03 · Object Oriented Programming

Flutter Practice Notes

Contents

03 · Object Oriented Programming

Introduction

OOP is how you **model a domain** in Dart: turning real-world concepts (a user, an account, a payment) into classes with state, behavior, and relationships. Flutter itself is deeply OO — every widget, element, and render object is a class in a hierarchy. This module teaches OOP as a *design* skill, not just syntax.

Why this module exists

Interviewers probe OOP because it reveals whether you can structure code that scales: encapsulation that protects invariants, inheritance vs composition tradeoffs, polymorphism for extensibility, and correct equality/identity. These decisions determine whether a codebase stays maintainable at 50k+ lines.

Module map

#	File	Topic	Level
1	01_classes_and_objects.md	Classes, objects, fields, methods, `static`, `this`, lifecycle, memory layout	🟢
2	02_encapsulation.md	Privacy (`_`), getters/setters, protecting invariants	🟢
3	03_inheritance.md	`extends`, `super`, `@override`, abstract base classes	🔵
4	04_polymorphism.md	Dynamic dispatch, method overriding, Liskov substitution	🔵
5	05_abstraction_and_interfaces.md	Abstract classes, implicit interfaces, `implements`, `sealed`	🔵
6	06_composition_and_relationships.md	Composition vs inheritance, aggregation, association, mixin-vs-composition	🔵
7	07_equality_and_copying.md	`==` / `hashCode`, identity, deep vs shallow copy	🔵

> **Cross-references:** Constructors (named/factory/private/`const`) and **mixins** are covered in depth in 02 Advanced Dart (09_constructors_and_singletons.md, 06_mixins.md). Dependency Injection has its own module (14). Immutability/value equality tooling is in 10_immutability.md.

Prerequisites

01 Dart Fundamentals and the constructors/mixins/immutability files of 02 Advanced Dart.

What you'll be able to do after this module

- Model a domain with well-encapsulated classes that protect their invariants.
- Choose **composition over inheritance** deliberately and justify it.
- Use abstract classes, interfaces, and sealed types to design for extension.
- Implement correct value equality and understand deep vs shallow copy.
- Explain dynamic dispatch and the Liskov Substitution Principle (bridge to SOLID, Module 04).

Capstones

Tier	Build
Beginner	A `Shape` hierarchy (area/perimeter) with polymorphism.
Intermediate	A `BankAccount → SavingsAccount` domain with encapsulated invariants + custom exceptions.
Advanced	A `PaymentMethod` design using sealed classes + composition.
Enterprise	A small **domain model** (orders/inventory) with value objects, aggregates, and equality (feeds DDD, Module 46).

Summary

OOP here is about *design judgment*: encapsulate to protect invariants, prefer composition, use polymorphism and abstraction for extensibility, and get equality/identity right. These are the foundations for SOLID, design patterns, and clean architecture.

Classes & Objects (Fields, Methods, `static`, `this`, Lifecycle)

> A class is a blueprint describing state (fields) and behavior (methods); an object is a concrete instance built from that blueprint and living on the heap.

Introduction

The class/object distinction is the atom of OOP. This file covers declaring classes, instance vs `static` members, `this`, getters, the object's lifecycle (construction → use → collection), and its memory layout. Constructors are covered in depth in 02 · constructors_and_singletons; here we focus on structure and lifecycle.

Why this concept exists

Programs manipulate *things* with state and behavior. Grouping related data and the operations on it into a class gives you a reusable, testable unit and a vocabulary that mirrors the problem domain — the essence of modeling.

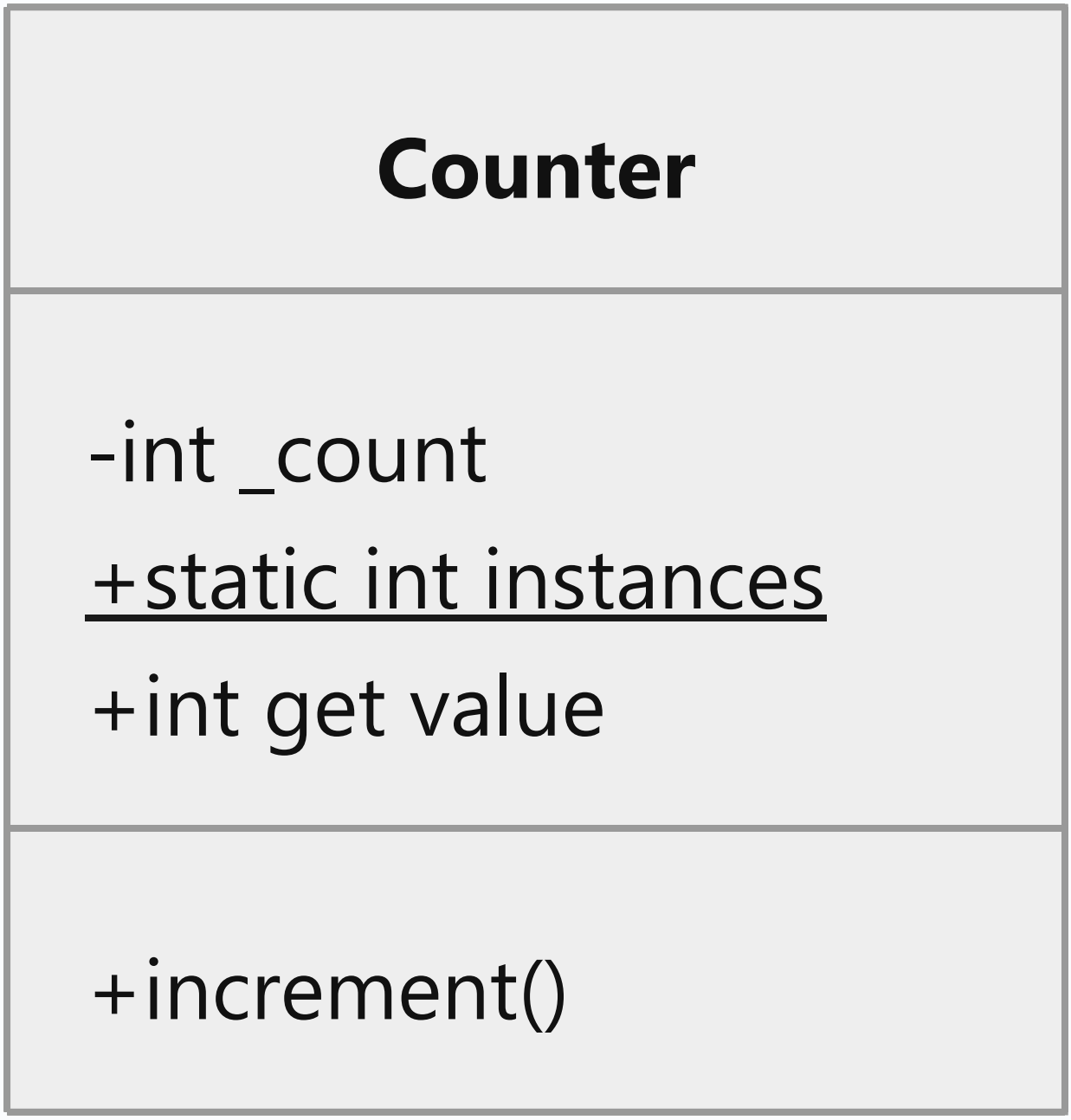
Real-world analogy

A class is an **architectural blueprint**; an object is a **house built from it**. Many houses (objects) share one blueprint (class). Instance fields are each house's own furniture; `static` members are shared neighborhood facilities (one park for all houses).

Problem Statement

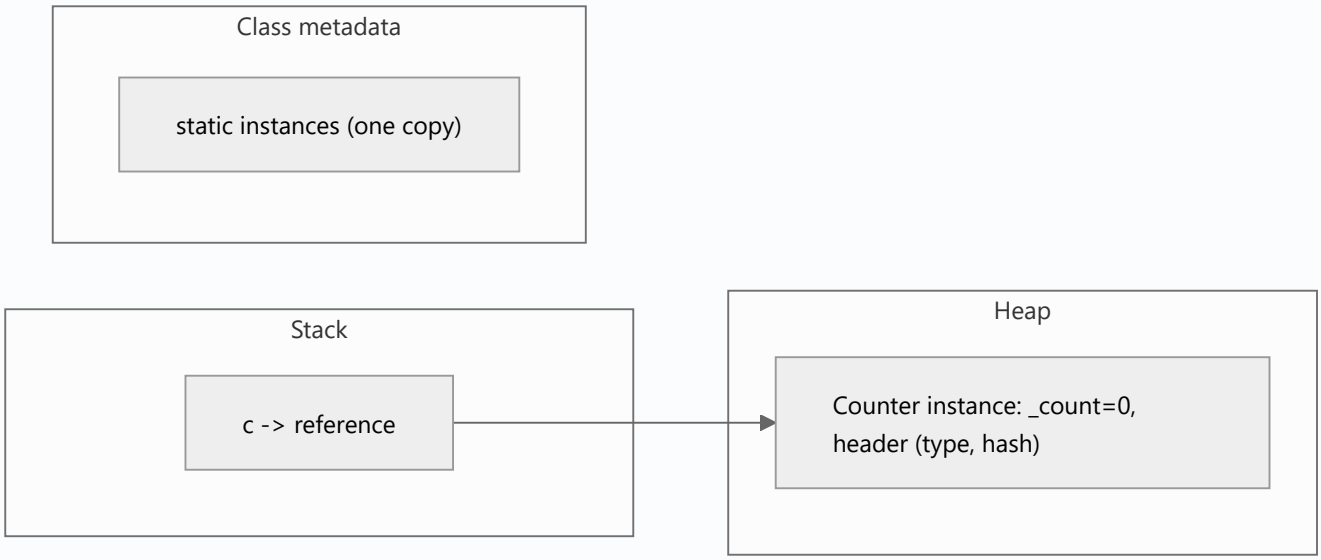
Model a `Counter` (instance state + methods), track how many counters exist (`static`), expose a read-only computed value (getter), and understand when the object becomes eligible for garbage collection. You'll build it and reason about its lifecycle.

Internal Working



- **Instance members** belong to each object (`_count` , `increment`).
- `static` **members** belong to the class itself (shared): `static int instances` .
- `this` refers to the current instance; needed to disambiguate (`this.x = x`) or pass the object.
- **Getters/setters** expose computed/controlled access (`int get value => _count`).
- A class with no explicit constructor gets a default no-arg one.

Memory Representation



- An object on the heap holds its **instance fields** plus a header (type info, identity hash).
- `static` **fields** live once per class (not per instance).
- The variable holds a **reference** (on the stack/enclosing object), not the object itself — Dart is reference-semantics for objects.

Compiler Behavior

- Field access/method calls are type-checked against the static type.
- `static` members are resolved via the class, not an instance (`Counter.instances`).
- Getters look like fields at call sites (`c.value`) — uniform access principle.

