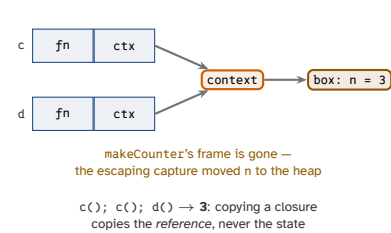
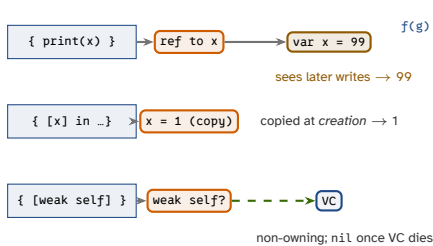


In one line: A closure is a **function + a context** holding what it captured, and it is a **reference type**. By default it captures **variables**, not values (a captured var moves into a shared heap box); a **capture list** copies values *when the closure is created*. Parameters are **non-escaping** by default — @escaping says the closure may outlive the call, and that is what brings heap contexts, ARC and cycles.

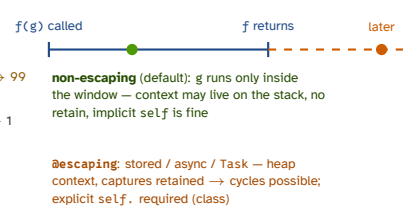
A Let d = c — one context, shared



B What the context holds



C Escaping = outlives the call



How it works

- **Capture:** without a list a closure refers to the *variable*; reads and writes are shared with the enclosing scope. `[x]` is an immutable copy made when the closure is *created*; for a class it copies the pointer (same object). `[weak self]`, `[unowned self]`, `[y = expr]` are the same mechanism.
- **Non-escaping by default** (Swift 3, SE-0103): may only be called during the call, cannot be stored or captured by an escaping closure. An **optional** closure parameter `() -> Void`? is **implicitly escaping**.
- **Implicit self** in escaping closures: allowed when `self` is a value type or listed as `[self]` (5.3, SE-0269), and after guard `let self` inside `[weak self]` (5.8, SE-0365).
- **withoutActuallyEscaping(f) { g in ... }** lends a non-escaping `f` to an API typed @escaping (e.g. `lazy.filter`); at the end of the block the runtime checks `g` was not kept — if it was, **crash**.
- **@autoclosure () -> T:** the caller writes a plain expression, the callee decides *whether and when* to evaluate it (assert, precondition, `??`, `&&`, `||`, `XCTAssert*`, `@StateObject(wrappedValue:)`). Forward it by calling it: `inner(msg())`. Combine with @escaping to store it.
- **@Sendable** (5.5, SE-0302): may run concurrently, so every capture must be Sendable and a local var can't be captured at all (not even read) — copy it with `[x]`. Swift 6: `Task {}` takes a sending closure, so a non-Sendable value may go in if nothing uses it afterwards.
- **Function types:** `(Int) async throws -> Int`; rethrows HOFs (map). `Type.method` is curried `(Type) -> (Args) -> R`; `obj.method` is a closure that **retains** `obj`.
- **Trailing closures:** last closure after the parens; **multiple** (5.3, SE-0279): first unlabeled, the rest keep their labels.

Example — the syntax ladder

```
names.sorted(by: { (a: String, b: String) -> Bool in
    return a < b }) // full form
names.sorted(by: { a, b in a < b }) // implicit return
names.sorted { $0 < $1 } // shorthand + trailing
names.sorted(by: <) // operator = function
users.map(\.name) // key path (5.2)
UIView.animate(withDuration: 0.3) { v.alpha = 0 }
completion: { _ in v.removeFromSuperview() } // 2 trailing
```

Example — capture

```
func makeCounter() -> () -> Int {
    var n = 0 // boxed: outlives the frame
    return { n += 1; return n }
}
let c = makeCounter(); let d = c // one shared context
c(); c(); print(d()) // 3
var x = 1
let live = { print(x) } // captures the variable
let snap = { [x] in print(x) } // copies the value now
x = 99; live(); snap() // 99, then 1
var fs: [() -> Int] = []
for i in 0..<3 { fs.append { i } } // new i each pass: 0 1 2
var j = 0
while j < 3 { fs.append { j }; j += 1 } // one j: 3 3 3
func debugLog(_ m: @autoclosure () -> String) {
    if isDebug { print(m()) } // m may never run
```

Interview traps

- ! **Loop trap:** `for-in` makes a *fresh* binding per iteration (`-> 0 1 2`); one var mutated by a `while` is shared (`-> 3 3 3`).
- ! **Closures have no identity:** not Equatable, no `===` — to remove a handler, register it under a token/ID.
- ! `button.onTap = handleTap` passes `self.handleTap`: `self` is retained strongly `-> cycle`. Wrap it: `{ [weak self] in self?.handleTap() }`.
- ! An escaping closure in a struct's mutating method cannot capture `self` (it is inout) — copy the values you need first.
- ! A trailing closure inside an `if/guard` condition confuses the parser: `if xs.contains(where: { $0 > 3 }) { }`.
- ! `@autoclosure` hides laziness from the caller: a side effect in the argument may run later, twice, or never.
- ! `defer` is not a closure: it reads the variable's value at scope exit.

Remember

"Variables by reference, lists by value, escaping to the heap."

Likely questions

1. Why non-escaping by default? — no heap context, no retains, no cycles, implicit `self`.
2. `[x]` vs no list? — snapshot at creation vs the live variable.
3. @autoclosure examples? — `assert`, `??`, `&&`.
4. Can a @Sendable closure mutate a captured var? — no; copy or use an actor.