TypeScript Core

1. Values, references & copying

primitives vs reference types — pass-by-sharing: mutating through a param is visible to the caller, reassigning the param is not

structuredClone — deep copy keeping Map/Set/Date/cycles — but throws on functions; a payload holding both a Map and a function has no working clone

deep clone via JSON — JSON.parse(JSON.stringify(v)) loses Map/Set/undefined, and a Date comes back a string while the type still says Date

compare two arrays — length check, then position by position: sorted COPIES via [...a].sort((x,y)=>x-y), then a.every((el,i)=>el===b[i]); includes() throws the index away and fails on duplicates

add/replace key immutably — { ...defaults, ...o } or Object.assign({}, defaults, o) — Object.assign writes into its FIRST argument, so a shared target is poisoned for the process lifetime

conditional key spread — {...(cond && {k:v})} — && not ??, and test !== undefined so 0/'' survive

2. Syntax fluency

optional call / index — fn?.() and arr?.[0] — the short-circuit covers the whole rest of the chain

logical assignment — ||= fires on any falsy left side (a stored 0 is overwritten); ?? = tests nullish only, so the 0 survives; &&= writes only when truthy

destructuring forms — rename {a: b}, default in the pattern AND ? in the type, nested, in params (parens around a destructured arrow param), skip elements [, , z]

destructuring runs the iterator protocol — const [a, b] = x works on ANY iterable; destructuring an ITERATOR consumes what it takes (const [first] = it advances the cursor), an iterable gets a fresh cursor every time

computed keys — {[k]: v}

3. Closures & scope

closures capture the BINDING — not the value — a write after the closure was made is seen; let in a loop head makes a fresh binding per iteration, so fns.push(() => i) gives [0,1,2]

TDZ in callbacks — a closure may REFERENCE a const before its initializer; CALLING it before the initializer runs throws ReferenceError, calling it after is fine

recursion + accumulator — sum(xs.slice(1), acc + xs[0]) — thread the accumulator, shrink the input; xs.at(-1) is a method call, xs.at[-1] silently reads a property off the function

4. Arrays, loops & iteration

reduce — callback (acc, val, i), MUST return the accumulator; count into {} as Record<string, number> with (acc[k] ?? 0) + 1

multi-key comparator — (x, y) => x.ext.localeCompare(y.ext) || y.size - x.size — the || falls through on ties; b - a is descending

sort a copy — [...a].sort((x, y) => x - y) or toSorted — sort mutates and returns the SAME array; default sort is lexicographic

Array.isArray — the narrowing test for string | string[] parameters; typeof [] is only 'object'

which methods mutate — push/pop/shift/unshift/splice/sort/reverse/fill mutate AND return the same array (an alias, not a copy); concat/slice/map/filter/flat/toSorted copy

splice — mutates; returns the array of removed elements; deleteCount 0 inserts without removing; negative start = length + n

predicate searches — find → the element or undefined; some / every / filter take predicates; every([]) is true, some([]) is false

build an n×m grid — Array.from({ length: rows }, () => Array(cols).fill(0)) — braces around length, the comma OUTSIDE them; never Array(n).fill([]) — one shared array in every slot

index + value loop — for (const [i, v] of arr.entries()) — entries() yields [index, value], index first: the reverse of map's (value, index)

switch exhaustiveness — explicit cases with an assertNever(x: never) default make a new union member a compile error

the four range lines — Array(5).map(fn) NEVER runs the callback (map skips holes); Array.from({length: 5}, (_, i) => i) is the form; [...Array(5)].map(...) and Array(5).fill(null).map(...) also work

5. Objects, Map & Set

transform an object — Object.entries(o) → filter/map with ([k, v]) → Object.fromEntries — the full roundtrip in one expression

seen-set idiom — new Set<T>(), check .has BEFORE .add, early-return — O(1) against arr.includes-in-a-loop O(n²)

LRU on a Map — iteration order is insertion order, so move-to-back is m.delete(k); m.set(k, v) and the eviction target is m.keys().next().value

Map ↔ array — [...m] gives [k, v] pairs — filter with ([, v]) => ... and hand the pairs straight back to new Map(...)

