

In one line: `@propertyWrapper` (SE-0258, Swift 5.1) is a type with a `wrappedValue`; annotating `@W var x` makes the compiler store a hidden `_x: W` and turn `x` into a **computed property** forwarding to `_x.wrappedValue` — plus `$x` for `projectedValue`. SwiftUI's `@State`, `@Binding`, `@AppStorage` and Combine's `@Published` are just such types.

A you write → the compiler writes

```
@Clamped(0...100) var volume = 50
```

```
private var _volume = Clamped(wrappedValue: 50, 0...100)
var volume: Int {
  get { _volume.wrappedValue }
  set { _volume.wrappedValue = newValue }
}
var $volume: _ { _volume.projectedValue }
```

= 50 feeds `init(wrappedValue:)`, attribute args follow it. `_x` is private; `$x` exists only if `projectedValue` does. Memberwise `init` takes `volume: Int` when the wrapper has `init(wrappedValue:)`, else the wrapper type.

B @Published reaches its owner

```
vm.name = "Ann"
```

```
static subscript(_enclosingInstance: vm,
  wrapped: \.name, storage: \._name)
{
  vm.objectWillChange.send()
}
```

```
$name subscribers get "Ann"
```

both in `willSet`: `vm.name` still old here
no `self` in a struct → `@Published` is class-only

C @State: handle vs storage

```
Counter()
_n: State<Int>
```

```
graph storage
  n = 3
```

```
$n: Binding<Int>
```

view struct is rebuilt every update;
storage lives in SwiftUI, keyed by identity.
nonmutating set: writable from body.
Initial value used only the first time.

How it works

- **Required:** `var wrappedValue`. Optional: `projectedValue` (any type — `Binding`, a publisher, the wrapper itself), `init(wrappedValue:)` (enables = default), `init(projectedValue:)` (enables passing `$arg`).
- **Composition** `@A @B var x: T: storage _x: A<B<T>>`, access `_x.wrappedValue.wrappedValue`; outer to inner, left to right; `$x` is the **outermost** wrapper's projection.
- **Where:** stored `var` properties of types; **locals** (Swift 5.4); function and closure **parameters** (SE-0293, 5.5) — that is how `ForEach($items) { $item in TextField("", text: $item.name) }` works (`Binding` has `init(projectedValue:)`).
- **Enclosing-instance subscript** (underscored, used by `@Published`): if the wrapper declares `static subscript(_enclosingInstance:wrapped:storage:)`, the compiler routes access through it with the **owning object** and two key paths — so the wrapper can reach `self` (e.g. `objectWillChange`). Class-only: `Published.wrappedValue` is marked unavailable, “*@Published is only available on properties of classes*”.
- **@State:** a `DynamicProperty` holding a handle; SwiftUI finds it on the view, allocates storage in its graph (per view *identity*), updates the handle before body. `$n = Binding`.
- **@Binding:** a struct of `get/set` closures to someone else's storage; `@dynamicMemberLookup`, so `$user.name` is a `Binding<String>`. No `init(wrappedValue:)` → the child's memberwise `init` takes a `Binding: Child(n: $count)`.
- **@AppStorage("key"):** a `DynamicProperty` over `UserDefaults`; the view re-renders when that key changes; `$ = Binding`. `Bool/Int/Double/String/URL/Data + RawRepresentable`.

Limits

- Not on `let`, lazy, weak/unowned, computed properties, or a property with its own `get/set` (`willSet/didSet` are allowed).
- A protocol cannot require a wrapper — only the plain property.
- Inline storage (`@Clamped`): the setter is mutating — needs a `var owner`; external storage (`@State`) uses nonmutating `set`.

Example — a wrapper + the enclosing subscript

```
@propertyWrapper struct Clamped<V: Comparable> {
  private var value: V; let range: ClosedRange<V>
  init(wrappedValue v: V, _ r: ClosedRange<V>) {
    range = r; value = Self.clamp(v, r) } // = 50 lands here
  var wrappedValue: V {
    get { value }
    set { value = Self.clamp(newValue, range) } } // mutating set
  var projectedValue: ClosedRange<V> { range } // $volume
  static func clamp(_ v: V, _ r: ClosedRange<V>) -> V {
    min(max(v, r.lowerBound), r.upperBound) }
}
// the shape @Published uses to reach its owner:
static subscript<Owner: AnyObject>(_enclosingInstance o: Owner,
  wrapped: ReferenceWritableKeyPath<Owner, Value>,
  storage: ReferenceWritableKeyPath<Owner, Self>) -> Value
```

Interview traps

- ! `self.count = 5` in a View's `init` does not seed `@State` — write `_count = State(initialValue: 5)`; and only the first `init` wins.
- ! `@State var items = load()` runs `load()` on every view `init`, result discarded after the first.
- ! `$vm.name.sink { }` reading `vm.name` inside gets the **old** value (`willSet`); use the closure's parameter.
- ! Two wrappers: `$x` projects only the outermost one.
- ! Wrappers hide semantics (persistence, threading) at the call site — great for cross-cutting concerns, costly when overused.

Remember

“**x is a computed façade, _x is the wrapper, \$x is the projection. Storage inside = value; storage outside = survives the struct.**”

Likely questions

1. What is generated? — private `_x: W`, computed `x`, optional `$x`.
2. How does `@Published` notify? — enclosing-instance `subscript` → `objectWillChange`.
3. Why does `@State` survive re-creation? — storage lives in SwiftUI's graph.
4. How does `@Clamped(0...100) var v = 50 init?` — `init(wrappedValue: 50, 0...100)`.