freeze(JSON.parse(require("fs").readFileSync(0, "utf8")));
let top;
try { cfg.retries = 99; top = "changed"; } catch (e) { top = e.constructor.name; }
let nested;
try { cfg.hosts.push("z"); nested = `changed to ${cfg.hosts.length}`; } catch (e) {
console.log(`top=${top} nested=${nested}`);
console.log(`frozen=${Object.isFrozen(cfg)} nestedFrozen=${Object.isFrozen(cfg.hosts
console.log(`retries=${cfg.retries} hosts=${JSON.stringify(cfg.hosts)}`);
```

QUIZ

1. `const b = { ...a }; b.inner.x = 5;` — is `a.inner.x` affected?

A. No, spread copies everything

B. Yes — spread is shallow; `b.inner` is the same object as `a.inner`

C. Only if `a` is frozen

D. It throws

2. Which deep-copies a plain-data object on Node 16?

A. `Object.assign({}, o)`

B. `JSON.parse(JSON.stringify(o))` or a recursive clone — `structuredClone` arrives in Node 17

C. `{ ...o }`

D. `Object.freeze(o)`

3. `const cfg = Object.freeze({ hosts: ["a"] }); cfg.hosts.push("b");` in strict mode…

A. Throws a `TypeError`

B. Succeeds — `freeze` is shallow; the array is a separate, unfrozen object

C. Silently does nothing

D. Freezes the array too

4. Which of these mutate the array they are called on?

A. `map`, `filter`, `slice`

B. `push`, `sort`, `splice`, `reverse`

C. `concat`, `join`

D. None — arrays are immutable

5. The idiomatic immutable update of `user.address.city`:

A. `user.address.city = c`

B. `{ ...user, address: { ...user.address, city: c } }` — spread each level on the path

C. `Object.freeze(user)`

D. `JSON.parse(JSON.stringify(user))` then assign

ANSWER KEY · REFERENCES, SHALLOW AND DEEP COPIES, AND FREEZING

1. B One level is copied; nested objects are shared references.

2. B The JSON trick loses `undefined`, functions, Dates and cycles; recursion handles what you tell it to.

3. B Freeze recursively for a deep freeze; `cfg.retries = 1` on the frozen object itself would throw.

4. B Copy first (`[...a].sort()`) when the original is shared.

5. B Every object on the changed path is new; untouched siblings are shared — cheap, and a changed thing is a new reference.

LESSON 3 · 13 MIN

Arrays — dynamic, sparse, and sorted lexically by default

A JavaScript array is an object with integer-like keys, a `length` that follows the largest index, and a prototype full of methods. It grows on demand, may have holes, holds anything, and sorts its elements as **strings** unless told otherwise — the last fact alone has produced a million wrong leaderboards. This lesson covers construction, indexing, `length`, the mutating core (`push` / `pop` / `shift` / `unshift` / `splice`), slicing and searching, `sort` with a comparator, and the conversions from and to other shapes.

Making arrays

```
const a = [1, 2, 3];                          // literal — the normal way
const b = new Array(3);                       // [ <3 empty items> ] — length 3,
const c = Array.of(3);                        // [3]
const d = Array.from({ length: 3 }, (_, i) => i * i);   // [0, 1, 4] — length
const e = Array.from("abc");                  // ["a", "b", "c"] — any iterable
const f = [...someSet];                       // spread an iterable
const g = Array(5).fill(0);                   // [0, 0, 0, 0, 0]
```

`new Array(n)` with one numeric argument is the trap: it creates holes, not zeros, and `map` skips holes. `Array.from({ length: n }, fn)` or `Array(n).fill(x)` are the idioms for "n of something". `Array(n).fill([])` fills with the **same** array `n` times — use `Array.from({ length: n }, () => [])` for independent ones.

Indexing and `length`

```
a[0]; a[a.length - 1]; a.at(-1);          // at() accepts negatives (2022)
a[10] = 1;                                 // extends length to 11, leaving h
a.length = 2;                              // truncates: [1, 2]
a[-1]                                      // undefined — a property named "-
```

`length` is the highest index + 1, not a count of defined elements; assigning to it truncates or extends. Out-of-range reads give `undefined`, never an error. Holes (`[1, , 3]`, `new`

`Array(n)`) are rare in practice and skipped by `forEach` / `map` / `filter` but visited by `for…of` (as `undefined`) — avoid creating them.

The mutating core

```
a.push(4, 5);       // append, returns new length      — O(1)
a.pop();            // remove last, returns it         — O(1)
a.unshift(0);       // prepend, returns new length     — O(n): every element s
a.shift();          // remove first, returns it        — O(n)
a.splice(1, 2);             // remove 2 elements at index 1, returns them
a.splice(1, 0, "x", "y");   // insert at 1 without removing
a.splice(1, 1, "z");        // replace one
```

Stack = `push` / `pop`; queue = `push` / `shift` (fine for small queues; for large ones the O(n) `shift` matters — use an index or a deque). `splice` is the Swiss-army knife for in-place edits; its return value is the removed elements.

Non-mutating slices and joins

```
a.slice(1, 3);        // copy of [1, 3) — end exclusive; negatives from the er
a.concat([9], 10);    // new array; spread [...a, 9, 10] is the modern spellir
a.join("-");          // "1-2-3"; join() defaults to ","; String(a) is the sar
a.indexOf(2); a.lastIndexOf(2); a.includes(2);   // includes finds NaN; indexO
a.flat(); a.flat(Infinity);                       // flatten one level / compl
```

`sort` — the trap and the fix

