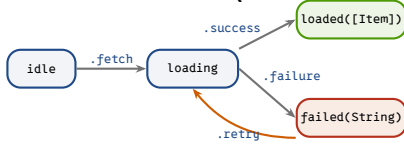
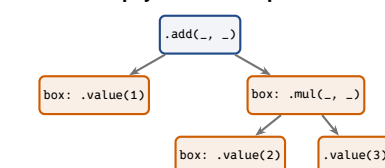


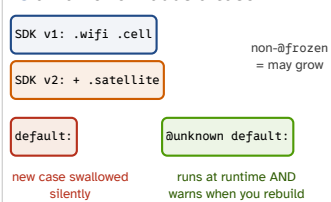
**In one line:** An enum value is **exactly one** of its cases (a *sum type* / tagged union); **raw values** are one compile-time constant per case, **associated values** are per-instance payloads. **switch** must be **exhaustive**, patterns bind and filter, and for library enums `@frozen` vs `@unknown` default decides who may add a case.

**A enum = state machine (code below)**

one case at a time: no `isLoading` && `error != nil`  
impossible combos; a payload exists only in its state

**B indirect: payload in a heap box**

without it: infinite size → compile error.  
boxes are refcounted + immutable: value semantics kept

**C a framework adds a case****How it works**

- Raw values** enum `S: Int { case ok = 200 }`: one literal type, fixed at compile time, gives `RawRepresentable` + `failable init?(rawValue:)`. Implicit: `Int` counts up from the previous value (start 0); `String` = the case name. **Raw XOR associated** — never both.
- Associated values** case `user(id: Int, name: String)`: payload per instance, different types per case. Size = largest payload + tag (spare bits reused); payload-less: 1 byte up to 256 cases, a single case: 0 bytes.
- indirect** on the enum or one case: payload boxed on the heap → recursive types (trees, linked lists, ASTs).
- Synthesis:** payload-less enums are `Equatable` + `Hashable` *without declaring it*. With payloads: declare it, all payloads must conform. Comparable by declaration order (Swift 5.3, SE-0266; not for raw-value enums); `Codable` with payloads (5.5, SE-0295); `CaseIterable` → `allCases` only without payloads.
- No stored instance properties (static ones are fine); methods, computed props, mutating func that assigns `self = .x`. A caseless enum API `{}` is a namespace that cannot be instantiated.
- Exhaustive switch:** no implicit fallback; `fallthrough` skips the next pattern check and cannot enter a case that binds variables. Swift 5.9: `let x = switch s { ... }` (SE-0380).
- Library evolution** (`BUILD_LIBRARY_FOR_DISTRIBUTION`): enums are *non-frozen* by default — clients need `@unknown` default (SE-0192, Swift 5). `@frozen` (SE-0260) promises no new case ever: faster layout, no `@unknown` needed, adding a case breaks ABI. ObjC: `NS_ENUM` imports non-frozen, `NS_CLOSED_ENUM` frozen. Without library evolution every enum is effectively frozen: a new case breaks clients' switches at **compile time**.

**Example — a state machine**

```
enum Load { case idle, loading, loaded([Item]), failed(String) }
enum Event { case fetch, retry, success([Item]), failure(String) }
extension Load {
    mutating func send(_ e: Event) {
        switch (self, e) {
            case (.idle, .fetch), (.failed, .retry): self = .loading // tuple pattern
            case (.loading, .success(let items)): self = .loaded(items)
            case (.loading, .failure(let m)) where !m.isEmpty: self = .failed(m) // illegal: ignore
            default: break
        }
    }
}
if case .loaded(let items) = state { show(items) }
for case .failed(let m) in history { log(m) } // filter + bind
```

**The 8 pattern kinds (TSPL)**

| Pattern       | Looks like                                  | Note                                |
|---------------|---------------------------------------------|-------------------------------------|
| wildcard      | -                                           | matches, ignores                    |
| identifier    | <code>x</code> (in <code>let x = _</code> ) | binds anything                      |
| value-binding | <code>let (x, y), .a(let x)</code>          | <code>let</code> distributes inward |
| tuple         | <code>(200, _), (let c, true)</code>        | element-wise                        |
| enum case     | <code>.failed(let m), .some(x)</code>       | works on <code>Optional</code> too  |
| optional      | <code>let x?</code>                         | sugar for <code>.some(x)</code>     |
| type-casting  | <code>is Int, let s as String</code>        | check / check+cast                  |
| expression    | <code>400 .. &lt;500, "GET"</code>          | calls <code>~=</code>               |

- `~=`: default for `Equatable` is `==`; ranges use `contains`. Overload static func `~=(pattern: P, value: V) -> Bool` to match your own (e.g. a predicate, a regex).
- where is a **guard on the case**, not a pattern: runs after binding.
- case `.a(let x), .b(let x)`: — every alternative must bind the **same names with the same types**.
- Where patterns live: `switch`, `if/guard/while` case, for case ... `in`, `catch`, and every `let`.

**Interview traps**

- ! `default:` on an SDK enum hides new cases; `@unknown` default still warns about the ones you did not list.
- ! Persisted raw values: renaming a case (`String raw = name!`) → `init?(rawValue:)` gives `nil`. Pin raw values.
- ! `Equatable` on a payload enum is not free: one non-`Equatable` payload and synthesis fails.
- ! Adding a case to a public enum is a breaking change either way — compile-time (source libs) or runtime-path (`@unknown`).
- ! `state == .loading` needs `Equatable` (payloads too); `if case .loading = state` needs nothing.

**Likely questions**

- Raw and associated at once? — no; mutually exclusive.
- Why indirect? — a recursive payload needs a box.
- `@unknown` default vs `default`? — future cases + keeps the warning.
- Why enums over flags/optionals for state? — illegal states unrepresentable.
- How does case 1 ... 5 match? — expression pattern via `~=`.
- Payload without a switch? — `if case let, or a computed var data: Data?`.

**Remember**

**“Raw = a label, associated = a payload; switch covers all; frozen = closed forever.”**