keys compare by IDENTITY — an object literal key never hits a Map/Set — use a string key like `${id}:${day}`, or the same reference

plain-object key order — integer-like keys ascending FIRST, then string keys in insertion order — when order matters, use a Map

mutating while iterating a Map — delete the current entry: safe; delete an entry ahead of the cursor: it is skipped; an added entry IS visited (possibly forever)

6. Strings

immutable — every "mutation" returns a new string; there is no s.reverse() — [...s].reverse().join('')

split / join — '2026-08-14'.split('-').reverse().join('/') → '14/08/2026'; split KEEPS empty fields ('1--3' → ['1','','3']); join() defaults to ',' not ''

letter index — ch.charCodeAt(0) - 'a'.charCodeAt(0) — subtract 'a''s code, never a magic 97

extract around a delimiter — s.substring(s.indexOf(':') + 1) — the +1 skips the delimiter; guard idx === -1 or an absent delimiter silently returns the whole string

7. this

the call-site rule — this is whatever sits left of the dot AT THE CALL: obj.m() → obj, arr[0]() → arr, bare f() → undefined, f.call(x) → x, new F() → the new object; extraction (const {m} = o, setTimeout(o.m)) loses it and --strict says nothing

arrow field vs prototype method — an arrow class field pins this per instance (one closure per instance; subclasses are forced into the field form too, TS2425); a prototype method takes this from the call site; an arrow field CAN use super

8. Absence

null vs undefined in an API — { age?: number | null } — absent means leave unchanged, null means erase; the two states stay distinguishable, which is why APIs send null instead of dropping the key

10. Memory & failure modes

the reachability model — GC frees what is UNREACHABLE from the roots, not what is unused; no refcounting, so cycles are fine; a module-level binding is a root, and a live container roots everything it holds

leak: live timer — a pending setInterval/setTimeout is a root — the callback's closure and its this live until clearInterval/clearTimeout

leak: unbounded Map cache — strong keys grow forever; the fixes are delete on a lifecycle end, WeakMap, or a bounded/LRU cache

WeakMap / WeakSet — lets a key be GC'd while the container is alive; the cost is the whole enumeration surface — no size, no iteration, no clear — because collection timing must stay unobservable

leak: closure over a big object — sibling closures share ONE scope object — a never-called sibling that reads huge keeps huge alive for the closure that escapes

11. Classes

get / set accessors — inside get x / set x always this.#x — touching this.x re-invokes the accessor (infinite recursion); a setter takes no return-type annotation

abstract — new AbstractClass() is a compile error (TS2511); a subclass that skips an abstract member is TS2515 — that is the reason to declare it abstract

extends / super — super is for the parent constructor and methods; inherited FIELDS are read via this; a parent method called via super still runs with this = the child instance

12. Errors

throw new Error(msg) — the throw keyword does the raising — new Error(...) alone builds the error and discards it, and execution continues

class X extends Error — .name stays 'Error' until the constructor sets this.name = 'X'; instanceof X still works either way

14. JSON & the module boundary

JSON.parse returns any — the type says Date, the value is a string — the round trip lies; await res.json() and express.json() carry the same hole whether or not you type the words

what stringify drops — undefined and functions as object PROPERTIES are dropped; inside an ARRAY they become null (the length must not change); a Date emits its toJSON ISO string (milliseconds included); a Set/Map becomes {}; give your own class a toJSON

named vs default export — import X from './x' binds whatever was defaulted to ANY local name — a typo compiles and fails at the call site; named imports are checked (TS2305): the argument for named. Producing one is export default class X {}

ESM vs CommonJS — decided by package.json "type" plus the extension — .mjs/.cjs (.mts/.cts) OVERRIDE "type"; never by whether the file says import or require

15. Module resolution (diagnosis, not recall)

moduleResolution: bundler vs node16 — node16/nodenext model Node: extensions required in relative ESM imports (./x.js even from ./x.ts), exports honoured strictly; bundler is extensionless and lax — choosing it for code Node runs fails at runtime, not compile time

exports beats the file layout — once a package declares exports, deep paths that exist on disk are unreachable — the usual cause of "the file is right there and it cannot be found"

esModuleInterop — import x from 'cjs-pkg' binds module.exports only with the interop flag — the symptom is "x is not a function" on a package that plainly exports one

import type — a type-only import must not survive into the emitted JS or it becomes a runtime require; import type states it, verbatimModuleSyntax stops the compiler guessing

failing-import diagnosis order — first the extension in the specifier (ESM relative imports need ./x.js under node16), then which system the file is ("type" + extension), then the package's exports map