Runtime Behavior

- `Counter()` allocates a heap object, runs the constructor, returns a reference.
- Instance methods dispatch on the runtime type (virtual dispatch — see 04_polymorphism.md).
- The object is collectible once unreachable (02 · memory_and_gc).

Flutter Engine Behavior

Not applicable directly. (Widgets/Elements/RenderObjects are all classes/objects; `State` objects have a framework-managed lifecycle — Module 08.)

Dart VM Behavior

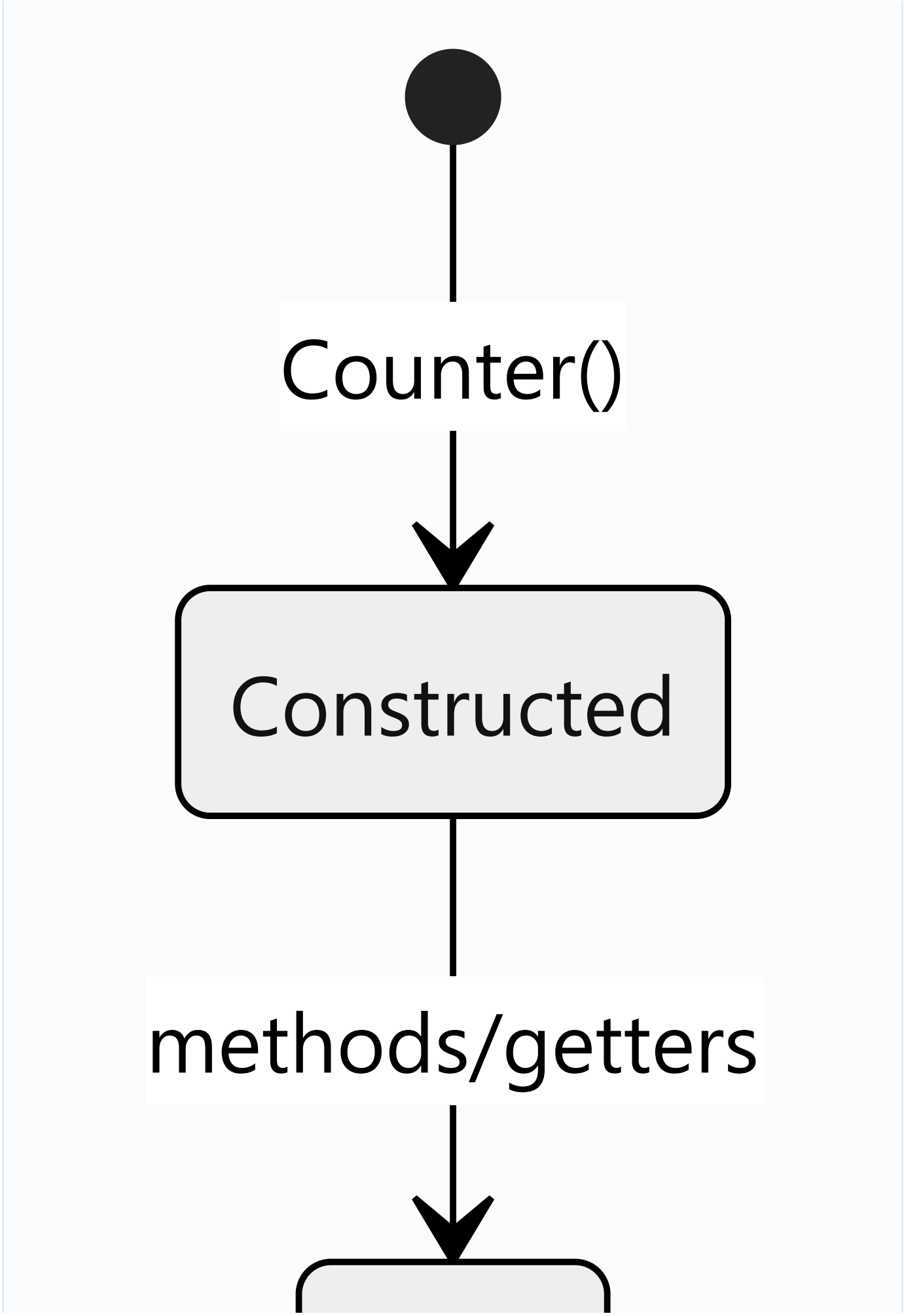
- Object layout is compact; the VM may inline/optimize field access. Identity hash is assigned lazily on first `identityHashCode` / `hashCode` use (for default identity).

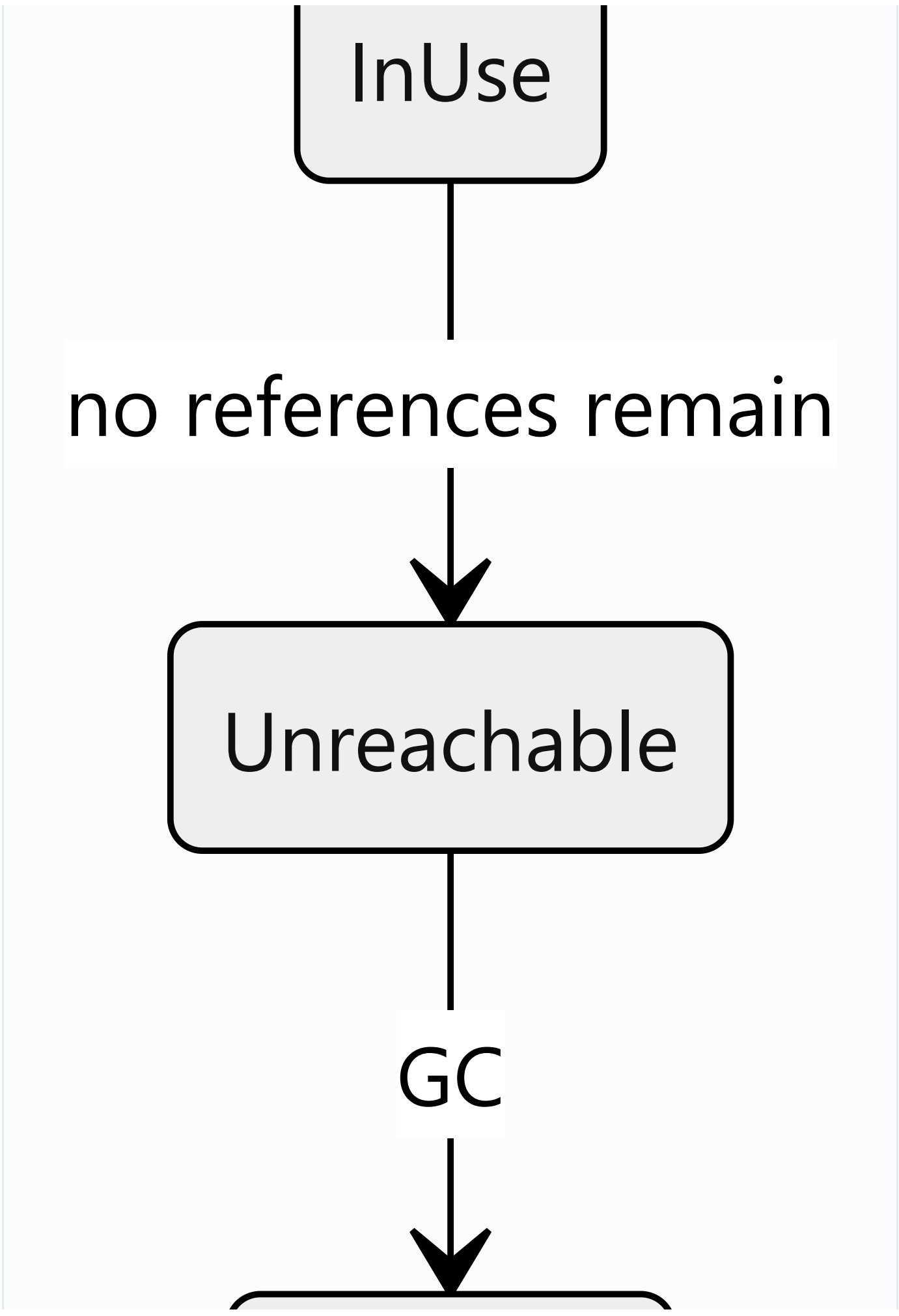
Examples

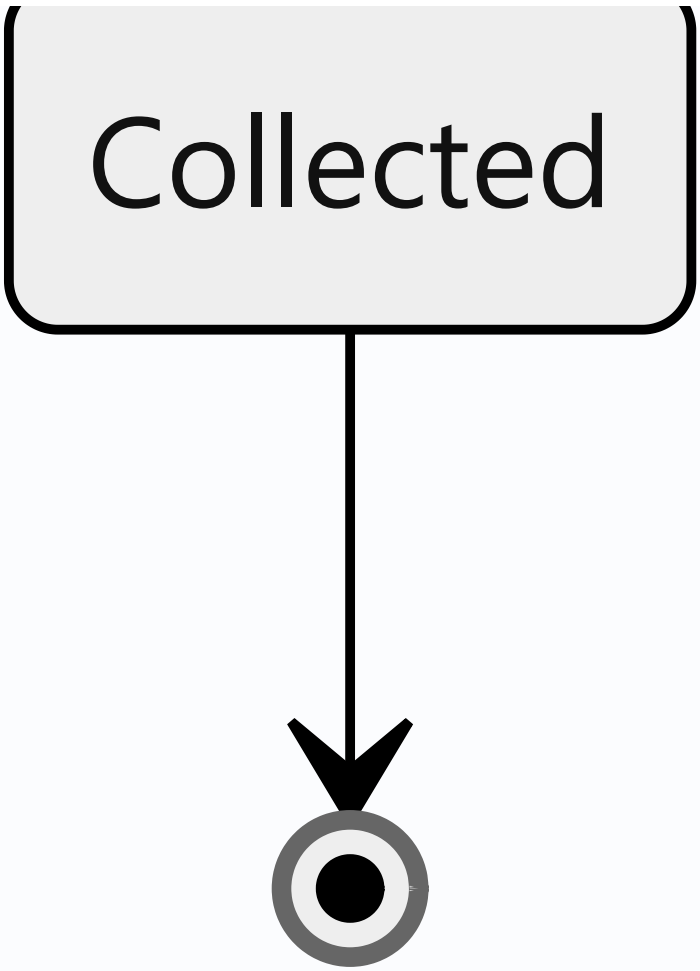
```
class Counter {
  int _count; // instance field
  static int instances = 0; // shared class field

  Counter({int start = 0}) : _count = start {
    instances++; // track how many created
  }

  int get value => _count; // read-only getter
  bool get isZero => _count == 0; // computed getter

  void increment() => _count++;
  void reset() => _count = 0;

  @override
  String toString() => 'Counter($_count)';
}

void main() {
  final a = Counter();
  final b = Counter(start: 5);
  a.increment();
  a.increment();

  print(a.value);            // 2
  print(b.value);            // 5
  print(a.isZero);           // false
  print(Counter.instances);  // 2 — static, shared across instances
  print(a);                  // Counter(2) — toString()

  // reference semantics:
  final c = a;   // c and a point to the SAME object
  c.increment();
```

```
  print(a.value); // 3 — mutating via c is visible via a
}
```

Diagrams


Counter()
Constructed
methods/getters


InUse
no references remain
Unreachable
GC



Common Mistakes

Mistake	Why	Fix
Public mutable fields	Breaks encapsulation/invariants	Private field + getter (see 02_encapsulation.md)
Using `static` for per-instance state	All instances share it (bug)	Use an instance field
Assuming value semantics	Objects are references	Copy explicitly when you need independence
Forgetting `toString()`	Prints `Instance of 'X'`	Override it for debugging
`static` as a global dumping ground	Hidden global state	Prefer instances + DI

Best Practices

- Keep fields private; expose intent via getters/methods.
- Use `static` only for truly class-level data/utilities (constants, counters, factories).
- Override `toString()` for debuggability.
- Keep classes cohesive (one responsibility — bridge to SRP, Module 04).