```
[10, 9, 1, 100].sort();                  // [1, 10, 100, 9] — converted to st
[10, 9, 1, 100].sort((x, y) ⇒ x - y);    // [1, 9, 10, 100] — numeric ascend
words.sort((x, y) ⇒ x.localeCompare(y)); // human order for strings
people.sort((p, q) ⇒ p.age - q.age || p.name.localeCompare(q.name));   // by
```

The comparator returns negative/zero/positive; `x - y` is fine for safe integers, and `(x > y) - (x < y)` or `Math.sign` when overflow or non-numbers are possible. `sort` is **stable** (since ES2019) and **in place** — copy first (`[...a].sort(cmp)`) when the original must survive.

`reverse` is also in place. Node 20+ has `toSorted` / `toReversed` that return copies; Node 16 does not.

Searching with predicates

```
a.find((x) ⇒ x > 1);         // first match or undefined
a.findIndex((x) ⇒ x > 1);    // or -1
a.some((x) ⇒ x > 1); a.every((x) ⇒ x > 0);
a.findLast(...)              // Node 18+ — not available here; use a reverse
```

Multidimensional arrays

```
const grid = Array.from({ length: rows }, () ⇒ Array(cols).fill(0));   // in
grid[r][c] = 1;
const transposed = grid[0].map((_, c) ⇒ grid.map((row) ⇒ row[c]));
```

There is no 2-D array type — an array of arrays — and the "same inner array n times" mistake from `Array(n).fill(Array(m))` is the one to remember.

Conversions

`Array.from(iterableOrArrayLike)`, `[...iterable]`, `Object.entries(obj)`, `Object.fromEntries(pairs)`, `Array.isArray(x)`, `arr.join`, `str.split`, `new Set(arr)` (dedupe) then `[...set]`, `Array.from(map.values())`. Array-like objects (`arguments`, DOM `NodeList`) are not arrays until `Array.from` makes them one.

Performance notes

Engines store dense arrays of small integers or of doubles very efficiently; mixing types, creating holes, or using `delete arr[i]` (which makes a hole — use `splice`) degrades them to dictionary mode. Typed arrays (`Int32Array`, `Float64Array`) give fixed-type numeric storage for heavy numeric work. For interview scale none of this matters; the habits — dense, homogeneous, no `delete` — are free.

Interview angle

"Why does `[10, 9, 1].sort()` *give* `[1, 10, 9]`*?"* Default sort compares as strings; pass a numeric comparator.

"`push` / `pop` *versus* `shift` / `unshift` *cost?"* O(1) at the end; O(n) at the front because elements shift.

"`slice` *versus* `splice`*?"* `slice` copies a range and does not mutate; `splice` removes/inserts in place and returns the removed items.

"How do you make an n×m grid of zeros?" `Array.from({ length: n }, () ⇒ Array(m).fill(0))` — not `fill(Array(m))`, which shares one row.

"How do you dedupe an array?" `[...new Set(arr)]`.

Key takeaways

Literals to create; `Array.from({ length }, fn)` or `Array(n).fill(x)` for n items; never `new Array(n)` alone.

`length` tracks the highest index; `at(-1)` for the end; out-of-range reads are `undefined`.

`push` / `pop` O(1), `shift` / `unshift` O(n), `splice` for in-place edits; `slice` /spread/ `concat` copy.

`sort` compares as strings by default and mutates — `(x, y) ⇒ x - y`, copy first when needed.

Grids are arrays of independent arrays; dedupe with `Set`; `includes` finds `NaN`, `indexOf` does not.

EXERCISE 1

Sort it right

Read `n` then `n` integers. Print `default=<JSON>` for `[...nums].sort()` (the string sort), `asc=<JSON>` for a numeric ascending sort, `desc=<JSON>` for numeric descending, and `original=<JSON>` to prove the input array was never mutated — every sort must run on a copy.

Example: `5 10 9 1 100 25` →

```
default=[1,10,100,25,9]
asc=[1,9,10,25,100]
desc=[100,25,10,9,1]
original=[10,9,1,100,25]
```

INPUT 1

```
5 10 9 1 100 25
```

OUTPUT 1

```
default=[1,10,100,25,9]
asc=[1,9,10,25,100]
desc=[100,25,10,9,1]
original=[10,9,1,100,25]
```

INPUT 2

```
4 -3 -20 5 7
```

OUTPUT 2

```
default=[-20,-3,5,7]
asc=[-20,-3,5,7]
desc=[7,5,-3,-20]
original=[-3,-20,5,7]
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const nums = tokens.slice(1, Number(tokens[0]) + 1).map(Number);
const byDefault = [...nums].sort();
const asc = [...nums].sort((a, b) => a - b);
const desc = [...nums].sort((a, b) => b - a);
console.log(`default=${JSON.stringify(byDefault)}`);
console.log(`asc=${JSON.stringify(asc)}`);
console.log(`desc=${JSON.stringify(desc)}`);
console.log(`original=${JSON.stringify(nums)}`);
```

EXERCISE 2

Splice surgery

Start with an empty array and run `n` commands: `push x`, `pop` (print `popped=<value>`), `shift` (print `shifted=<value>`), `unshift x`, `splice i del k a1 … ak` (remove `del` items at index `i`, insert the `k` items there; print `removed=<JSON of what splice returned>`), and `show` (print `<JSON> length=<length>`).

Example: `8 push 1 push 2 push 3 show splice 1 1 2 8 9 show pop show` →

```
[1,2,3] length=3
removed=[2]
[1,8,9,3] length=4
popped=3
[1,8,9] length=3
```

INPUT 1

```
8 push 1 push 2 push 3 show splice 1 1 2
8 9 show pop show
```

OUTPUT 1

```
[1,2,3] length=3
removed=[2]
[1,8,9,3] length=4
popped=3
[1,8,9] length=3
```

INPUT 2

