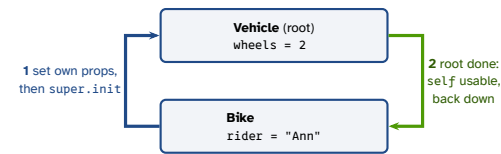


In one line: An initializer must give **every stored property** a value before the instance is used. Structs get a free **memberwise init**; classes split init into **designated** (initialise own properties, **delegate up** to super) and **convenience** (**delegate across** to `self.init`), and run **two-phase initialization**: phase 1 fills memory bottom-up, phase 2 customises top-down.

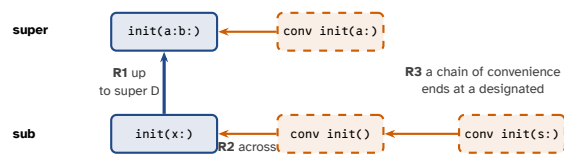
A two-phase init of `Bike(rider:)`



Phase 1 (up): each designated init sets ITS stored props, then delegates up. No `self`, no methods, no reading props — memory is half-formed.

Phase 2 (down): root returns first; each init may now call methods, set inherited props, pass `self` (e.g. `delegate = self`).

B delegation: up, across, end at designated



■ designated: the funnel, calls `super.init` ■ convenience: `self.init` only

Inherited automatically: (1) no designated init of your own (new props defaulted) → all super designateds; (2) all super designateds provided → all its convenience inits too.

How it works

- **Memberwise init** (structs only): synthesized if the *main declaration* has no `init`. `var` props with a default become defaulted params (SE-0242, Swift 5.1); a `let` with a default is excluded. Access: internal at most, private if any stored prop is. Custom init in an **extension** keeps it. Struct inits delegate with `self.init`, no convenience keyword.
- **Four safety checks** (TSPL): (1) designated sets all own props *before* delegating up; (2) designated delegates *up before* assigning an inherited prop; (3) convenience delegates *before* assigning anything; (4) no methods, no prop reads, no `self` as a value until phase 1 ends.
- Overriding a super designated init: override `init`. A **required** init must exist in every subclass (written `required`, not `override`); satisfied by inheritance too. Protocol init requirements force required unless the class is `final`.
- **Failable** `init?` (returns `nil`) / `init!` (I/O result). Classes may return `nil` anywhere, even before all props are set (Swift 2.2+). A failable may delegate to a non-failable; a non-failable override may replace a failable one, never the reverse. `init()` throws when the caller needs *why*.
- **Let** props: assigned exactly once per path during `init`.
- **Observers** `willSet/didSet` do **not** fire for assignments in the class's own `init` (nor for defaults); they **do** fire when a subclass sets an inherited prop in its phase 2.
- **lazy var**: computed on first access, may use `self`; not thread-safe (two threads can both run it); on a struct the first access mutates, so it needs a `var` instance.
- **deinit**: classes (and `~Copyable` structs/enums, Swift 5.9). Runs subclass-first, super's `deinit` is called automatically; never call it yourself.

Remember

“Designated up, convenience across. Phase 1 fills (bottom-up), phase 2 uses (top-down). Own props before super.”

Example — phase 1 protects an override

```
class Vehicle {
    let wheels: Int
    init(wheels: Int) { self.wheels = wheels; setup() } // designated
    convenience init() { self.init(wheels: 4) } // across
    func setup() {}
}
final class Bike: Vehicle {
    var rider: String // no default
    init(rider: String) {
        self.rider = rider // phase 1: own props FIRST
        super.init(wheels: 2) // then up
        print(wheels) // phase 2: self usable
    }
    override func setup() { print(rider) } // runs INSIDE super.init
}
// Bike() is an error: Bike has its own designated init,
// so Vehicle's convenience init() is not inherited.
```

Interview traps

- ! UIKit subclass with `init(viewModel:)` stops inheritance → you must write `required init?(coder:)` (usually `fatalError() / @available(*, unavailable)`).
- ! `self.method()` before `super.init` — compile error (check 4). Assign own props, call `super.init`, then configure.
- ! Setup in `didSet` “does nothing” in `init` — observers skip the own `init`. Call the setup explicitly.
- ! public struct in a package: the memberwise `init` is **internal** — clients cannot call it; write a public `init`.
- ! Adding one `init` to a struct body silently deletes the memberwise `init` for every caller.
- ! Convenience `init` cannot call `super.init` — only `self.init`.
- ! A super `init` calling an overridable method = a subclass sees phase-1 state; keep `init` bodies side-effect-light.

Likely questions

1. Designated vs convenience? — own props + `super.init` vs `self.init`.
2. Own props before `super.init` — why? — super may call an override reading them.
3. When are inits inherited? — no own designated (1); all designateds (2).
4. required? — every subclass must have it; protocol inits, non-final classes.
5. Keep memberwise + add an `init`? — put the new one in an extension.
6. `init?` vs throws? — `nil` if the reason doesn't matter, else throw.