Performance

- Object allocation is cheap but not free; avoid churning short-lived objects in hot paths (02 · memory_and_gc).
- Getters are as fast as field access after optimization.

Advantages / Disadvantages

- **+** Bundles state+behavior, reusable, testable, models the domain.
- **−** Poorly designed classes (God objects, public mutable state) become maintenance hazards.

Interview Questions

1. 🟢 **Class vs object?** — A class is the blueprint (type); an object is a concrete instance of it on the heap.
2. 🟢 **Instance vs `static` member?** — Instance members exist per object; `static` members exist once per class and are accessed via the class.
3. 🟡 **What does `this` refer to and when is it required?** — The current instance; required to disambiguate a field from a same-named parameter or to pass/return the instance.
4. 🟡 **Does Dart use value or reference semantics for objects?** — Reference: a variable holds a reference; assigning/passing shares the same object (primitives-like `int` behave value-like but are still objects).
5. 🟡 **What is the uniform access principle?** — Callers can't tell a getter from a field (`c.value`), so you can switch a field to a computed getter without changing call sites.
6. 🔴 **When is an object eligible for GC?** — When it becomes unreachable from all roots (02 · memory_and_gc).
7. 🔴 **Where do `static` fields live vs instance fields?** — `static` fields: once per class in class metadata; instance fields: in each heap object.

Senior Engineer Tips

- Default fields to `private + final`; open them up only when a real need appears.
- Treat `static` mutable state as a smell — it's a global; prefer instances managed by DI.
- Model behavior *with* data: if a class is all getters/setters and logic lives elsewhere, you have an anemic model (bridge to DDD, Module 46).

Architect Perspective

Class design sets the grain of your system. Cohesive classes with clear responsibilities and private state compose into maintainable modules; God classes and public mutable state metastasize into coupling. This is the substrate on which SOLID, patterns, and clean architecture build.

Summary

- Class = blueprint; object = heap instance (reference semantics).
- Instance members are per-object; `static` members are per-class.
- Keep fields private, expose getters/methods, override `toString()`, and mind the object lifecycle.

Revision Notes

- Class = type/blueprint; object = instance on heap; variables hold references.
- Instance field = per object; `static` = one per class (`Class.member`).
- `this` disambiguates/returns instance; getters give uniform access.
- Object collectible when unreachable; override `toString()`.

Practice Questions

1. Why does mutating via `c = a; c.increment()` change `a`?
2. When is `static` correct and when is it a bug?
3. What's the difference between a getter and a public field to a caller?

Coding Questions

1. Build a `Stopwatch`-like class with instance state and a `static` count of created instances.
2. Add a computed getter (`elapsedSeconds`) and prove uniform access by swapping a field to a getter.
3. Demonstrate reference semantics: two variables mutating one object.

Mini Project

Domain object toolkit (pure Dart): Model a `Playlist` (private track list, add/remove methods, computed `duration` getter, `static` count of playlists, `toString`). Enforce that tracks can't be mutated externally. Acceptance: no public mutable fields; static used only for the counter; `dart analyze` clean; tests cover behavior + reference semantics.

Encapsulation (Privacy, Getters/Setters, Invariants)

> Encapsulation means hiding internal state behind a controlled interface so an object can *guarantee its own invariants* — the balance can never go negative because only the object can change it.

Introduction

Encapsulation bundles data with the methods that operate on it and **restricts direct access** to the internals. In Dart, privacy is library-scoped via a leading underscore (`_`). This file covers `_` privacy, getters/setters, validation, and the design goal that ties them together: protecting **invariants**.

Why this concept exists

If any code can mutate an object's fields directly, no one can guarantee the object is ever in a valid state — a `BankAccount._balance` set to `-9999` from anywhere is a bug waiting to happen. Encapsulation makes the object the *sole guardian* of its rules, so invalid states become impossible by construction.

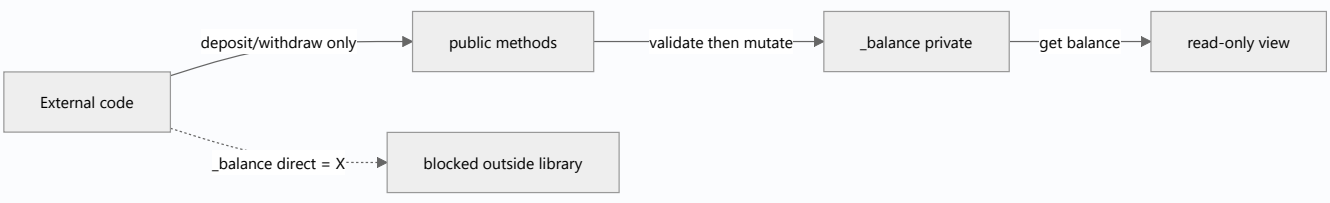
Real-world analogy

An ATM encapsulates the vault. You can't reach into the cash drawer (`_balance`); you interact through a controlled panel (`deposit` , `withdraw`) that enforces rules (sufficient funds, positive amounts). The vault's integrity is guaranteed because only the machine touches the cash.

Problem Statement

Model a `BankAccount` whose balance can only change through `deposit` / `withdraw` , never go negative, and be readable but not writable from outside. You'll use a private field, a getter, and validating methods.

Internal Working



- **Privacy:** `_balance` is visible only within its **library** (file, or files joined by `part`) — not per class. Same-file code *can* see another class's `_members`.
- **Getter without setter** → read-only public access.
- **Setter** → controlled write with validation (`set price(v) { if (v<0) throw; _price=v; }`).
- **Invariant:** a rule that must always hold (balance ≥ 0). Methods enforce it; construction validates it (asserts/initializer list — see 02 · constructors).

Memory Representation

Not applicable beyond normal object layout — privacy is a compile-time visibility rule, not a runtime tag (01_classes_and_objects.md).

Compiler Behavior

- Accessing another **library's** `_member` is a compile error (undefined name).
- Getters/setters are checked like fields (uniform access); a getter-only field can't be assigned.

Runtime Behavior

- Setters run their validation logic on each assignment; a rejected value throws where you decide.
- No runtime privacy enforcement is needed — the compiler already prevented cross-library access.

Flutter Engine Behavior

Not applicable. (Flutter widgets encapsulate config as `final` fields; `State` encapsulates mutable UI state behind `setState`.)

Dart VM Behavior

Not applicable — privacy is erased to normal member access at runtime.

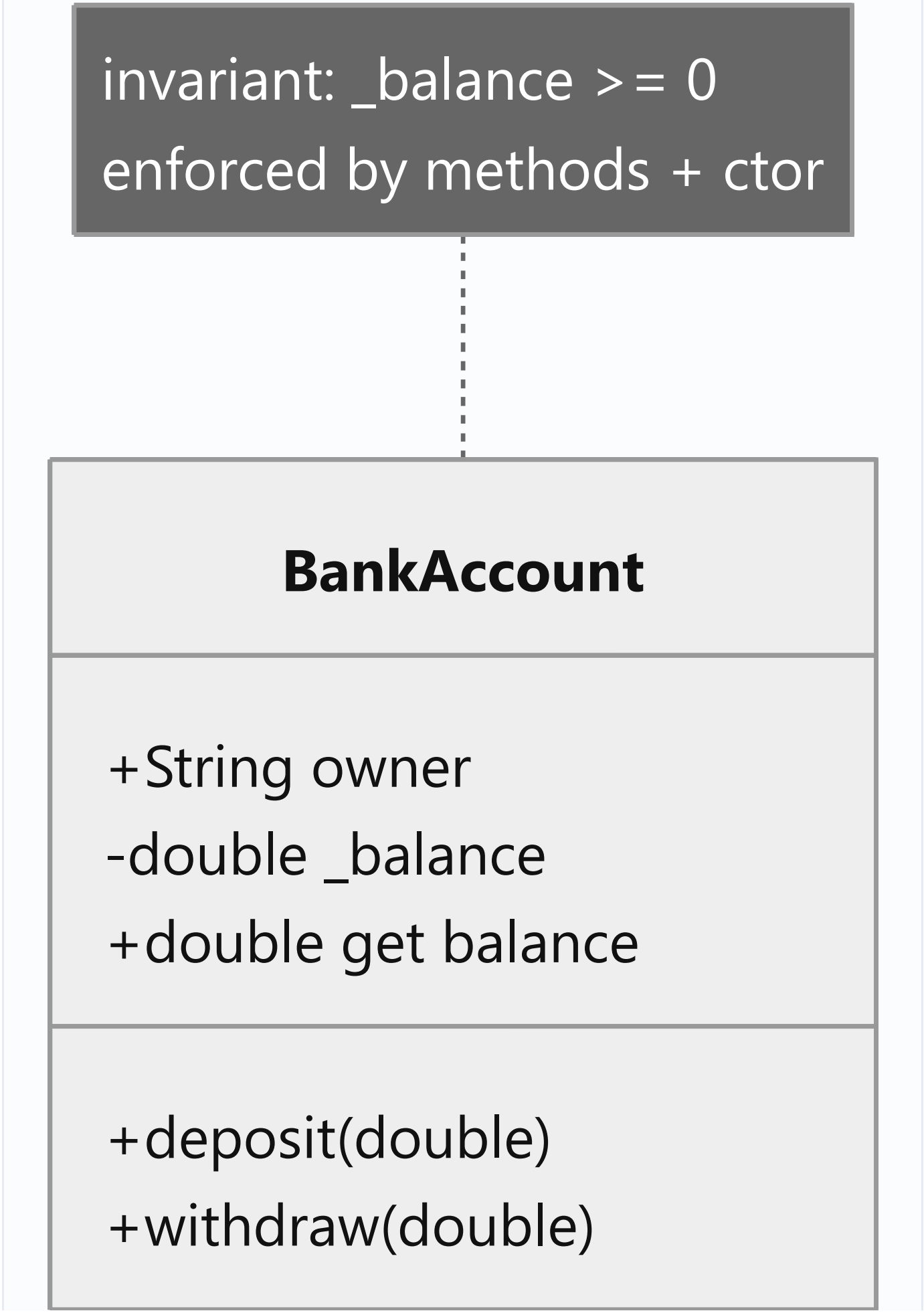
Examples

```
class BankAccount {
  final String owner;
  double _balance; // private: guarded state

  BankAccount({required this.owner, double opening = 0})
```

```
      : assert(opening >= 0, 'opening cannot be negative'),
        _balance = opening;

  double get balance => _balance; // read-only

  void deposit(double amount) {
    if (amount <= 0) {
      throw ArgumentError.value(amount, 'amount', 'must be positive');
    }
    _balance += amount; // invariant preserved
  }

  void withdraw(double amount) {
    if (amount <= 0) {
      throw ArgumentError.value(amount, 'amount', 'must be positive');
    }
    if (amount > _balance) {
      throw StateError('insufficient funds'); // invariant: never negative
    }
    _balance -= amount;
  }
}

// controlled setter with validation
class Product {
  double _price;
  Product(this._price);
  double get price => _price;
  set price(double v) {
    if (v < 0) throw ArgumentError('price cannot be negative');
    _price = v;
  }
}

void main() {
  final acc = BankAccount(owner: 'Ada', opening: 1000);
  acc.deposit(500);
  acc.withdraw(200);
  print(acc.balance); // 1300
  // acc._balance = -1; // COMPILE ERROR outside this library
```

```
  // acc.balance = 5000; // COMPILE ERROR: no setter (read-only)


  final p = Product(10);
  p.price = 20; // ok
  try {
    p.price = -5; // throws
  } on ArgumentError catch (e) {
    print(e.message); // price cannot be negative
  }
}
```

