ML-Style Domain Modeling in Kotlin — Cheat Sheet

Goal

- Make invalid states unrepresentable and push correctness into the type system, so: 'If it compiles, it’s probably correct.'

Core Principles

- Explicitness over convention: Model possibilities as types, not booleans.
- Constrain at the edges: Validate at construction; trust inside the domain.
- Purity in the core: Keep domain logic total and side-effect free; push I/O to adapters.
- Exhaustiveness: Compiler must force you to handle every case.

Kotlin Toolkit

- Sealed interface/class — closed sets of possibilities (sum types).
- Data class — structured entities (product types).
- JvmInline value class — zero-cost newtypes to prevent type confusion.
- Non-nullable types by default; use T? only when absence is part of the model.
- Smart constructors — private ctor + factory functions to enforce invariants.
- Exhaustive when expressions — compiler ensures every case is handled.
- Enum class — closed sets of simple alternatives.
- Generics + variance annotations — strong type relationships.
- Immutability first (val by default).
- Null-safety operators — force explicit handling of absence.
- Extension/infix functions — fluent, typesafe domain transformations.

Arrow Toolkit

- Option<A>, Either<E, A> — explicit presence/failure instead of null/throw.
- ValidatedNel<E, A> — accumulate validation errors instead of failing fast.
- NonEmptyList<A> — guarantee non-empty collections.
- Raise + context parameters (context(Raise<E>)) — ergonomic, typesafe error handling.
- Optics (Lens, Prism) — lawful navigation/updates for nested immutable data.
- Effect discipline — keep Either/Validated in domain, I/O at edges.

Construction Patterns

- Refined values via value classes + smart constructors.
- Closed domain modeling via sealed interfaces and data classes.
- Exhaustive when to handle all cases.

Error Handling Recipes

- Happy-path ergonomics with Raise — track errors in the type system, write code as if success only.
- Accumulate all errors with ValidatedNel — useful for forms/aggregates.

Module Boundaries

- domain (pure): ADTs, value classes, smart constructors, no suspend or I/O.
- app/service (impure): orchestrates effects, interprets domain.
- adapters: DB/HTTP/IO; convert raw data → domain via safe constructors.

Testing & Guarantees

- Property tests for invariants (Kotest property).
- Exhaustiveness tests ensure sealed types fully covered.
- No defensive checks inside domain code — constructors guarantee validity.

Anti-Patterns

- Raw primitives/strings for domain values (stringly-typed).
- Public constructors that bypass validation.
- Nullable everywhere instead of Option or ADTs.
- Catch-and-forget exceptions in domain logic.
- Boolean flags hiding state machines.

Quick Checklist

- All domain choices are sealed ADTs.
- Every primitive wrapped in a value class with smart constructor.
- Public APIs return Either/Validated or use context(Raise<E>).
- No suspend or I/O in the domain module.
- All when over sealed types are exhaustive.
- Property tests cover constructors and invariants.