14b. Regex

the /g state bug — a shared /g regex literal carries lastIndex across test() calls — re.test inside a filter skips every other hit; drop the /g, build the regex inside the callback, or reset lastIndex

quantifier placement — (\w+) captures the whole run; (\w)+ repeats a one-char group and keeps only the last char

16. Interview-only runtime semantics

the one sanctioned == — null and undefined are ==-equal to each other and to NOTHING else (not 0, not '', not false) — which is why x == null is the single legitimate ==

the 8 falsy values — false 0 -0 0n "" null undefined NaN; the surprises: [] is truthy, and '0'/'false' are truthy — env vars and query params arrive as strings

+ is overloaded, - is not — one string operand turns + into concatenation and the result stays a string down the chain: 100 + '5' + 10 → '10010'

the prototype chain — class FIELDS are own, METHODS live on the prototype (non-enumerable): Object.keys(new Foo()) lists data only; {...instance} and JSON.stringify(instance) copy own enumerable only, so a "copy" loses its methods; in and for...in walk the chain, Object.keys/hasOwn/entries/spread stop at the object

typeof — the 8 results — undefined object boolean number bigint string symbol function; typeof null → 'object'

R0. Construction & new

the new rule — ES6+ = a real class = THROWS without new (Map, Set, WeakMap, WeakSet, Promise); ES5 and earlier = a function = new optional (Array, Object, Error, RegExp)

Date() without new — returns a STRING, not a Date — the one pre-ES6 special case

Number / String / Boolean — bare they are converters to primitives; with new they build OBJECT wrappers — new Number(5) === 5 is false, new Boolean(false) is truthy: never write new with them

new Map(iterable) — an iterable of [k, v] pairs — new Map(Object.entries(o)); new Set(iterable) takes any iterable, flat: new Set('hello') has 4 members

R1. Array — mutators

push / unshift — mutate; return the NEW LENGTH (unshift is O(n))

pop / shift — mutate; return the REMOVED element (shift is O(n))

splice(start, deleteCount, ...items) — mutates; returns the removed elements ([] if none); insert-only is deleteCount 0

sort is stable — since ES2019 equal elements keep their relative order — sort-by-B-then-sort-by-A works

arr.length = n — assigning a smaller length deletes the tail in place; a.length = 0 empties the array

R2. Array — non-mutating

slice — slice() clones the whole array; slice(start, end) is end-EXCLUSIVE; a negative index means length + n

flat / flatMap — flat(depth) defaults to depth 1, flat(Infinity) for all; flatMap = map then flat(1) — return [] from the callback to DROP an element (map + filter in one pass)

the copying twins — toSorted / toReversed / with(i, v) / toSpliced — ES2023 copies of sort / reverse / index write / splice

R3. Array — search

indexOf vs findIndex — a VALUE vs a PREDICATE, both -1 if absent; findLast / findLastIndex scan from the end

find — first match — the ELEMENT or undefined (filter gives the array)

includes / some / every — booleans; some = at least one, every = all; the two short-circuit, which is also the early-exit replacement for forEach

at(i) vs arr[-1] — at(-1) reads the last element; arr[-1] is ordinary property lookup for the key "-1" — undefined, silently

search uses === — indexOf/includes never match an object literal — reference identity; find objects by content with findIndex/some and a predicate

R4. Array — iteration

entries / keys / values — iterators (not arrays): entries yields [index, value] with the index a NUMBER; keys yields indices; values yields elements

for...in on an array — yields STRING keys — '0' + 1 is '01'; for...of yields the values

no break out of forEach — return only skips that element; use for...of with break, or some/every; forEach returns undefined so it never chains

R5. Object statics

Object.keys / values / entries — real ARRAYS of own enumerable string keys — entries yields [key, value] tuples, fromEntries inverts; on an array: [['0', el], ...] with string keys

Object.assign(target, ...src) — mutates and returns TARGET — hence the {} throwaway first argument

Object.hasOwn(o, k) vs k in o — hasOwn is own-only; in includes inherited properties and is true even when the value is undefined

Object.freeze — shallow — nested objects stay writable

R6/R7. Map & Set

set / add / delete returns — map.set and set.add return THE COLLECTION (chainable); .delete returns a boolean; .size is a property, no parens

map.get / map.has — get → the value or undefined; has differs from get(k) !== undefined exactly when the stored value IS undefined

collection iterators — map/set .keys() .values() .entries() hand back one-shot ITERATORS, not arrays; iteration order is insertion order

dedupe / set algebra — [...new Set(arr)]; union = merge + Set, intersect = a.filter(el => b.includes(el)), diff with the !