Diagrams


invariant: _balance >= 0
enforced by methods + ctor
BankAccount
+String owner
-double _balance
+double get balance
+deposit(double)
+withdraw(double)

Common Mistakes

Mistake	Why	Fix
Public mutable field	Anyone breaks invariants	Private field + getter/methods
Getter + unguarded setter that just assigns	No protection gained	Validate in the setter or drop it
Assuming `_` is class-private	It's *library*-private	Split files if you need stricter boundaries
Exposing internal mutable collection	Callers mutate internals	Return `List.unmodifiable` (see immutability)
Anemic model (all getters/setters, logic elsewhere)	Invariants unprotected	Put behavior on the object

Best Practices

- Default to **private fields**; expose read-only getters and intention-revealing methods.
- Validate at **construction** (asserts/initializer list) and at every mutation point.
- Return **unmodifiable views** of internal collections.
- Keep the public surface minimal — "make illegal states unrepresentable."

Performance

- Negligible; getters/validation are cheap. Encapsulation is a design/correctness concern, not a perf one.

Advantages / Disadvantages

- **+** Guaranteed invariants, safer refactoring (change internals freely), clearer API, better testability.
- **−** Slightly more boilerplate (getters/methods); over-encapsulation can add ceremony for simple data.

Interview Questions

1. 🟢 **What is encapsulation?** — Bundling state with behavior and hiding internals behind a controlled interface so the object protects its own invariants.
2. 🟢 **How is privacy implemented in Dart?** — A leading underscore makes a member **library-private** (per file/ `part`), not class-private; Dart has no `public` / `private` / `protected` keywords.

3. 🟡 **Getter vs public field — when prefer a getter?** — When the value is computed/derived, when you want read-only access, or to add validation/logic without changing call sites (uniform access).
4. 🟡 **What is an invariant and how do you protect it?** — A rule that must always hold (e.g., balance ≥ 0); protect it by validating at construction and confining all mutations to methods that enforce it.
5. 🟡 **Why return `List.unmodifiable` from a getter?** — To prevent callers from mutating the object's internal collection and breaking encapsulation.
6. 🔴 **Since `_` is library-scoped, how do you get stricter boundaries?** — Put the class in its own library/file (and package `src/`) so no same-file code can reach its privates (02 · libraries).
7. 🔴 **What is an anemic domain model and why is it discouraged?** — A class that's just data (getters/setters) with logic elsewhere; it can't protect invariants and scatters behavior — prefer rich models (DDD, Module 46).

Senior Engineer Tips

- Design the **public API first** (what callers should do), then hide everything else.
- Prefer construction-time validation so an object is *never* born invalid.
- Expose immutable/unmodifiable views; mutation should go through named, meaningful operations.

Architect Perspective

Encapsulation is the local enforcement of invariants that, aggregated, keeps a large system consistent. It's the class-level analogue of module boundaries (02 · libraries) and the foundation of DDD aggregates that guard consistency rules (Module 46). Weak encapsulation is a leading cause of "spooky action at a distance" bugs at scale.

Summary

- Encapsulation hides internals behind a controlled interface so objects guard their invariants.
- Dart privacy is library-scoped (`_`); use getters for read-only/computed access and setters (validated) sparingly.
- Validate at construction and at every mutation; expose unmodifiable views.

Revision Notes

- `_member` = library-private (file/`part`), not class-private.
- Getter-only = read-only; validating setter or none.
- Invariant protected by ctor validation + method-only mutation.

- Return `List.unmodifiable`; keep public surface minimal; avoid anemic models.

Practice Questions

1. Why can't external code set `acc.balance = 5000`?
2. When is a validated setter worth it vs a method like `changePrice`?
3. Why is exposing a raw internal `List` a leak of encapsulation?

Coding Questions

1. Model a `TemperatureSensor` with a private reading, a clamped setter, and a read-only `celsius` getter.
2. Build a `Cart` that exposes items only as an unmodifiable list and mutates via `add` / `remove`.
3. Add construction-time validation so a `Rectangle` can never have negative sides.

Mini Project

Guarded account domain (pure Dart): Implement `BankAccount` + `SavingsAccount` (min-balance rule) with fully encapsulated balances, validated construction, custom exceptions, and read-only exposure. Add tests proving invariants can't be violated (negative opening, over-withdraw, direct field access impossible). Acceptance: no public mutable state; invariants enforced everywhere; `dart analyze` clean.

Inheritance (`extends`, `super`, `@override`)

> Inheritance models an **is-a** relationship: a subclass reuses and specializes a superclass's implementation — powerful, but easy to overuse where composition fits better.

Introduction

Inheritance (`class B extends A`) lets `B` reuse `A`'s fields/methods and add or override behavior. This file covers `extends`, calling `super`, `@override`, constructor chaining with `super(...)`, abstract base classes, and — crucially — *when inheritance is the wrong tool*.

Why this concept exists

Many types share behavior (all accounts can deposit; all shapes have a position). Inheritance lets you write that once in a base class and specialize per subtype, avoiding duplication and enabling polymorphism. Dart supports **single inheritance** (one superclass) to keep dispatch unambiguous.

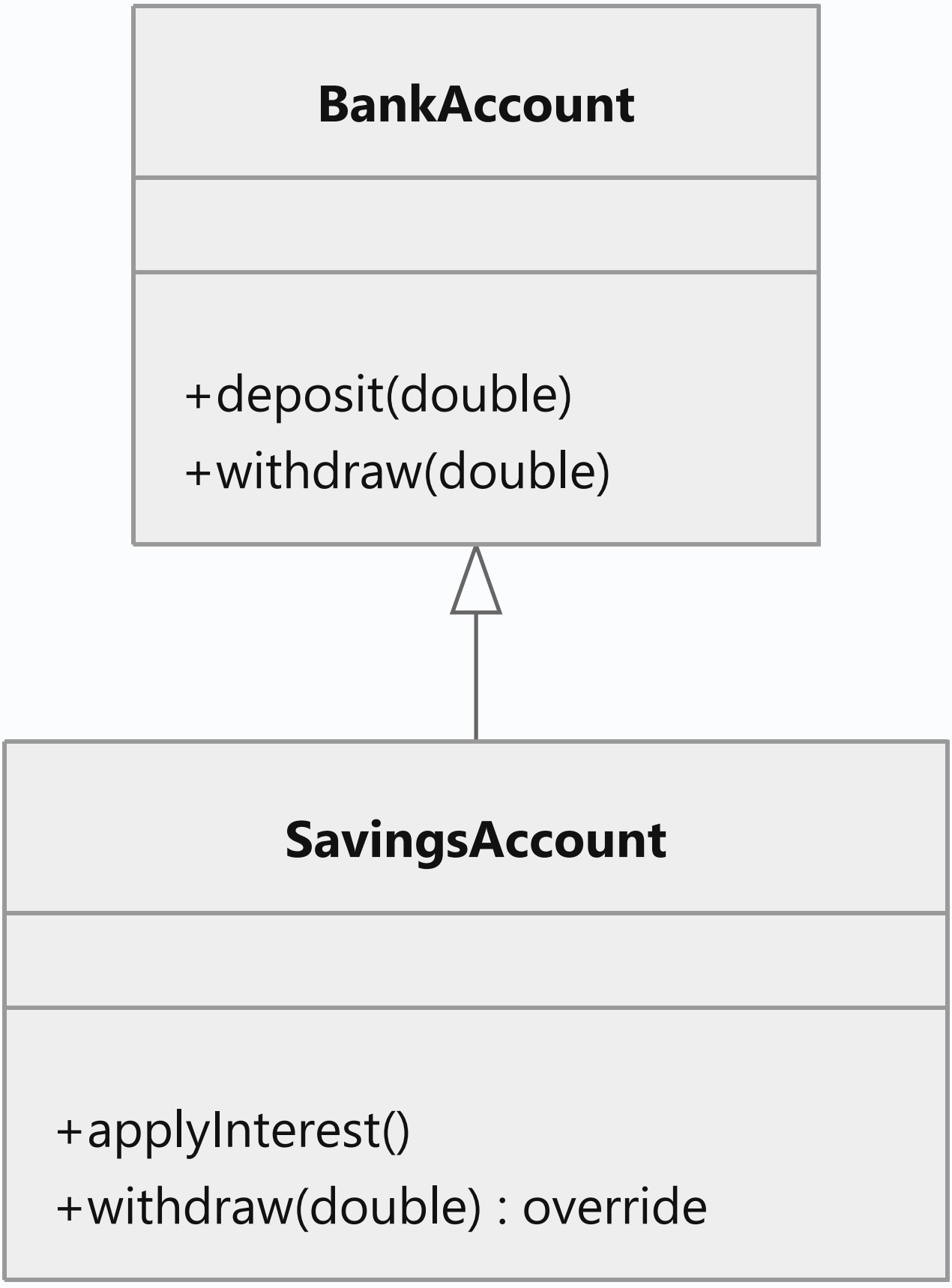
Real-world analogy

Inheritance is a **family trait**: a `SavingsAccount` inherits everything a `BankAccount` can do (deposit, withdraw) and adds its own (interest), the way a child inherits general human abilities and adds personal skills. But "is-a" must be *true*: a `Stack` is not really a `List` even if it could reuse one — that's a classic inheritance misuse.