```
6 unshift 5 push 7 shift show splice 0 0
1 4 show
```

OUTPUT 2

```
shifted=5
[7] length=1
removed=[]
[4,7] length=2
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
let pos = 0;
const next = () => tokens[pos++];
const n = Number(next());
const arr = [];
for (let i = 0; i < n; i++) {
  const cmd = next();
  if (cmd === "push") arr.push(Number(next()));
  else if (cmd === "pop") console.log(`popped=${arr.pop()}`);
  else if (cmd === "shift") console.log(`shifted=${arr.shift()}`);
  else if (cmd === "unshift") arr.unshift(Number(next()));
  else if (cmd === "splice") {
    const at = Number(next()), del = Number(next()), k = Number(next());
    const items = [];
    for (let j = 0; j < k; j++) items.push(Number(next()));
    console.log(`removed=${JSON.stringify(arr.splice(at, del, ...items))}`);
  } else console.log(`${JSON.stringify(arr)} length=${arr.length}`);
}
```

QUIZ

1. `[10, 9, 1, 100].sort()` returns…

A. `[1, 9, 10, 100]`

B. `[1, 10, 100, 9]` — elements are compared as strings by default

C. `[100, 10, 9, 1]`

D. A `TypeError`

2. `new Array(3)` is…

A. `[0, 0, 0]`

B. Three holes — `length` 3 and no elements; `map` skips them

C. `[3]`

D. `[undefined, undefined, undefined]` with `map` visiting each

3. `Array(3).fill([])` gives three…

A. Independent empty arrays

B. References to the **same** array — pushing into one shows in all

C. `undefined` values

D. Holes

4. Cost of `shift()` on an array of n elements:

A. O(1) like `pop`

B. O(n) — every remaining element moves down one index

C. O(log n)

D. O(n²)

5. `slice` versus `splice`:

A. Synonyms

B. `slice(start, end)` copies a range without mutating; `splice(start, del, ...items)` removes/inserts in place and returns the removed items

C. `splice` copies, `slice` mutates

D. Both mutate

ANSWER KEY · ARRAYS — DYNAMIC, SPARSE, AND SORTED LEXICALLY BY DEFAULT

1. B Pass `(a, b) ⇒ a - b` for numbers; `sort` also mutates in place.

2. B `Array(3).fill(0)` or `Array.from({ length: 3 }, fn)` are the "n of something" idioms.

3. B `Array.from({ length: 3 }, () ⇒ [])` creates independent rows; the same trap applies to grids.

4. B A large queue built on `shift` is quadratic; use an index or a deque.

5. B One letter, opposite behaviour — a classic interview gotcha.

LESSON 4 · 14 MIN

The array methods toolbox — map, filter, reduce and friends

Most array code in modern JavaScript is a chain of methods: `filter` to keep, `map` to transform, `reduce` to summarise, `find` to locate, `some` / `every` to test, `flatMap` to expand. Each takes a callback, runs the loop for you, and returns something new — which makes pipelines readable and, because they never mutate the source, safe. This lesson is the toolbox with the shape of each method, the `reduce` patterns that replace half the loops you would otherwise write, when to chain and when a plain loop is better, and the mistakes that recur.

The callback signature

Every iteration method calls its callback with `(element, index, array)`; `reduce` prepends the accumulator. Use only what you need — `arr.map((x) ⇒ x * 2)` — and never pass a function that has optional extra parameters (`.map(parseInt)` reads the index as a radix). A `thisArg` can be passed as a second argument; arrows make it obsolete.

Transform and keep

```
nums.map((x) ⇒ x * 2)                          // same length, each element tra
nums.filter((x) ⇒ x % 2 === 0)                 // those passing the test
nums.filter(Boolean)                           // drop falsy values
users.map((u) ⇒ u.email)                       // pluck a field
users.filter((u) ⇒ u.active).map((u) ⇒ u.name)   // chain: filter first, ma
nested.flat(); nested.flat(2);                 // [[1,[2]],[3]] → flat() [1,[2],
words.flatMap((w) ⇒ w.split(""))               // map then flatten one level — 
[1, 2, 3].flatMap((x) ⇒ (x % 2 ? [x, x] : []))   // duplicate odds, drop ever
```

`flatMap` returning `[]` for "drop" and `[a, b]` for "expand" is the idiom for a filter-and-map with variable output count.

Search and test

```
users.find((u) ⇒ u.id === id)                // the element or undefined
users.findIndex((u) ⇒ u.id === id)           // index or -1
users.some((u) ⇒ u.admin)                    // any?
users.every((u) ⇒ u.verified)                // all? (true for an empty array
users.includes(x)                            // === membership (finds NaN)
```

`find` returns the element itself, so `find(...)?.name` is the safe follow-up. `some` and `every` short-circuit.

`reduce`: one fold, many patterns

```
nums.reduce((sum, x) ⇒ sum + x, 0)                          // sum; ALWAYS pa
nums.reduce((best, x) ⇒ (x > best ? x : best), -Infinity)   // max (Math.max(
words.reduce((counts, w) ⇒ { counts[w] = (counts[w] ?? 0) + 1; return counts;
users.reduce((byId, u) ⇒ { byId[u.id] = u; return byId; }, {})
users.reduce((groups, u) ⇒ { (groups[u.city] ??= []).push(u); return groups;
pairs.reduce((m, [k, v]) ⇒ m.set(k, v), new Map())
fns.reduce((acc, f) ⇒ f(acc), x)
```

Without an initial value `reduce` uses the first element as the seed and **throws on an empty array** — pass the seed. The accumulator may be mutated when you created it (`{}` / `[]` seeds), which is faster than spreading a new object every iteration; the `{ ...acc, [k]: v }` style is O(n²) on large inputs. When a `reduce` grows past four or five lines, a `for…of` loop with the same variables is clearer — `reduce` is for folds that read as one idea.

Iterate for effect

