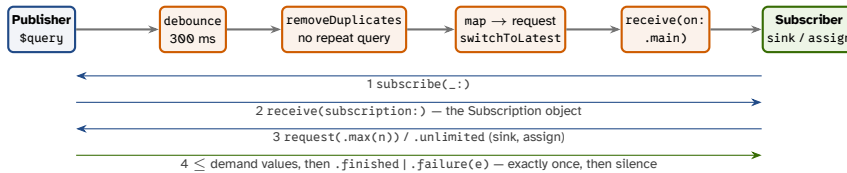
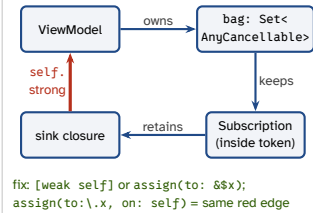


In one line: Four ways to observe on iOS — **NotificationCenter** (broadcast bus), **KVO** (one NSObject property), **Combine** (typed streams + operators + *demand*), **@Observable** (per-property read tracking, iOS 17). Every one hands you a **lifetime object**; the interview is about **who owns it** and **whether the closure owns you back**.

pipeline: values flow right, demand flows left



the leak you missed



The four mechanisms

	Shape / needs	Lifetime handle	Thread / trap
Notif. Center	many-to-many, Notification.Name, untyped userInfo	block API returns a token; call removeObserver(token); center retains the block	runs on the <i>posting</i> thread unless queue: given
KVO	one property; NSObject + @objc dynamic	NSKeyValueObservation — invalidated on deinit	fires on the thread that <i>mutated</i> ; no Swift structs
Combine	typed Output/Failure, operators, demand	AnyCancellable — cancels on deinit	receive(on:); one error ends it
@Observable	macro on a class; iOS 17	SwiftUI: automatic; else withObservationTracking	onChange fires once , on willSet — re-register

How Combine works

- Publisher** = a recipe; nothing runs until a **Subscriber** attaches — a cold one (dataTaskPublisher) re-runs per subscriber (share()) to multicast; Subjects are hot). Future is the exception: runs *eagerly*, once (wrap in Deferred).
- Demand** is backpressure: receive(_:) returns *extra* demand (additive). sink/assign ask .unlimited.
- Subjects**: PassthroughSubject — no value, events only; CurrentValueSubject — holds .value, replays it to new subscribers (state). Dead after send(completion:).
- @Published** emits on **willSet**: inside the sink, self.x is still the *old* value — use the argument.
- Schedulers**: receive(on:) moves *downstream* delivery (put it just before UI work); subscribe(on:) moves the *upstream* subscribe / request / cancel work. DispatchQueue.main over RunLoop.main (paused while the user drags a scroll view).

Operators — the pairs they ask

- debounce = emit last after a *quiet gap* (search box); throttle = at most one per interval (Latest: first/last).
- combineLatest = latest of each on any change, silent until *all* emitted once; zip = pairs by index; merge = interleave same type.
- flatMap keeps every inner alive (maxPublishers:); map+switchToLatest cancels the stale one.
- catch swaps in a fallback *publisher*; replaceError a value. Both still end the upstream — do it *inside* the inner publisher. removeDuplicates needs Equatable.

Example — typeahead VM, no cycle

```

final class SearchVM: ObservableObject {
    @Published var query = ""
    @Published private(set) var results: [Item] = []
    private var bag = Set<AnyCancellable>()
    init(api: API) {
        $query
            .debounce(for: .milliseconds(300),
                      scheduler: DispatchQueue.main)
            .removeDuplicates()
            .map { api.search($0).replaceError(with: []) }
            .switchToLatest() // drop the stale request
            .receive(on: DispatchQueue.main)
            .assign(to: &$results) // no self capture, no token
            // LEAK: .sink { self.results = $0 }.store(in: &bag)
            // FIX: .sink { [weak self] in self?.results = $0 }
    }
}

```

Interview traps

- The cycle (you missed it):** self → bag → cancellable → sink closure → self. deinit never runs, so the bag never cancels. [weak self].
- assign(to: \.x, on: self) retains self strongly. assign(to: &\$x) (iOS 14) ties the subscription to the @Published storage — no cycle, no AnyCancellable.
- Dropped cancellable (_ = pub.sink{ }) cancels at once — “my sink never fires”.
- Block-based NotificationCenter: keep the token and **call removeObserver(token)**; do not rely on releasing it. The block is retained by the center — [weak self] inside.
- An error reaching the sink *terminates* the subscription: a search box stops working after the first failed request.
- subscribe(on:) does *not* make the sink run on that queue.

Remember

Values right, demand left, cancellable owns the chain, closure must not own self. @Published = willSet. Passthrough = event, CurrentValue = state.

Likely questions

- sink not firing? — AnyCancellable not stored.
- VM never deinit? — sink/assign(on: self) captures self.
- Debounce vs throttle? — quiet gap vs fixed rate.
- @Observable vs ObservableObject? — per-property reads vs whole-object objectWillChange.
- Demand? — how many more values the subscriber accepts.