

**In one line:** A **value type** (struct, enum, tuple) is **copied** on assignment / argument passing — every variable owns an independent value. A **reference type** (class, actor, closure) is **shared** — assignment copies the *pointer*, all variables see one object with an *identity* (===). Default to struct; reach for class when you need identity.

## How it works

- **Equality vs identity:** == = same *value* (Equatable, auto-synthesised for structs/enums whose members conform). === = same *object*; classes only, never synthesised.
- **let differs:** let s = aStruct freezes the *whole value* (no property can change). let c = aClass freezes only the *reference* — c.x = 1 still compiles.
- **mutating:** a struct/enum method *cannot modify self by default* (self is an immutable copy). Mark it mutating ⇒ self is passed as inout; it may assign properties or self = ... (enum state change). Callable **only on a var** — on a let it is a compile error. Classes never need it (they mutate through the reference). nonmutating set = storage lives elsewhere (how @State works).
- **inout** = copy-in / copy-out (write-back on return), not a C pointer. Same var passed twice → exclusivity violation.
- **Copy-on-write (COW):** Array, Dictionary, Set, String are structs that wrap a private **class** buffer. b = a copies the header and bumps the buffer's retain count — O(1), shared. The first *mutation* asks isKnownUniquelyReferenced(&buffer): *false* → copy the buffer (O(n)) then write; *true* → write in place. Keep one owner while mutating in a loop, or every append re-copies.
- **Your own struct gets NO COW for free.** Memberwise copy is eager; a stored class property is copied as a *pointer* (shallow) — both copies share it. Want value semantics over a heap object? Write COW yourself (below).
- **Stack vs heap is an optimiser detail**, not the definition: a struct captured by an escaping closure or boxed in an existential lives on the heap; a class may be stack-promoted. Argue *copy vs share*.

## Choosing

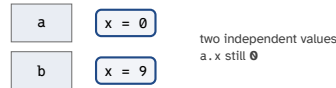
| struct (default)                | class — only when you need          |
|---------------------------------|-------------------------------------|
| models, DTOs, geometry, state   | <b>identity</b> ("the same" object) |
| no identity, thread-safe copies | <b>inheritance</b>                  |
| Sendable almost for free        | <b>ObjC interop</b> / NSObject      |
| SwiftUI state diffs by value    | <b>deinit</b> (cleanup)             |
| protocols for polymorphism      | shared mutable state, resources     |

## Example — COW by hand

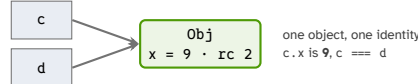
```
final class Box<T> { var v: T; init(_ v: T) { self.v = v } }
struct Pixels { // value semantics
    private var box: Box<UInt8> // shared heap buffer
    init(_ b: [UInt8]) { box = Box(b) }
    var bytes: [UInt8] { box.v } // reads: shared, free
    mutating func set(_ i: Int, _ x: UInt8) {
        if !isKnownUniquelyReferenced(&box) {
            box = Box(box.v) // shared -> copy first
        }
        box.v[i] = x // unique -> in place
    }
}
let p = Pixels([0, 0]); // p.set(0, 9) error: p is a let
```

## Picture — copy, share, copy-on-write

**struct:** var b = a; b.x = 9



**class:** let d = c; d.x = 9



**Array COW:** var b = a, then b.append(4)



**struct holding a class — shallow copy**



## Interview traps

- ! **“What does mutating do?”** (you did not know): value types cannot modify self by default; mutating makes self inout; needs a var; classes exempt.
- ! **“What is COW?”** (you did not know): stdlib collections wrap class storage + isKnownUniquelyReferenced. Say explicitly: *your own structs do not get it*.
- ! Struct vs class (you gave only identity/shared state): list all four — **identity, inheritance, ObjC interop, deinit** — and open with “*prefer struct*”.
- ! “Structs live on the stack” — implementation detail, not semantics.
- ! let arr = [S()]; arr[0].x = 1 — compile error; let freezes the elements too.
- ! Closures are *reference* types: two copies share captured vars.
- ! isKnownUniquelyReferenced takes inout (&box), native Swift classes only; call it inside the mutating method.

## Remember

**Struct = photocopy, class = shared Google Doc, COW = photocopy made lazily at the first pen stroke. I-I-O-D** — Identity, Inheritance, ObjC, Deinit — the four reasons for a class.

## Likely questions

1. b = a for a 1M-element array: cost? — O(1); O(n) on the first write to either.
2. Why mutating on a let? — not allowed: a let value is immutable.
3. == vs ===? — equal value vs same object (classes only).
4. Value types and concurrency? — independent copies, no shared state → Sendable.
5. Enum switching its own case? — a mutating method assigns self = .off.