```
items.forEach((item, i) ⇒ console.log(i, item));   // no return value; cannot
for (const item of items) { if (done(item)) break; }  // use a loop when you n
for (const [i, item] of items.entries()) { }          // index + element
```

`forEach` is for side effects; `map` is for producing a new array — a `map` whose result is ignored, or a `forEach` that pushes into an outside array, is the wrong method. Neither `forEach` nor `map` waits for `await` inside the callback (module 8); use `for…of`.

Sorting and ordering in pipelines

```
[...users].sort((a, b) => a.age - b.age)            // copy then sort — sort r
users.slice().sort(...)                              // same
scores.reduce((a, b) => Math.max(a, b), -Infinity)   // max without spreading
```

Chains that end in `.sort` without a copy sort the previous step's fresh array — fine — but `original.sort` is not.

Chaining, and its cost

```
const total = orders
  .filter((o) => o.paid)
  .flatMap((o) => o.lines)
  .map((l) => l.qty * l.price)
  .reduce((a, b) => a + b, 0);
```

Each step allocates an intermediate array; on ten thousand items it is negligible, on ten million it is not, and a single loop is faster. For readability chains win almost always; for the hot path, measure. Iterator helpers (`.values().map(...).filter(...)`, lazy, Node 22+) will remove the trade-off; on Node 16 they do not exist.

Removing duplicates, uniqueness by key

```
[...new Set(nums)]                                                  // primi
const seen = new Set(); users.filter((u) => !seen.has(u.id) && seen.add(u.id))
Object.values(users.reduce((m, u) => ({ ...m, [u.id]: u }), {}))    // last w
```

Common mistakes

- `.map(parseInt)` → `.map(Number)` or `.map((s) => parseInt(s, 10))`.
- `reduce` without a seed on a possibly-empty array.
- Mutating inside `map` / `filter` callbacks (pushing to an outer array, changing the element).
- `forEach` with `async` callbacks expecting sequential `await`.

`arr.sort()` for numbers; `original.sort()` when a copy was wanted.

`filter` then `[0]` instead of `find`; `filter(...).length > 0` instead of `some`.

Comparing objects with `includes` / `indexOf` (identity) when you meant "by id".

Interview angle

"*`map` versus `forEach`?*" `map` returns a new array of results; `forEach` returns nothing and is for side effects.

"Implement `groupBy` with `reduce`." Seed `{}`, `(groups[key(x)] ??= []).push(x)`, return `groups`.

"Why pass an initial value to `reduce`?" Empty arrays throw without one, and the accumulator type is explicit.

"*`find` versus `filter`?*" First match (or `undefined`) versus all matches (an array, possibly empty).

"*`flatMap`?*" Map to arrays then flatten one level — filter-and-map in one pass.

Key takeaways

`filter` keeps, `map` transforms, `flatMap` expands or drops, `find` / `some` / `every` search and test.

`reduce` with a seed: sums, max, frequency map, group by, index by key, pipe; mutate a `{}` / `[]` seed for speed.

`forEach` for effects, `for…of` when you need `break` or `await`.

Copy before `sort`; never `.map(parseInt)`; `[...new Set(a)]` to dedupe.

Chains for clarity, a loop for the hot path.

EXERCISE 1

An order report, no loops

Each input line is an order: `customer item qty price paid` where `paid` is `y` or `n`. Using only array methods (no `for`), print `revenue=<sum of qty*price over paid orders, 2 decimals>`; then one line per customer with paid orders, `<customer>: <total>`, sorted by name; then `top=<customer with the largest paid total, or none> unpaid=<count of unpaid orders>`. On ties for top, the customer that appears first in the report order (alphabetical) keeps it.

Example:

```
ada pen 3 1.50 y
bob ink 1 12.00 n
ada pad 2 4.25 y
cy pen 10 1.50 y
```

→

```
revenue=28.00
ada: 13.00
cy: 15.00
top=cy unpaid=1
```

INPUT 1

```
ada pen 3 1.50 y
bob ink 1 12.00 n
ada pad 2 4.25 y
cy pen 10 1.50 y
```

OUTPUT 1

```
revenue=28.00
ada: 13.00
cy: 15.00
top=cy unpaid=1
```

INPUT 2

```
zed cup 1 2.00 n
zed cup 1 2.00 n
```

OUTPUT 2

```
revenue=0.00
top=none unpaid=2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const orders = lines.map((l) => {
  const [customer, item, qty, price, paid] = l.trim().split(/\s+/);
  return { customer, item, qty: Number(qty), price: Number(price), paid: paid === "y
});
const paid = orders.filter((o) => o.paid);
const revenue = paid.reduce((sum, o) => sum + o.qty * o.price, 0);
const byCustomer = paid.reduce((acc, o) => { acc[o.customer] = (acc[o.customer] ?? 0
console.log(`revenue=${revenue.toFixed(2)}`);
for (const [customer, total] of Object.entries(byCustomer).sort(([a], [b]) => a.loca
  console.log(`${customer}: ${total.toFixed(2)}`);
}
const top = Object.entries(byCustomer).reduce((best, e) => (best === null || e[1] >
console.log(`top=${top ? top[0] : "none"} unpaid=${orders.length - paid.length}`);
```

EXERCISE 2

Six methods, six lines

Read `n` then `n` integers and print, one per line: `evensSquared=<JSON of even values squared>`, `sum=<reduce with seed 0>`, `firstOver10=<find(x > 10) or none>`, `anyNegative=<some>` `allPositive=<every>`, `mirrored=<flatMap each x to [x, -x], dropping zeros>`, and `distinct=<JSON of [...new Set(nums)]>`.

Example: `6 3 8 0 12 5 8` →

```
evensSquared=[64,0,144,64]
sum=36
firstOver10=12
anyNegative=false allPositive=false
mirrored=[3,-3,8,-8,12,-12,5,-5,8,-8]
distinct=[3,8,0,12,5]
```

INPUT 1