Problem Statement

Model `BankAccount` (base) and `SavingsAccount` (adds interest, tightens `withdraw` with a minimum balance, reuses base validation via `super`). You'll use `extends`, `super(...)` in the constructor, `@override`, and `super.method()`.

Internal Working



- `class B extends A` — `B` gets `A`'s non-private members and can add/override.

- **Constructor chaining:** `B`'s constructor must initialize `A` via `super(...)` (or `super.field` forwarding). The superclass constructor runs before `B`'s body.
- `@override` annotates a method that replaces the superclass version (analyzer-checked).
- `super.method()` calls the superclass implementation — to *extend* rather than fully replace behavior.
- **Abstract base class:** `abstract class A` can't be instantiated; defines shared code + abstract methods subclasses must implement.

Memory Representation

- A subclass instance is one object containing **all** fields (inherited + own). The method table links to inherited and overridden implementations for dynamic dispatch (04_polymorphism.md).

Compiler Behavior

- Missing `super(...)` when the superclass has no default constructor is a compile error.
- `@override` mismatches (wrong signature, no such super method) are flagged.
- Overriding must respect the supertype's signature (parameters may be covariantly narrowed only with `covariant`).

Runtime Behavior

- Construction order: superclass initializer list/constructor → subclass initializer list → superclass body? Precisely: field initializers + `super` call resolve, base constructor body runs, then the subclass constructor body. (Initializer lists run before bodies.)
- Method calls dispatch to the **runtime type's** override; `super.m()` statically targets the superclass version.

Flutter Engine Behavior

Not applicable, but Flutter uses inheritance pervasively: `StatelessWidget` / `StatefulWidget` extend `Widget`; your widgets extend those; `RenderBox` extends `RenderObject`.

Dart VM Behavior

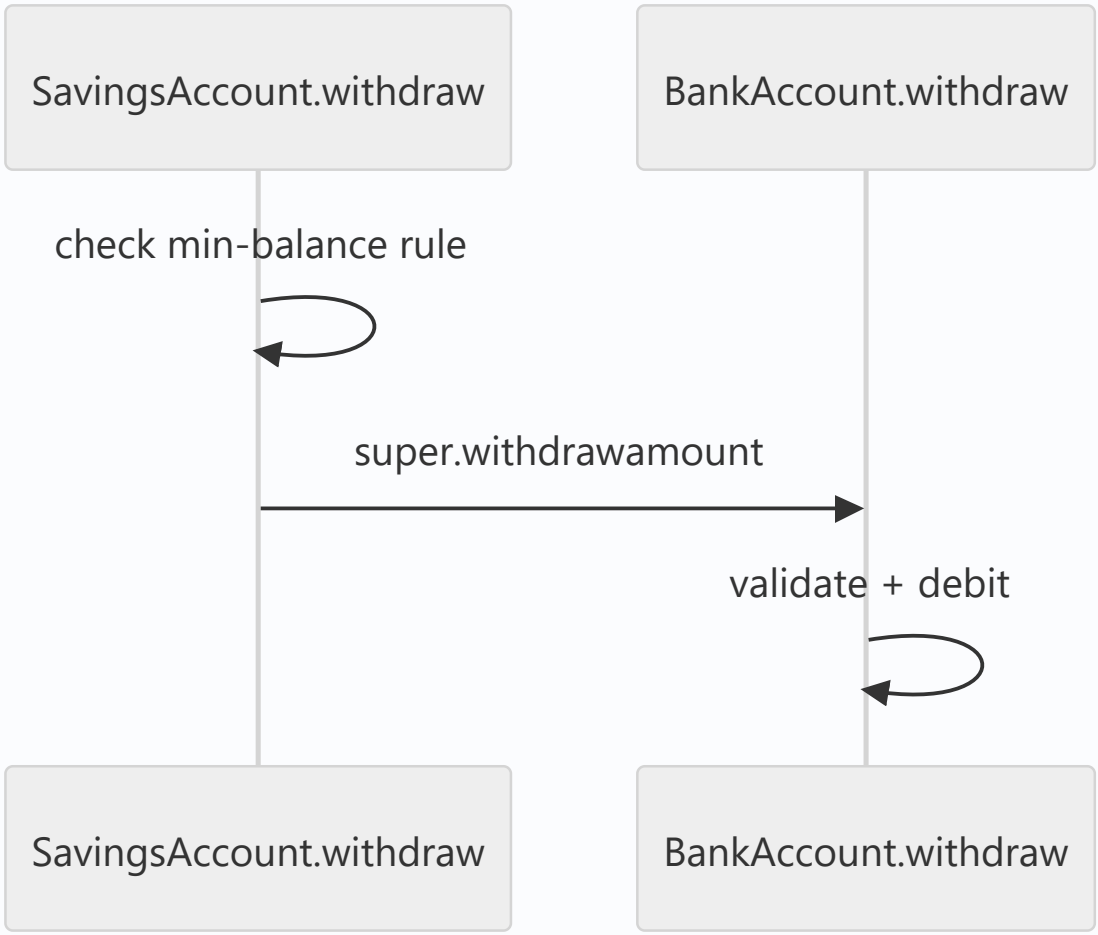
- Virtual dispatch via method tables; monomorphic call sites are optimized/inlined in AOT.