R8. String API

substring vs slice on negatives — slice(-2) counts from the end; substring CLAMPS negatives to 0 and silently returns the whole string

replace vs replaceAll — replace swaps the FIRST match only; replaceAll every one

pad / repeat / trim — padStart(len, pad) / padEnd pad to a TOTAL length; repeat(n); trim / trimStart / trimEnd

char ↔ code — charCodeAt(i) char → number; String.fromCharCode(n) number → char; str.at(-1) reads the last char

localeCompare — a METHOD on one string taking the other: (a, b) => a.localeCompare(b) → negative/0/positive — the string comparator; < > compare by code unit

String(x) vs x.toString() — both stringify; toString throws on null/undefined

R15. JSON & misc

JSON.stringify(v, replacer, space) — 3rd arg is indentation; JSON.parse throws SyntaxError on bad input

?? vs || — nullish only vs any falsy — they differ on 0 and ''

R16. Iterable vs iterator

the two contracts — iterable: has [Symbol.iterator]() returning an iterator; iterator: has .next() returning {value, done} — an entries-style pair goes INSIDE value, the envelope never changes

every iterator is also iterable — its [Symbol.iterator]() returns itself — why for...of arr.entries() needs no spread

one-shot vs re-walkable — iterators are exhausted after one pass; arrays/Map/Set/strings re-iterate fresh; plain objects are not iterable at all — hence Object.keys/values/entries

what consumes an iterable — for...of, spread, destructuring, Array.from, new Map / new Set, Promise.all

for...of over a Map — yields [key, value] pairs — the loop head is for (const [k, v] of m)

materialise — [...it] or Array.from(it); [...s] yields code points while s.split('') yields UTF-16 units — [..."😀"] is 1 element, split gives 2 broken halves

generator basics — function* — calling it runs NONE of the body and returns a generator object, iterator AND iterable; .next() resumes to the next yield with locals intact

array or iterator? — real arrays: Object.keys/values/entries, Array.from, slice/map/filter/concat, [...x], split; one-shot iterators: arr/map/set .keys/.values/.entries, matchAll, a generator call

return an iterator when — the sequence is lazy, infinite, or expensive per item and walked once; if the whole sequence is already in memory, an iterator only takes capability away

R17. Iterator helpers (ES2025, node 22+)

eleven helpers on every iterator — lazy (return another iterator): map filter flatMap take drop; consuming (drain and return a value): reduce some every find forEach toArray

on the ITERATOR, not the collection — s.drop(2) is TS2339 — ask for the cursor first: s.values().drop(2).toArray()

no sort on an iterator — sort/reverse/at/length do not exist — sorting cannot be lazy; materialise first: [...m.entries()].sort(...)

take(n) / shared cursor — take(n) on an infinite generator pulls exactly n and stops; a chain pulls FROM the source, so consuming the chain advances the original iterator

Iterator.from(x) — wraps an iterable or a bare {next()} object so it gains the helpers

R14. Date & functions

Date.now() vs new Date() — a NUMBER (ms since epoch) vs an object; new Date().getTime() and +date give the same number

date parts — new Date(y, m, d): month 0-indexed, day 1-based; getMonth 0–11; getDate = day-of-month, getDay = weekday (0 = Sunday)

toISOString / arithmetic — always UTC with milliseconds; d2 - d1 coerces to ms (/ 86_400_000 for days); getHours is machine TZ, getUTCHours is safe

fn.length / fn.name — arity = params before the first default or rest; name = the binding it was assigned to ('' for a truly anonymous expression)

Either/or — the deciding question

slice vs splice — copy vs mutate

Object.keys(o) vs map.keys() — a real ARRAY vs a one-shot cursor — the live difference is .sort and index access

concat vs join — a new array vs one string

map vs filter — transform (same length) vs drop (shorter)

for...of vs for...in — values vs keys — and the keys are strings