```
6 3 8 0 12 5 8
```

OUTPUT 1

```
evensSquared=[64,0,144,64]
sum=36
firstOver10=12
anyNegative=false allPositive=false
mirrored=[3,-3,8,-8,12,-12,5,-5,8,-8]
distinct=[3,8,0,12,5]
```

INPUT 2

```
3 -1 2 20
```

OUTPUT 2

```
evensSquared=[4,400]
sum=21
firstOver10=20
anyNegative=true allPositive=false
mirrored=[-1,1,2,-2,20,-20]
distinct=[-1,2,20]
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const nums = tokens.slice(1, Number(tokens[0]) + 1).map(Number);
console.log(`evensSquared=${JSON.stringify(nums.filter((x) => x % 2 === 0).map((x) =
console.log(`sum=${nums.reduce((a, b) => a + b, 0)}`);
console.log(`firstOver10=${nums.find((x) => x > 10) ?? "none"}`);
console.log(`anyNegative=${nums.some((x) => x < 0)} allPositive=${nums.every((x) =>
console.log(`mirrored=${JSON.stringify(nums.flatMap((x) => (x === 0 ? [] : [x, -x]))
console.log(`distinct=${JSON.stringify([...new Set(nums)])}`);
```

QUIZ

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

A. `[1, 2, 3]`

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

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

D. A `TypeError`

2. `[].reduce((a, b) ⇒ a + b)` (no initial value)…

A. Returns `0`

B. Throws a `TypeError` — with no seed the first element is the seed, and there is none

C. Returns `undefined`

D. Returns `[]`

3. `map` versus `forEach`:

A. Identical

B. `map` returns a new array of the callback's results; `forEach` returns `undefined` and is for side effects

C. `forEach` is faster and returns the array

D. `map` can `break`

4. `[].every((x) ⇒ x > 0)` is…

A. `false`

B. `true` — vacuous truth: no element fails the test

C. `undefined`

D. A `TypeError`

5. The `reduce` shape for **group by**:

A. `arr.reduce((g, x) ⇒ g.push(x), [])`

B. `arr.reduce((groups, x) ⇒ { (groups[key(x)] ??= []).push(x); return groups; }, {})`

C. `arr.map(key).filter(Boolean)`

D. `arr.sort().reduce((a, b) ⇒ a + b)`

ANSWER KEY · THE ARRAY METHODS TOOLBOX — MAP, FILTER, REDUCE AND FRIENDS

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

2. B Always pass the initial value; it also makes the accumulator's type explicit.

3. B A `map` whose result is ignored, or a `forEach` pushing into an outer array, is the wrong method.

4. B `some` on an empty array is `false` for the same reason.

5. B Seed `{}`, create the bucket on first sight, push, return the accumulator.

LESSON 5 · 13 MIN

Destructuring and spread — taking things apart and putting them together

Destructuring pulls values out of objects and arrays into variables by shape; spread does the reverse, pouring an iterable or an object's properties into a literal or a call. Together they replaced a huge amount of `const x = obj.x` and `Array.prototype.slice.call` boilerplate, and they are the syntax behind "named parameters", swapping variables, immutable updates, and half of every React component. This lesson covers both in full: defaults, renaming, nesting, rest, the parameter-list forms, and the places where the syntax has sharp edges.

Object destructuring

```
const user = { name: "Ada", born: 1815, address: { city: "London" } };
const { name, born } = user;                          // name = "Ada", born = 181
const { name: fullName } = user;                      // rename: fullName = "Ada"
const { role = "guest" } = user;                      // default when the proper
const { address: { city } } = user;                   // nested: city = "London"
const { name: n = "anon", ...rest } = user;           // rename + default + rest
let a, b;
({ a, b } = { a: 1, b: 2 });                          // assigning to existing v
```

The pattern mirrors a literal: `{ prop }` binds `prop`; `{ prop: newName }` binds `newName`; `= default` applies when the value is `undefined` (not `null`); `...rest` collects the remaining own enumerable properties into a new object. Destructuring `null` or `undefined` throws (`Cannot destructure property 'name' of undefined`) — default the object itself when it may be missing: `const { name } = user ?? {}`.

Array destructuring

```
const [first, second] = [1, 2, 3];        // by position
const [, , third] = [1, 2, 3];            // skip with empty slots
const [head, ...tail] = [1, 2, 3];        // head = 1, tail = [2, 3]
const [x = 0, y = 0] = [5];               // y defaults to 0
[a, b] = [b, a];                          // swap — no temporary variable
const [k, v] = Object.entries(obj)[0];
for (const [key, value] of map) { }       // Map entries are [k, v] pairs
const [, year, month] = /(\d{4})-(\d{2})/.exec(text);   // regex groups
```

Array destructuring works on any **iterable**, not only arrays — strings, Maps, Sets, generators. Positions past the end give `undefined` (then the default). The trailing-rest element must be last.

Destructuring parameters: named arguments

```
function createUser({ name, role = "member", tags = [] } = {}) { /* … */ }
createUser({ name: "Ada" });
createUser();                                          // the = {} defaul

const area = ({ width, height }) => width * height;
const sumPairs = ([a, b]) => a + b;
[[1, 2], [3, 4]].map(([a, b]) => a + b);               // destructure in
Object.entries(counts).map(([word, n]) => `${word}: ${n}`);
```

An options object with destructured, defaulted parameters is the idiomatic replacement for positional parameters past two or three: callers name what they pass, omit what they do not, and reorder freely. The `= {}` default on the whole parameter is what lets the function be called with no argument.

Spread

