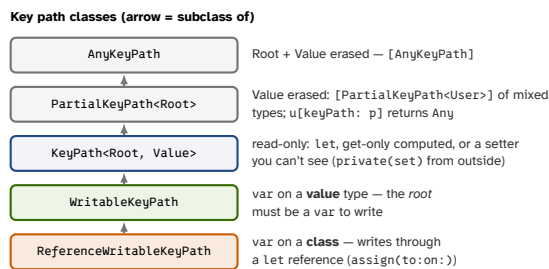
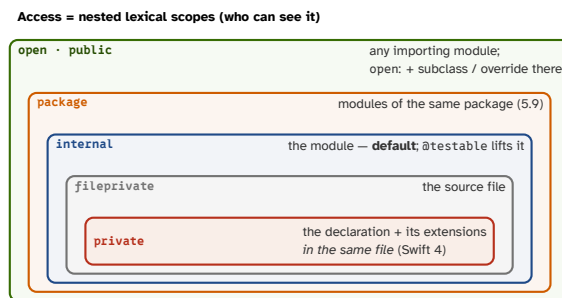


In one line: A **key path** (`\User.name`) is a typed, reusable, *uninvoked* reference to a property — a class instance you apply with `x[keyPath: kp]`; its class (read-only vs writable) is decided by what *you* may do with the property. **Access control** is lexical: six levels, default internal, and nothing may be exposed through something less visible than itself.



The compiler infers the most specific class for a literal.



How it works — key paths

- **Components:** `\User.address.city`; optional chaining `\.address?.city` (Value becomes `String?`); subscripts `[\Int].[]`; identity `\.self`; `kp.appending(path:)`. Key paths are Hashable — usable as dictionary keys.
- **As functions** (5.2, SE-0249): `users.map(\.name)`, `filter(\.isActive)`. Only a key path **literal** converts; a stored let kp needs `{ $0[keyPath: kp] }`.
- **@dynamicMemberLookup with key paths** (5.1, SE-0252): `subscript<T>(dynamicMember kp: KeyPath<W, T>) -> T` forwards *real* members, type-checked (string-keyed lookup is not). SwiftUI: `$model.name` is Binding's dynamic member lookup over a `WritableKeyPath`.
- **Where you meet them:** `ForEach(items, id: \.id)`, `SortDescriptor/KeyPathComparator(\.name)`, KVO `observe(\.count)` (needs `@objc dynamic`), Combine `assign(to:on:)`.

How it works — access rules

- **No leaking:** a declaration can't be more visible than the types in its signature (`public func f(_: InternalType)` is an error). Tuple/function types take the *lowest* level of their parts.
- **Member defaults:** members of a public type are internal unless marked; members of a private/fileprivate type are fileprivate. Enum cases and protocol requirements take the enclosing level.
- **Setters:** `public private(set) var count` — read everywhere, write in the scope.
- **Inheritance:** a subclass can't be more visible than its superclass; an override *may* widen access.
- **open vs public** (classes only): `public` = use it outside the module; `open` = also subclass/override it there. `final` = nobody, anywhere — and lets the compiler call it directly (private lets the optimiser *infer* final; final is the guarantee).
- **@testable import:** tests see internal (build with `ENABLE_TESTABILITY`, on in Debug), never private/fileprivate.
- **@inlinable:** the body ships in the module interface so clients can inline and specialise it; it may use only public or `@usableFromInline` declarations. `@usableFromInline internal` = ABI-visible, source-invisible.
- Swift 6 (SE-0409): `internal import X / private import X` keep a dependency out of your module's public interface.

Example

```

public struct User {
    public let id: UUID
    public private(set) var name: String // write: module only
    var visits = 0 // internal
    public init(id: UUID, name: String) { // memberwise is internal
        self.id = id; self.name = name
    }
    let ids = users.map(\.id) // literal -> function
    let byName = users.sorted(using: KeyPathComparator(\.name))
    @dynamicMemberLookup struct Box<W> {
        var wrapped: W
        subscript<T>(dynamicMember kp: WritableKeyPath<W, T>) -> T {
            get { wrapped[keyPath: kp] }
            set { wrapped[keyPath: kp] = newValue }
        }
    }
    var box = Box(wrapped: user); box.visits += 1 // checked
  
```

Interview traps

- ! **A public struct has no public init:** the memberwise init is internal (lower still if a stored property is private). Clients can't construct it until you write `public init`.
- ! `\User.name` is a `WritableKeyPath` inside the module and a plain `KeyPath` outside — `private(set)` decides. Key paths never bypass access.
- ! `public class` is **not** subclassable by clients — that is open. Making an open class `public/final` later breaks every client that subclassed it.
- ! `assign(to: \.text, on: self)` **retains** self → cycle when self holds the `AnyCancellable`. Use `assign(to: &$prop)` (iOS 14) or `sink + [weak self]`.
- ! `@inlinable` code is compiled *into clients*: a fix to its body reaches them only when they rebuild.
- ! `fileprivate` is only needed to share between *different* types in one file; same-type extensions in the file already see private.

Remember

“Key path = a property as a value; its class = your rights.” Access: **“O-P-P-I-F-P”** — open, public, package, internal, fileprivate, private; public *uses*, open *extends*.

Likely questions

1. `WritableKeyPath` vs `ReferenceWritableKeyPath`? — value root must be var; class root writes through `let`.
2. Why `PartialKeyPath`? — a mixed-type list of one root's properties.
3. open vs public? — only open is subclassable by clients.
4. What does `@testable` expose? — internal, not private.
5. Default access of a public type's members? — internal.