Examples

```
class BankAccount {
  final String owner;
  double _balance;
  BankAccount({required this.owner, double opening = 0}) : _balance = opening;

  double get balance => _balance;

  void deposit(double amount) {
    if (amount <= 0) throw ArgumentError('positive only');
    _balance += amount;
  }

  void withdraw(double amount) {
    if (amount <= 0) throw ArgumentError('positive only');
    if (amount > _balance) throw StateError('insufficient funds');
    _balance -= amount;
  }

  @override
  String toString() => '$runtimeType($owner): ${_balance.toStringAsFixed(2)}';
}

class SavingsAccount extends BankAccount {
  static const double minBalance = 100;
  final double annualRate;

  SavingsAccount({
    required super.owner,      // forward to BankAccount.owner
    super.opening,
    required this.annualRate,
  });

  void applyMonthlyInterest() => deposit(balance * annualRate / 12); // reuse deposit

  @override
  void withdraw(double amount) {
    if (balance - amount < minBalance) {
      throw StateError('must keep min balance of $minBalance');
```

```
    }
    super.withdraw(amount); // extend: run base validation + debit
  }
}

void main() {
  final s = SavingsAccount(owner: 'Ada', opening: 1000, annualRate: 0.06);
  s.deposit(500);
  s.applyMonthlyInterest(); // +1500*0.06/12 = 7.5
  print(s); // SavingsAccount(Ada): 1507.50

  try {
    s.withdraw(1500); // would drop below min
  } on StateError catch (e) {
    print(e.message); // must keep min balance of 100.0
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Inheriting for code reuse without is-a	Fragile hierarchies, LSP violations	Use composition (06_composition_and_relationships.md)
Forgetting `super(...)`	Compile error / uninitialized base	Forward constructor args
Overriding without calling `super` when you meant to extend	Loses base behavior	Call `super.method()`
Deep inheritance chains	Rigid, hard to change	Flatten; prefer composition/mixins
Overriding + changing the contract	Breaks polymorphism/LSP	Keep the supertype's contract (Module 04)

Best Practices

- Use inheritance only for genuine **is-a** with a stable base contract.
- Prefer shallow hierarchies; favor **composition** and **mixins** for reuse across unrelated types.

- Call `super.method()` when extending; document overrides.
- Make base classes designed-for-inheritance (or `final`/sealed to forbid it) — don't leave it ambiguous.

Performance

- Virtual dispatch is cheap; deep hierarchies don't cost much at runtime but hurt *maintainability*.

Advantages / Disadvantages

- **+** Code reuse, polymorphism, clear taxonomy when is-a truly holds.
- **−** Tight coupling to the base, fragile-base-class problem, single-inheritance limit, easy to misuse for reuse.

Interview Questions

1. 🟢 **What does `extends` give a subclass?** — The superclass's non-private fields/methods, which it can use, add to, or override.
2. 🟢 **`extends` vs `with` vs `implements`?** — `extends`: inherit one superclass's implementation (`super` available). `with`: mix in reusable implementation(s). `implements`: adopt only the contract, reimplementing everything.
3. 🟡 **What is `super` for?** — Calling the superclass constructor (`super(...)`) and superclass method implementations (`super.method()`) to reuse/extend behavior.
4. 🟡 **Describe construction order in a subclass.** — Initializer lists resolve and the superclass constructor runs first, then the subclass constructor body; `this` isn't available in initializer lists.
5. 🟡 **Why prefer composition over inheritance?** — Inheritance couples you to the base's implementation (fragile base class) and forces an is-a; composition is more flexible, swappable, and testable.
6. 🔴 **What is the fragile base class problem?** — Changes to a superclass can unexpectedly break subclasses that depend on its internal behavior — a maintenance hazard of deep inheritance.
7. 🔴 **How do you forbid or design for inheritance?** — Mark classes `final`/`sealed`/`base` (Dart 3 class modifiers) to control subclassing, or document/`@protected` the extension points.

Senior Engineer Tips

- Ask "is-a or has-a?" before `extends`. If you're inheriting to reuse a method, you probably want composition.

- Use Dart 3 **class modifiers** (`final` , `sealed` , `base` , `interface`) to make your inheritance intent explicit and enforceable.
- When you must inherit, keep the base's protected surface small and stable.

Architect Perspective

Inheritance choices shape coupling. Shallow, intentional hierarchies (or sealed type families) with composition for reuse keep systems flexible; sprawling inheritance trees ossify them. This directly feeds the Liskov and Open/Closed principles (Module 04) and pattern choices like Strategy/Decorator over subclass explosions (Module 05).

Summary

- `extends` gives is-a reuse + specialization; chain constructors via `super(...)` , extend via `super.method()` , mark overrides with `@override` .
- Prefer shallow hierarchies and composition; reserve inheritance for true is-a with stable contracts.
- Use Dart 3 class modifiers to make inheritance intent explicit.

Revision Notes

- `extends` = single inheritance + `super` ; `@override` to specialize.
- Constructor: `super(...)` / `super.field` ; base runs before subclass body.
- `super.method()` extends base behavior; keep the contract (LSP).
- Prefer composition; avoid deep chains + fragile base class; use class modifiers.

Practice Questions

1. Why does `SavingsAccount.withdraw` call `super.withdraw` ?
2. When is `extends` the wrong choice and what do you use instead?
3. Explain the construction order for a two-level hierarchy.

Coding Questions

1. Build a `Vehicle → ElectricCar` hierarchy where `ElectricCar` overrides `refuel` (charge) and reuses `move` via `super` .
2. Create an abstract `Shape` with `area()` and concrete `Circle` / `Rectangle` subclasses.
3. Demonstrate the fragile base class problem with a base change breaking a subclass, then fix it via composition.

Mini Project

Account hierarchy (pure Dart): Implement `BankAccount`, `SavingsAccount` (min balance + interest), and `CheckingAccount` (overdraft limit), each overriding `withdraw` and reusing base validation via `super`. Add `toString` via `runtimeType`. Acceptance: correct construction chaining; overrides preserve the base contract; tests cover each subtype; `dart analyze` clean.

Polymorphism (Dynamic Dispatch, Overriding, Liskov)

> Polymorphism lets one reference of a base type invoke the *correct subtype behavior* at runtime — write code against an abstraction, and each concrete type does the right thing.

Introduction

Polymorphism ("many forms") means a variable of a supertype can hold any subtype, and calling a method runs the **subtype's** implementation via **dynamic dispatch**. This file covers method overriding, runtime dispatch, treating collections uniformly, and the **Liskov Substitution Principle** (subtypes must be usable wherever the base is).

Why this concept exists

Extensibility. Instead of `if (type == circle) ... else if (type == square) ...` scattered everywhere, you call `shape.area()` and each shape computes its own. Adding a new shape doesn't touch existing code — this is the engine of the Open/Closed Principle (Module 04) and most design patterns.

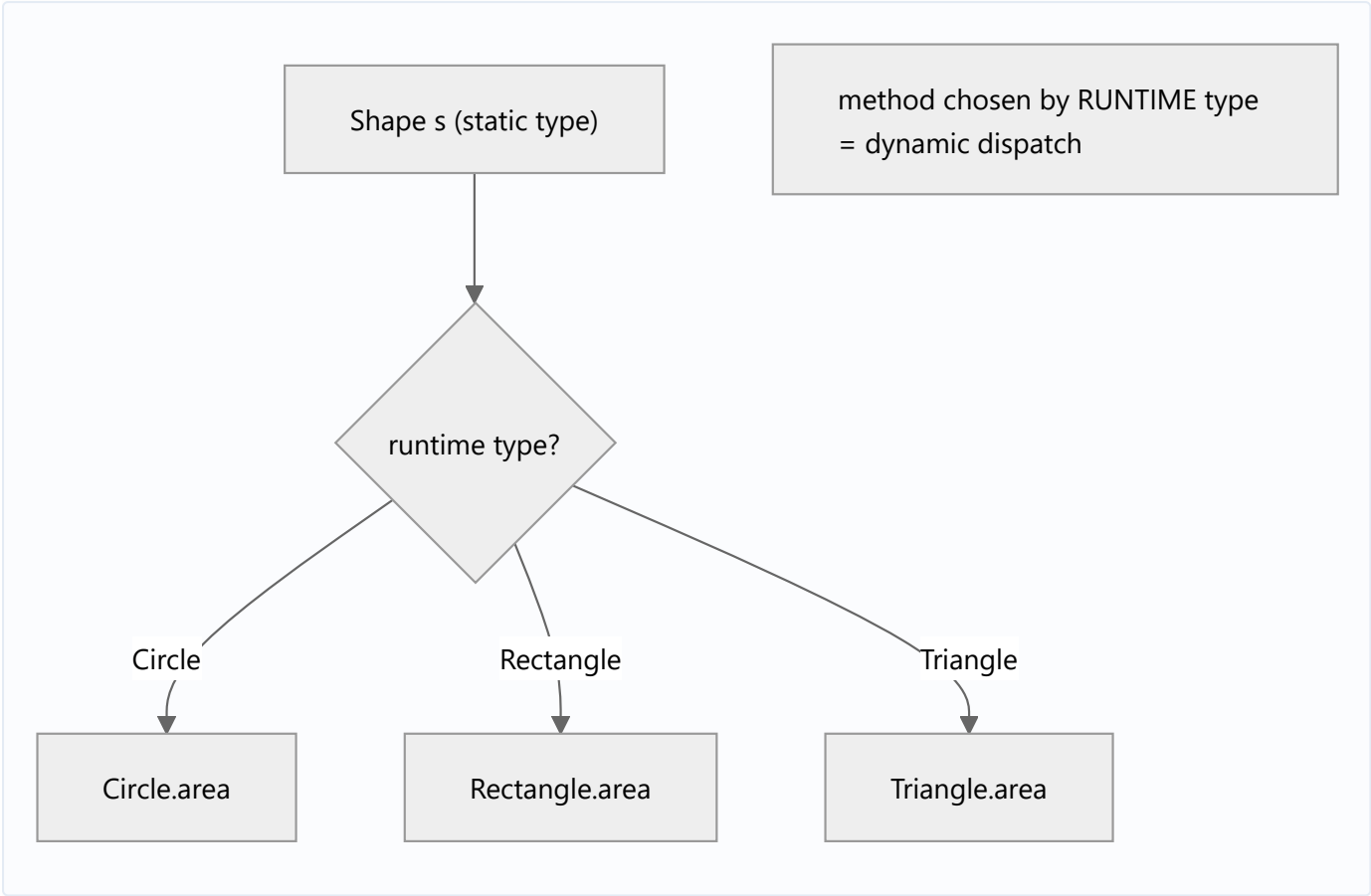
Real-world analogy

A universal **remote's** "play" button works on a TV, a soundbar, and a DVD player — same command, device-specific behavior. You (the caller) press "play" (`area()`) without caring which device (subtype) it is; each responds correctly.

Problem Statement

Compute the total area of a mixed list of shapes without type checks, and let a payment processor charge any `PaymentMethod` uniformly. Adding a new shape/method must require **zero** changes to the totaling/processing code. You'll use overriding + dynamic dispatch.

Internal Working



- A supertype reference (`Shape s`) can point to any subtype instance.
- Calling `s.area()` uses **dynamic (virtual) dispatch**: the runtime looks up the method on the object's actual class.
- **Overriding** provides the subtype-specific behavior (`@override double area()`).
- **Liskov Substitution Principle (LSP):** any subtype must honor the base's contract so it can substitute the base without surprising callers (no strengthening preconditions, no weakening postconditions, no throwing where the base wouldn't).

Memory Representation

- Each object carries its class identity; the method table maps method names to the concrete implementations for that class, enabling O(1) virtual dispatch (01_classes_and_objects.md).

Compiler Behavior

- The compiler checks the call against the **static type** (`Shape` must declare `area()`), but the *implementation* is chosen at runtime.
- Overrides are validated (`@override`) against the supertype signature.

Runtime Behavior

- Virtual dispatch resolves to the actual class's method. `is` / `as` let you narrow when you truly need subtype-specific API (but heavy `is` chains often signal a missing polymorphic method).

Flutter Engine Behavior

Not applicable, but the framework is polymorphism in action: it calls `build()` / `paint()` / `layout()` on your subclasses through base references without knowing your concrete types.

Dart VM Behavior

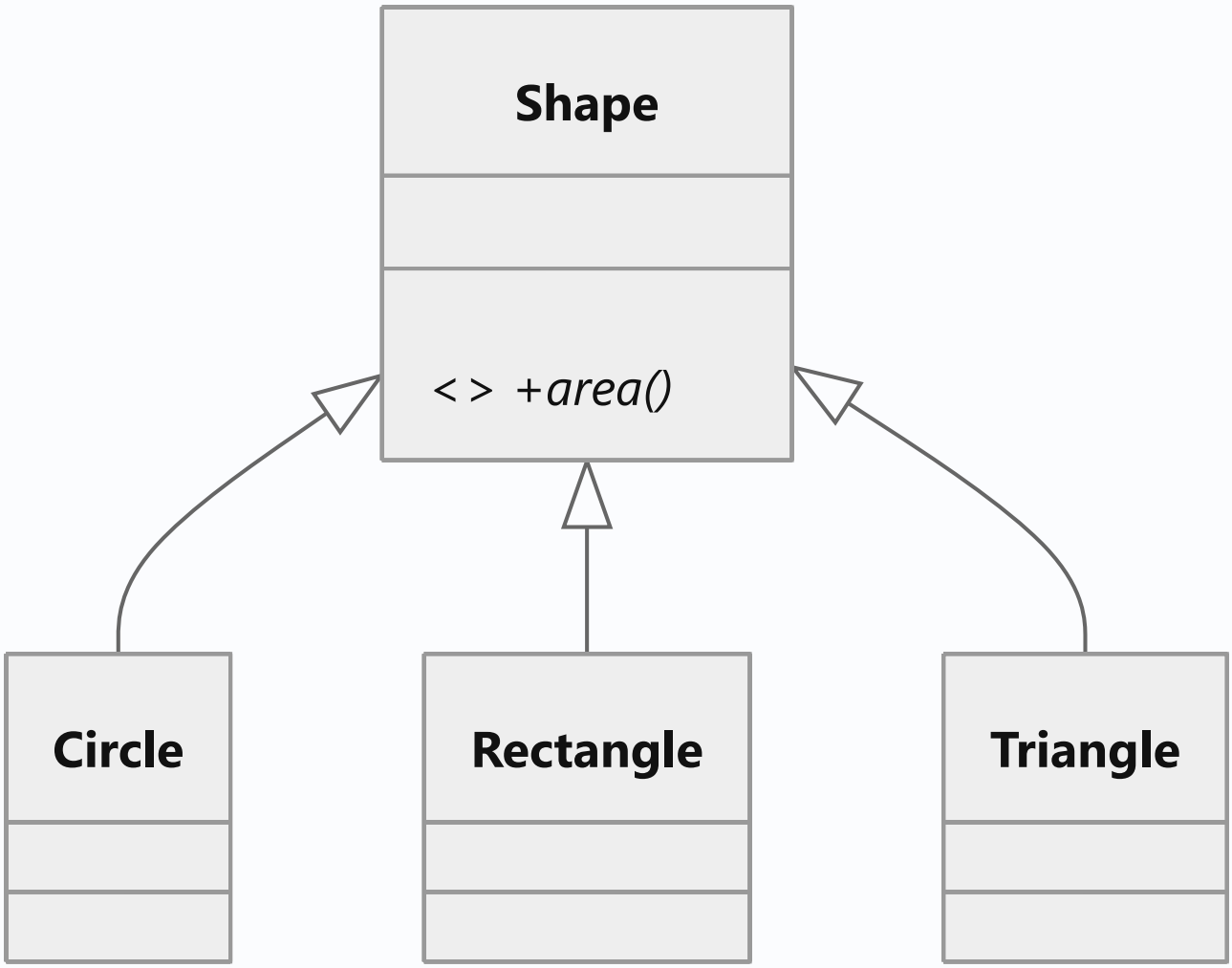
- Monomorphic/polymorphic inline caches optimize hot virtual call sites; AOT devirtualizes where the concrete type is provable.

Examples

```
abstract class Shape {
  double area(); // contract
}

class Circle extends Shape {
  final double r;
  Circle(this.r);
  @override
  double area() => 3.141592653589793 * r * r;
}

class Rectangle extends Shape {
  final double w, h;
  Rectangle(this.w, this.h);
  @override
  double area() => w * h;
}

class Triangle extends Shape {
  final double b, h;
  Triangle(this.b, this.h);
  @override
  double area() => 0.5 * b * h;
}
```

```
// Works for ANY Shape — no type checks, open for new subtypes:
double totalArea(List<Shape> shapes) =>
    shapes.fold(0.0, (sum, s) => sum + s.area());

void main() {
  final shapes = <Shape>[Circle(1), Rectangle(2, 3), Triangle(4, 5)];
  print(totalArea(shapes)); // 3.14... + 6 + 10 = 19.14...