```
Math.max(...nums);                          // an iterable → separate arguments
const merged = [...a, ...b];                // concatenate arrays
const copy = [...arr];                      // shallow copy
const chars = [...str];                     // string → code points
const unique = [...new Set(arr)];           // dedupe
const withDefaults = { ...defaults, ...options };   // later properties win —
const updated = { ...user, born: 1816 };            // immutable update
f(...args);                                         // forward arguments
```

Array spread requires an iterable (an object throws: `{...}` is not iterable); object spread takes own enumerable properties of anything (arrays spread into `{0: …, 1: …}`; `null` / `undefined` spread to nothing without error). Order matters in object spread: the rightmost wins, which is exactly what you want for `{ ...defaults, ...overrides }`. Both are **shallow** (previous lesson).

Rest versus spread, once more

Same three dots, opposite direction. **Rest** appears where names are bound — a parameter list (`(...args)`), a destructuring pattern (`[a, ...others]`, `{ x, ...rest }`) — and *collects*. **Spread** appears where values are supplied — a call (`f(...xs)`), an array or object literal — and *expands*. In a destructuring pattern the rest must be last; in an object literal spread can appear anywhere, several times.

Sharp edges

- `const { a } = null` throws; `const { a } = maybe ?? {}` does not.
- Starting a statement with `{` destructuring assignment needs parentheses: `({ a } = obj);`.
- Defaults apply on `undefined` only — a `null` property stays `null`.
- Deep patterns bind only the leaves: `const { address: { city } } = user` gives `city`, not `address`.
- Array patterns on non-iterables throw; object patterns on primitives work through boxing (`const { length } = "abc"`).
- Rest in object patterns copies own enumerable properties only — no getters' *values* are recomputed later, no prototype properties.

Spreading a huge array into `Math.max(...arr)` can overflow the argument limit (~100k); reduce instead.

Idioms worth memorising

```
const [min, max] = [Math.min(...xs), Math.max(...xs)];
const { data: { items = [] } = {} } = response;          // deep with defaults
const pick = ({ id, name }) ⇒ ({ id, name });             // project an object
const omit = ({ password, ...safe }) ⇒ safe;              // drop a field
const [{ name: firstName } = {}] = users;                 // first element, de
```

Interview angle

"What does destructuring do?" Binds variables from an object's properties or an iterable's elements by shape, with renaming, defaults and rest.

"How do you swap two variables?" `[a, b] = [b, a]`.

"How do you implement named/optional parameters?" A destructured options object with defaults, defaulted to `{}`.

"`{ ...a, ...b }` — who wins on duplicate keys?" The later spread, `b`.

"Is spread a deep copy?" No — one level; nested objects are shared.

Key takeaways

Object patterns match by name (`{ a, b: renamed = default, ...rest }`); array patterns by position on any iterable (`[x, , y, ...more]`).

Defaults fire on `undefined`; destructuring `null` / `undefined` throws — default the container.

Destructured parameters give named, optional arguments; `= {}` makes the object itself optional.

Spread expands iterables into calls/arrays and properties into objects; later wins; always shallow.

Rest collects (last in a pattern), spread expands — same dots, opposite jobs.

EXERCISE 1

Named arguments

Implement `createUser({ name = "anon", role = "member", tags = [] } = {})` returning `<name> (<role>) tags=<tags.length>`. Each input line is a JSON options object, or `-` meaning "call with no argument at all" — the `= {}` default is what makes that legal. Print one result per line.

Example:

```
{"name":"Ada","role":"admin","tags":["x","y"]}
{"name":"Bob"}
-
{"tags":["a"]}
```

→

```
Ada (admin) tags=2
Bob (member) tags=0
anon (member) tags=0
anon (member) tags=1
```

INPUT 1

```
{"name":"Ada","role":"admin","tags":
["x","y"]}
{"name":"Bob"}
-
{"tags":["a"]}
```

OUTPUT 1

```
Ada (admin) tags=2
Bob (member) tags=0
anon (member) tags=0
anon (member) tags=1
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l !==
function createUser({ name = "anon", role = "member", tags = [] } = {}) {
  return `${name} (${role}) tags=${tags.length}`;
}
for (const line of lines) {
  const opts = line.trim() === "-" ? undefined : JSON.parse(line);
  console.log(createUser(opts));
}
```

EXERCISE 2

Swap, split, rest

Read `n` then `n` integers into `arr`. With **destructuring only** — no index access — bind `[first = 0, second = 0, ...rest]` and print `first=<> second=<> rest=<JSON>`; swap the two with `[first, second] = [second, first]` and print `swapped: first=<> second=<>`; bind `{ length }` from the array and print `length=<> min=<> max=<>` (both 0 for an empty array, otherwise via `Math.min(...arr)` / `Math.max(...arr)`); finally bind the third element with a skip pattern and default `"none"` and print `third=<>`.

Example: `5 4 9 1 7 3` →

```
first=4 second=9 rest=[1,7,3]
swapped: first=9 second=4
length=5 min=1 max=9
third=1
```

INPUT 1

```
5 4 9 1 7 3
```

OUTPUT 1

```
first=4 second=9 rest=[1,7,3]
swapped: first=9 second=4
length=5 min=1 max=9
third=1
```

INPUT 2