  // dynamic dispatch: same call, different behavior
  for (final s in shapes) {
    print('${s.runtimeType}: ${s.area().toStringAsFixed(2)}');
  }
  // Adding a Pentagon subtype needs ZERO changes to totalArea.
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`if (s is Circle) ... else if (s is Rectangle)`	Not polymorphic; must edit for each new type	Put behavior on the type (override)
Subtype throwing where base doesn't	Breaks LSP; callers surprised	Honor the base contract
Overriding but changing semantics	Callers relying on base behavior break	Keep pre/postconditions compatible
Downcasting everywhere (`as`)	Fragile, defeats abstraction	Add a polymorphic method to the base
Calling a subtype-only method via base ref	Not in the contract	Narrow with `is` only when justified

Best Practices

- Program to the **abstraction** (base type/interface), not concrete types.
- Add polymorphic methods instead of type-switch chains.
- Uphold **LSP**: subtypes are drop-in replacements; don't weaken guarantees or add surprising preconditions.
- Use sealed classes + exhaustive `switch` when a *closed* set of variants needs different handling (05_abstraction_and_interfaces.md).

Performance

- Virtual dispatch is fast; inline caches + AOT devirtualization minimize overhead. Don't micro-optimize away polymorphism for negligible gains.

Advantages / Disadvantages

- **+** Extensible (Open/Closed), removes type-switch sprawl, decouples callers from concretes.
- **−** Behavior spread across subclasses can be harder to trace; misuse (LSP violations) causes subtle bugs.

Interview Questions

1. 🟢 **What is polymorphism?** — The ability to use a supertype reference to invoke the correct subtype behavior, resolved at runtime via dynamic dispatch.
2. 🟢 **What is method overriding?** — A subclass providing its own implementation of a superclass method (`@override`), selected at runtime.
3. 🟡 **Static type vs runtime type in dispatch?** — The compiler checks the call against the static (declared) type; the actual method runs based on the object's runtime type.
4. 🟡 **What is the Liskov Substitution Principle?** — Subtypes must be substitutable for their base without breaking correctness: don't strengthen preconditions, weaken postconditions, or violate the base's invariants/contract.
5. 🟡 **Why are `is` / `as` chains a smell?** — They centralize type knowledge that should be distributed as polymorphic methods; each new subtype forces edits (violates Open/Closed).
6. 🔴 **How does the VM make virtual dispatch fast?** — Method tables + inline caches for hot call sites; AOT devirtualizes when the concrete type is known.
7. 🔴 **Polymorphism vs sealed-class `switch` — when each?** — Polymorphism for open extension (new subtypes without editing callers); sealed `switch` for a closed, known set where the compiler should enforce exhaustive handling.

Senior Engineer Tips

- When you feel a growing `switch` / `is` chain on a type, that's the signal to introduce a polymorphic method (or the Strategy pattern).
- LSP violations often hide in overrides that throw `UnsupportedError` — a sign the hierarchy is wrong (the classic `Square extends Rectangle` setter trap).
- Choose polymorphism (open) vs sealed types (closed) based on who's expected to add variants — you, or external code.

Architect Perspective

Polymorphism is the mechanism behind pluggable architectures: define stable abstractions (repositories, payment methods, notification channels) and let concrete implementations vary independently. It's the runtime counterpart to dependency inversion and the backbone of testability (swap real for fake) and extensibility across the whole system (Modules 04, 05, 40).

Summary

- Polymorphism = supertype reference, subtype behavior, via dynamic dispatch.
- Prefer overriding to type-switch chains; program to abstractions; uphold LSP.
- Use polymorphism for open extension, sealed `switch` for closed exhaustive handling.

Revision Notes

- Dynamic dispatch: method chosen by runtime type; compiler checks static type.
- Override with `@override`; program to the base/abstraction.
- LSP: subtypes substitutable — no stronger preconditions/weaker postconditions/surprise throws.
- `is` / `as` chains → replace with polymorphic method or Strategy.

Practice Questions

1. Why does adding a `Pentagon` require no change to `totalArea`?
2. Give a concrete LSP violation and explain the breakage.
3. When would you pick a sealed `switch` over polymorphism?

Coding Questions

1. Build a `Notifier` base with `EmailNotifier` / `SmsNotifier` / `PushNotifier` overrides and a `sendAll(List<Notifier>)`.

2. Refactor an `if (shape is ...)` area calculator into polymorphic `area()` methods.
3. Demonstrate the `Square extends Rectangle` LSP trap and fix it (separate types).

Mini Project

Payment processor (pure Dart): Define an abstract `PaymentMethod` with `charge(amount)`; implement `Card`, `Upi`, and `Wallet` subtypes; write a `Checkout` that charges any method uniformly and totals fees — all without type checks. Add a new method type to prove Open/Closed. Acceptance: no `is` / `as` in `Checkout`; new subtype added with zero edits to `Checkout`; LSP honored; `dart analyze` clean.

Abstraction & Interfaces (`abstract`, `implements`, `sealed`)

> Abstraction exposes *what* a type does while hiding *how*; in Dart, abstract classes define partial contracts, every class is an implicit interface, and `sealed` classes create closed, exhaustively-checkable type families.

Introduction

This file covers **abstract classes** (can't be instantiated; mix concrete + abstract members), Dart's **implicit interfaces** (every class defines one; `implements` adopts the contract with no implementation reuse), Dart 3 **class modifiers** (`sealed`, `final`, `base`, `interface`, `mixin`), and when to use each.

Why this concept exists

Callers should depend on **capabilities**, not concrete implementations — so you can swap a real database for a fake in tests, or one payment provider for another. Abstraction provides that seam. Sealed classes solve the opposite need: a *closed* set of variants where the compiler forces you to handle every case (state machines, results).

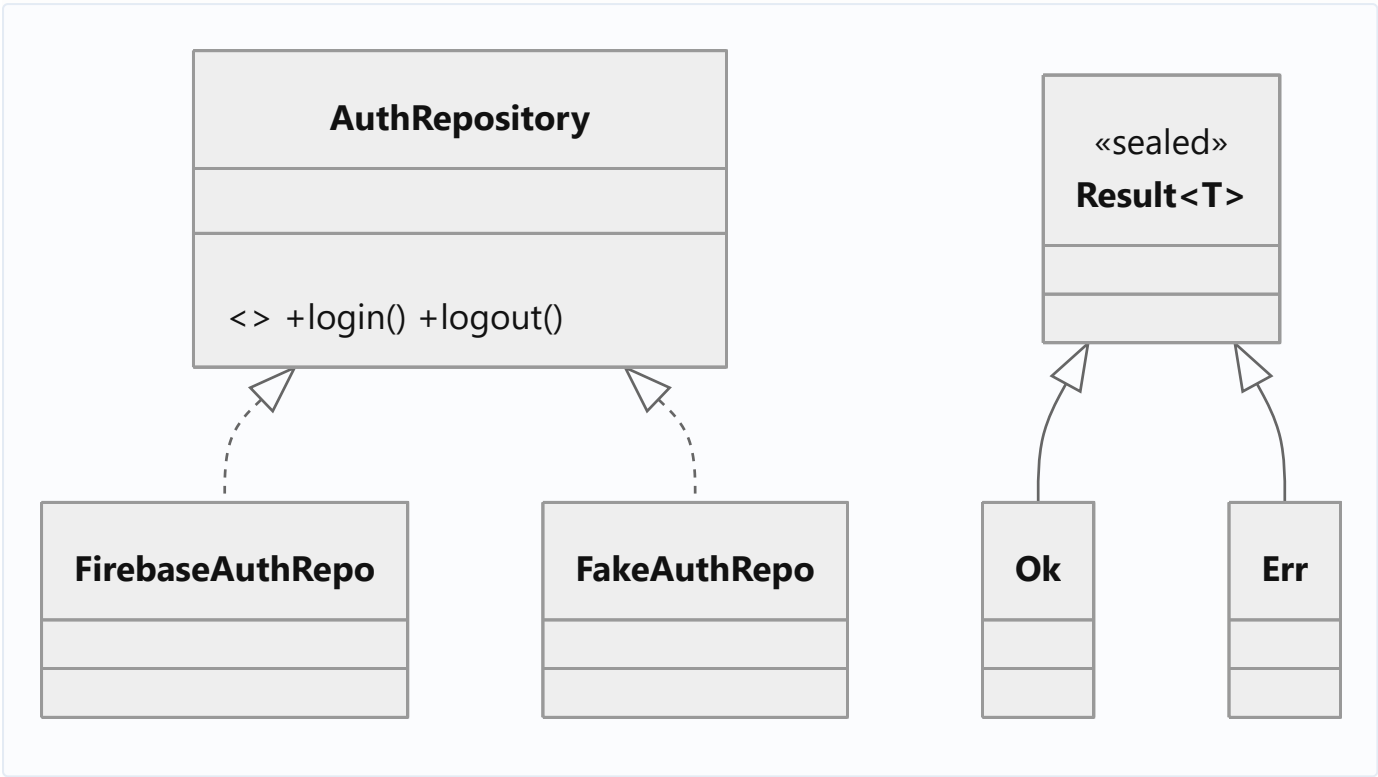
Real-world analogy

- **Abstract class / interface** = a **job description** ("must be able to `authenticate` and `logout`"). Anyone who fulfills it can do the job; you hire against the description, not the person.
- **Sealed class** = a **fixed multiple-choice question** — the answer is exactly one of a known set (`Loading | Success | Error`), and you must account for all of them.

Problem Statement

Define an `AuthRepository` contract so your app doesn't depend on Firebase directly (swap it in tests), and model a `Result` type as a closed set (`Ok` / `Err`) the compiler forces you to handle exhaustively. You'll use an abstract class/interface and a sealed class.