```
1 8
```

OUTPUT 2

```
first=8 second=0 rest=[]
swapped: first=0 second=8
length=1 min=8 max=8
third=none
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const arr = tokens.slice(1, Number(tokens[0]) + 1).map(Number);
let [first = 0, second = 0, ...rest] = arr;
console.log(`first=${first} second=${second} rest=${JSON.stringify(rest)}`);
[first, second] = [second, first];
console.log(`swapped: first=${first} second=${second}`);
const { length } = arr;
const [min, max] = length ? [Math.min(...arr), Math.max(...arr)] : [0, 0];
console.log(`length=${length} min=${min} max=${max}`);
const [, , third = "none"] = arr;
console.log(`third=${third}`);
```

QUIZ

1. `const { a = 1 } = { a: null };` — `a` is…

A. `1`

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

C. `undefined`

D. A `TypeError`

2. `const { name } = undefined;`

A. `name` is `undefined`

B. Throws — you cannot destructure `undefined`; write `= maybe ?? {}`

C. `name` is `null`

D. Binds nothing silently

3. `{ ...defaults, ...options }` — on a key present in both, whose value wins?

A. `defaults`

B. `options` — the later spread overwrites

C. Neither; it throws

D. The one with the truthy value

4. `const { address: { city } } = user;` binds…

A. `address` and `city`

B. Only `city` — nested patterns bind the leaves

C. Only `address`

D. An object `{ city }`

5. Same three dots: how do you tell rest from spread?

A. Rest is for arrays, spread for objects

B. Rest sits where names are bound (parameters, patterns) and collects; spread sits where values are supplied (calls, literals) and expands

C. Spread must be last

D. They are the same operation

ANSWER KEY · DESTRUCTURING AND SPREAD — TAKING THINGS APART AND PUTTING THEM TOGETHER

1. B `{ a: undefined }` or a missing property would give 1.

2. B Default the container when it may be missing.

3. B That is exactly the config-merge idiom: defaults first, overrides last.

4. B Add `address` explicitly if you need it too: `{ address, address: { city } }`.

5. B Rest must be last in a pattern; spread may appear anywhere, several times.

CHECKPOINT · 24 MIN

Checkpoint — Objects, arrays and destructuring

This checkpoint covers object literals and their shorthands, property access and testing, key iteration order and JSON, references versus shallow and deep copies with `Object.freeze` and immutable updates, arrays — construction, `length`, the mutating core, `sort` with a comparator — the method toolbox with `reduce` patterns, and destructuring and spread in all their forms.

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:

- When bracket access is required, and what `Object.hasOwn` adds over `in`.
- What `JSON.stringify` drops and what `JSON.parse` cannot bring back.
- Shallow versus deep copy; what `Object.freeze` does not freeze; how to update a nested field immutably.
- Why `new Array(3)` is not `[0, 0, 0]`, and what `[10, 9, 1].sort()` returns.
- `map` versus `forEach`, `find` versus `filter`, and why `reduce` needs a seed.
- Object versus array destructuring, defaults on `undefined`, and which spread wins on duplicate keys.

The programs are a grouping-and-summary report built from `reduce` and `Object.entries`, an immutable-update engine that applies dotted-path changes to a JSON document without mutating the original, and a records processor using destructuring in every parameter list.

EXERCISE 1

Sales summary by region

Each line is `region product qty price` (integers). Group the sales by region with `reduce`, then for every region in **alphabetical order** print `<region>: revenue=<sum of qty*price> items=<sum of qty> top=<product with the largest revenue in that region>` (ties broken by product name ascending). Finish with `regions=<count>`.

Example:

```
north pen 10 2
south pad 3 5
north pad 2 5
north ink 4 3
south pen 1 2
```

→

```
north: revenue=42 items=16 top=pen
south: revenue=17 items=4 top=pad
regions=2
```

INPUT 1

```
north pen 10 2
south pad 3 5
north pad 2 5
north ink 4 3
south pen 1 2
```

OUTPUT 1

```
north: revenue=42 items=16 top=pen
south: revenue=17 items=4 top=pad
regions=2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sales = lines.map((l) => {
  const [region, product, qty, price] = l.trim().split(/\s+/);
  return { region, product, qty: Number(qty), price: Number(price), revenue: Number(
});
const byRegion = sales.reduce((acc, s) => { (acc[s.region] ??= []).push(s); return a
for (const region of Object.keys(byRegion).sort()) {
  const rows = byRegion[region];
  const revenue = rows.reduce((a, s) => a + s.revenue, 0);
  const items = rows.reduce((a, s) => a + s.qty, 0);
  const byProduct = rows.reduce((acc, s) => { acc[s.product] = (acc[s.product] ?? 0)
  const top = Object.entries(byProduct).sort(([pa, ra], [pb, rb]) => rb - ra || pa.l
  console.log(`${region}: revenue=${revenue} items=${items} top=${top}`);
}
console.log(`regions=${Object.keys(byRegion).length}`);
```

EXERCISE 2

Immutable path updates

The first line is a JSON document; every following line is `set <dotted.path> <JSON value>`. Implement `setPath(obj, keys, value)` returning a **new** object with the value placed at the path

— spreading each object on the path, creating intermediate objects when missing, and sharing every untouched subtree. Apply the commands in order, then print `final=<JSON>`, `original=<JSON>` (which must be unchanged) and `shared=<top-level keys whose value is still the same reference in both, joined by ",", or - if none>`.

Example:

```
{"name":"cfg","db":{"host":"x","port":1},"tags":["a"]}
set db.port 5432
set name "prod"
```

→

```
final={"name":"prod","db":{"host":"x","port":5432},"tags":["a"]}
original={"name":"cfg","db":{"host":"x","port":1},"tags":["a"]}
shared=tags
```

INPUT 1

```
{"name":"cfg","db":
{"host":"x","port":1},"tags":["a"]}
set db.port 5432
set name "prod"
```

OUTPUT 1

```
final={"name":"prod","db":
{"host":"x","port":5432},"tags":["a"]}
original={"name":"cfg","db":
{"host":"x","port":1},"tags":["a"]}
shared=tags
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const original = JSON.parse(lines[0]);
function setPath(obj, keys, value) {
  if (keys.length === 0) return value;
  const [head, ...tail] = keys;
  const inner = obj !== null && typeof obj === "object" ? obj[head] : undefined;
  return { ...obj, [head]: setPath(inner ?? {}, tail, value) };
}
let current = original;
for (const line of lines.slice(1)) {
  const [, path, ...rest] = line.trim().split(/\s+/);
  current = setPath(current, path.split("."), JSON.parse(rest.join(" ")));
}
console.log(`final=${JSON.stringify(current)}`);
console.log(`original=${JSON.stringify(original)}`);
const untouched = Object.keys(original).filter((k) => original[k] === current[k]);
console.log(`shared=${untouched.join(",") || "-"}`);
```

EXERCISE 3

Records, destructured everywhere

Each line is `name, age, city`. Using destructuring in **every** callback parameter list — `({ city, name })`, `([a], [b])`, `{ age }` — print the people grouped by city (cities alphabetical, names within a city sorted), one line each as `<city>: <names space-separated>`; then `oldest=<name of the oldest; first one on ties> average=<mean age, 1 decimal>`; then `first=<first person's name> others=<how many follow>` via `const [{ name: first }, ...others] = people`.

Example:

```
Ada, 36, London
Bob, 41, Paris
Cy, 29, London
Dee, 41, Rome
```

→

```
London: Ada Cy
Paris: Bob
Rome: Dee
oldest=Bob average=36.8
first=Ada others=3
```

INPUT 1

```
Ada, 36, London
Bob, 41, Paris
Cy, 29, London
Dee, 41, Rome
```

OUTPUT 1

```
London: Ada Cy
Paris: Bob
Rome: Dee
oldest=Bob average=36.8
first=Ada others=3
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const people = lines.map((l) => {
  const [name, age, city] = l.split(",").map((s) => s.trim());
  return { name, age: Number(age), city };
});
const byCity = people.reduce((acc, { city, name }) => { (acc[city] ??= []).push(name
for (const [city, names] of Object.entries(byCity).sort(([a], [b]) => a.localeCompar
  console.log(`${city}: ${[...names].sort().join(" ")}`);
}
const { name: oldest } = people.reduce((best, p) => (p.age > best.age ? p : best));
const averageAge = people.reduce((sum, { age }) => sum + age, 0) / people.length;
console.log(`oldest=${oldest} average=${averageAge.toFixed(1)}`);
const [{ name: first }, ...others] = people;
console.log(`first=${first} others=${others.length}`);
```

CHECKPOINT QUESTIONS

1. `const o = {}; o[1] = "a"; o["1"] = "b";` — `Object.keys(o).length` is…

A. `2`

B. `1` — property keys are strings; `1` and `"1"` are the same key

C. `0`

D. It throws

2. `JSON.stringify({ a: NaN, b: Infinity })` is…

A. `{"a":NaN,"b":Infinity}`

B. `{"a":null,"b":null}` — non-finite numbers serialise as `null`

C. `{}`

D. A `SyntaxError`

3. `const a = [1, 2]; const b = a; b.push(3);` — `a.length` is…

A. `2`

B. `3` — `b` is an alias; assignment copies the reference, not the array

C. `undefined`

D. It throws

4. Which pair copies one level of an object?

A. `structuredClone`, recursion

B. `{ ...o }` and `Object.assign({}, o)`

C. `Object.freeze(o)`, `Object.seal(o)`

D. `JSON.parse(JSON.stringify(o))`

5. `Object.isFrozen(Object.freeze({ list: [] }).list)` is…

A. `true`

B. `false` — freeze is shallow

C. `undefined`

D. A `TypeError`

6. `const a = [1, 2, 3]; a.length = 1;` leaves `a` as…

A. `[1, 2, 3]` — `length` is read-only

B. `[1]` — assigning a smaller `length` truncates

C. `[1, undefined, undefined]`

D. An error

7. `[3, 20, 100].sort().join()` is…

A. `"3,20,100"`

B. `"100,20,3"` — string comparison: `"1" < "2" < "3"`

C. `"20,3,100"`

D. It throws without a comparator

8. `[1, [2, [3]]].flat()` is…

A. `[1, 2, 3]`

B. `[1, 2, [3]]` — `flat()` flattens one level; `flat(Infinity)` flattens all

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

D. `[1, [2, [3]]]` unchanged

9. `users.filter((u) ⇒ u.id === 7)[0]` is better written as…

A. `users.some((u) ⇒ u.id === 7)`

B. `users.find((u) ⇒ u.id === 7)` — stops at the first match and returns it or `undefined`

C. `users.map((u) ⇒ u.id === 7)`

D. `users.indexOf(7)`

10. `const [x = 0, y = 0] = [5];` gives…

A. `x = 5, y = undefined`

B. `x = 5, y = 0` — positions past the end are `undefined`, so the default applies

C. `x = 0, y = 5`

D. A `TypeError`

11. Making the options argument itself optional in `function f({ a = 1, b = 2 } = {})` is the job of…

A. `a = 1`

B. The trailing `= {}` — without it `f()` destructures `undefined` and throws

C. `b = 2`

D. Nothing; it is optional automatically

12. `Math.max(...hugeArray)` on a few hundred thousand numbers…

A. Is the fastest option

B. Can throw a range/stack error — argument lists have a limit; `reduce` with `Math.max` instead

C. Returns `NaN`

D. Returns `Infinity`

ANSWER KEY · CHECKPOINT — OBJECTS, ARRAYS AND DESTRUCTURING

1. B Use a `Map` when keys must keep their type.

2. B JSON has no NaN or Infinity literals.

3. B `const b = [...a]` would have copied one level.

4. B Nested objects are shared after a shallow copy.

5. B Deep freeze needs recursion.

6. B Assigning a larger `length` extends with holes.

7. B Numeric order needs `(a, b) ⇒ a - b`.

8. B `flatMap` is `map` followed by a one-level `flat`.

9. B `filter(...).length > 0` is likewise `some`.

10. B Defaults fire on `undefined` — including missing positions.

11. B Defaults inside the braces handle missing properties; the outer default handles a missing object.

12. B Spread into a call pushes every element as an argument.

CodeKairo · JavaScript study plan · Module 4: Objects, arrays and destructuring. Solve the exercises online to track your progress and earn XP.