Internal Working



- **Abstract class:** `abstract class A { void doIt(); String shared() => '...'; }` — can't be instantiated; subclasses implement abstract members and inherit concrete ones (`extends`).
- **Implicit interface:** *every* class induces an interface (its public members). `class B implements A` must provide **all** of `A` 's members — no implementation is inherited.
- `extends` **vs** `implements` **:** `extends` reuses code (one superclass, `super`); `implements` adopts only the contract (any number).
- **Dart 3 class modifiers:**
  - `sealed` — abstract + **exhaustively switchable**; all direct subtypes must be in the same library. Enables compiler-checked `switch` .
  - `final` — cannot be extended or implemented outside its library.
  - `base` — can be extended (inheriting implementation) but not implemented; forces subclasses to use `extends` .
  - `interface` — can be implemented but not extended outside its library (pure contract).

Memory Representation

Not applicable beyond normal objects. `sealed` /modifiers are compile-time constraints; instances are ordinary.

Compiler Behavior

- Instantiating an abstract class is a compile error.

- `implements X` without providing all of `X`'s members is a compile error.
- `switch` over a `sealed` type is checked for **exhaustiveness** (missing a subtype = compile error, no `default` needed).
- Modifiers enforce extend/implement rules across library boundaries.

Runtime Behavior

- Interfaces/abstracts dispatch polymorphically (04_polymorphism.md).
- Sealed `switch` picks the arm by runtime subtype; adding a subtype breaks compilation until handled.

Flutter Engine Behavior

Not applicable. (Flutter's `Listenable`, `Comparable`, and many contracts are interfaces you implement; sealed types are increasingly used for BLoC states.)

Dart VM Behavior

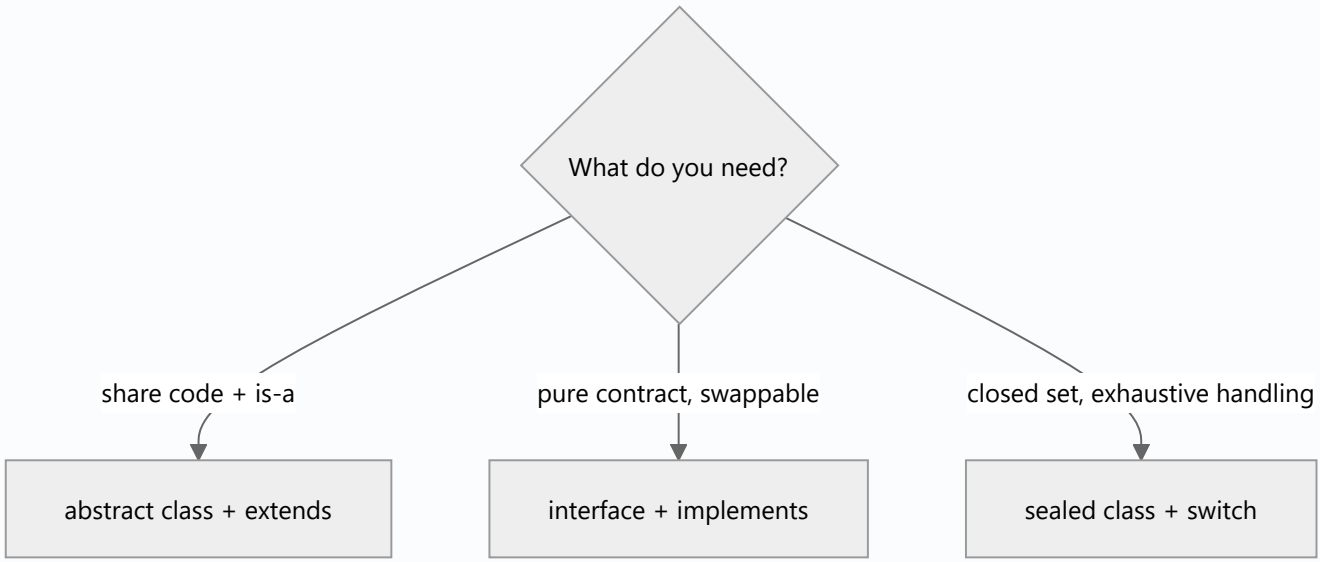
- Modifiers affect only compile-time checking; no runtime cost. Exhaustive switches may compile to efficient dispatch.

Examples

```
// Interface via abstract class (contract only, then implemented):
abstract interface class AuthRepository {
  Future<String> login(String user, String pass);
  Future<void> logout();
}

class FakeAuthRepo implements AuthRepository {
  @override
  Future<String> login(String user, String pass) async => 'token_$user';
  @override
  Future<void> logout() async {}
}

// Abstract class with SHARED code + abstract member (use extends):
abstract class Animal {
  String get name;              // abstract
  String describe() => 'I am $name'; // shared concrete
```

```
}

class Dog extends Animal {
  @override
  String get name => 'Dog';
}

// Sealed: closed set, exhaustive switch
sealed class Result<T> {
  const Result();
}
class Ok<T> extends Result<T> {
  final T value;
  const Ok(this.value);
}
class Err<T> extends Result<T> {
  final String message;
  const Err(this.message);
}

String render<T>(Result<T> r) => switch (r) {
      Ok(:final value) => 'ok: $value',
      Err(:final message) => 'err: $message',
      // no default needed — compiler knows the set is closed & complete
    };

Future<void> main() async {
  final AuthRepository repo = FakeAuthRepo(); // depend on the abstraction
  print(await repo.login('ada', 'pw'));       // token_ada

  print(Dog().describe()); // I am Dog

  print(render(const Ok(42)));       // ok: 42
  print(render(const Err('boom')));  // err: boom
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`implements` a class then forgetting members	Compile error (must provide all)	Implement the full contract
Depending on a concrete class instead of an interface	Not swappable/testable	Depend on the abstraction (DIP)
Using `sealed` for an open/extensible set	External code can't add variants	Use abstract/interface for open extension
Adding `default` to a sealed `switch`	Defeats exhaustiveness protection	Handle each subtype
Confusing `extends` (reuse) with `implements` (contract)	Wrong reuse semantics	Pick by whether you want implementation

Best Practices

- **Depend on abstractions** (interfaces) for anything you'll swap/mock (repositories, services, clients).
- Use **abstract classes** when subtypes share real implementation; **interfaces** for pure contracts.
- Use **sealed classes** for closed variant sets (results, UI states, events) to get exhaustive `switch`.
- Apply Dart 3 modifiers (`final` / `base` / `interface` / `sealed`) to make your design intent enforceable.

Performance

- No runtime cost from abstraction/modifiers; dispatch is normal virtual dispatch.

Advantages / Disadvantages

- **+** Decoupling, testability, extensibility (interfaces); compiler-enforced completeness (sealed).
- **−** More types/indirection; sealed sets can't be extended by consumers (by design).

Interview Questions

1. 🟢 **Abstract class vs interface in Dart?** — Dart has no separate `interface` keyword by default: every class is an implicit interface. An abstract class can't be instantiated and may mix concrete + abstract members; you `extend` it to reuse code or `implement` it to take only its contract.
2. 🟢 `extends` **vs** `implements`**?** — `extends` inherits implementation (one superclass, `super`); `implements` adopts only the contract and forces you to reimplement everything.
3. 🟡 **What is a sealed class and why use it?** — A closed type family (all subtypes in one library) that enables exhaustive `switch` — the compiler forces handling of every variant, ideal for results/states.
4. 🟡 **Name the Dart 3 class modifiers and their effect.** — `sealed` (closed + exhaustive), `final` (no extend/implement outside lib), `base` (extend-only), `interface` (implement-only), `mixin` / `mixin class`.
5. 🟡 **Why depend on an interface instead of a concrete class?** — Decoupling and testability: you can swap implementations (real/fake) without changing callers (Dependency Inversion — Module 04).
6. 🔴 **When choose sealed vs an open interface?** — Sealed when the variant set is closed and you want compiler-enforced exhaustiveness; open interface/abstract when external code should add new implementations.
7. 🔴 **Can you implement multiple interfaces? Extend multiple classes?** — Implement many interfaces: yes. Extend multiple classes: no (single inheritance); use mixins for extra behavior.

Senior Engineer Tips

- Define interfaces at the **consumer's** side (what the caller needs), not mirroring an implementation — keeps them lean and stable (Interface Segregation).
- Reach for `sealed` + patterns to replace enum-plus-nullable-fields state modeling (02 · records_and_patterns).
- Use `base` / `final` to stop accidental implementation of classes not designed for it.

Architect Perspective

Abstraction boundaries are where architecture lives: interfaces define the contracts between layers (UI↔domain↔data), enabling substitution, testing, and parallel team work; sealed types make domain state spaces explicit and total. Together they are the backbone of Clean Architecture and robust state management (Modules 40, 11).

Summary

- Abstract classes: partial contracts + shared code (`extends`). Interfaces: pure contracts every class induces (`implements`).
- Sealed classes: closed variant sets with exhaustive `switch` .
- Depend on abstractions for swap/test; use Dart 3 modifiers to enforce design intent.

Revision Notes

- Every class = implicit interface; `implements` = contract only (reimplement all); `extends` = reuse (one).
- Abstract class = no instances; mix concrete + abstract members.
- `sealed` = closed + exhaustive `switch` (subtypes same library, no `default`).
- Modifiers: `sealed` / `final` / `base` / `interface` ; depend on abstractions (DIP).

Practice Questions

1. Why does a sealed `switch` need no `default` ?
2. When would you pick an abstract class over an interface?
3. What does `base` prevent and why might you want it?

Coding Questions

1. Define a `Cache` interface and provide `MemoryCache` / `NoopCache` implementations; depend on the interface.
2. Model a sealed `RemoteData<T>` (`Loading` / `Data` / `Failure`) and an exhaustive renderer.
3. Create an abstract `Report` with shared `header()` + abstract `body()` ; implement two report types.

Mini Project

Auth + result contracts (pure Dart): Define an `AuthRepository` interface with `FakeAuthRepo` / `HttpAuthRepo` implementations, and a sealed `AuthResult` (`Authenticated` / `Failed` / `Locked`) handled by an exhaustive `switch` . Wire a `LoginUseCase` depending only on the interface. Acceptance: no concrete dependency in the use case; exhaustive result handling (no `default`); swappable in tests; `dart analyze` clean.

Composition & Relationships (Composition vs Inheritance, Aggregation, Association)

> "Favor composition over inheritance": build behavior by *combining* objects (has-a) rather than *extending* them (is-a) — it's more flexible, swappable, and testable.

Introduction

Objects relate in several ways: **inheritance** (is-a), **composition** (has-a, owned/lifecycle-bound part), **aggregation** (has-a, independent lifecycle), and **association** (uses/knows-a). This file defines each, contrasts composition with inheritance (and mixins), and gives the decision rules senior engineers actually use.

Why this concept exists

Inheritance couples a subclass to a base's implementation and forces a rigid taxonomy. Most reuse needs are better served by **plugging in collaborators** — you can swap them, test them in isolation, and change behavior at runtime. Recognizing the *kind* of relationship also keeps ownership and lifecycles clear (who disposes what).

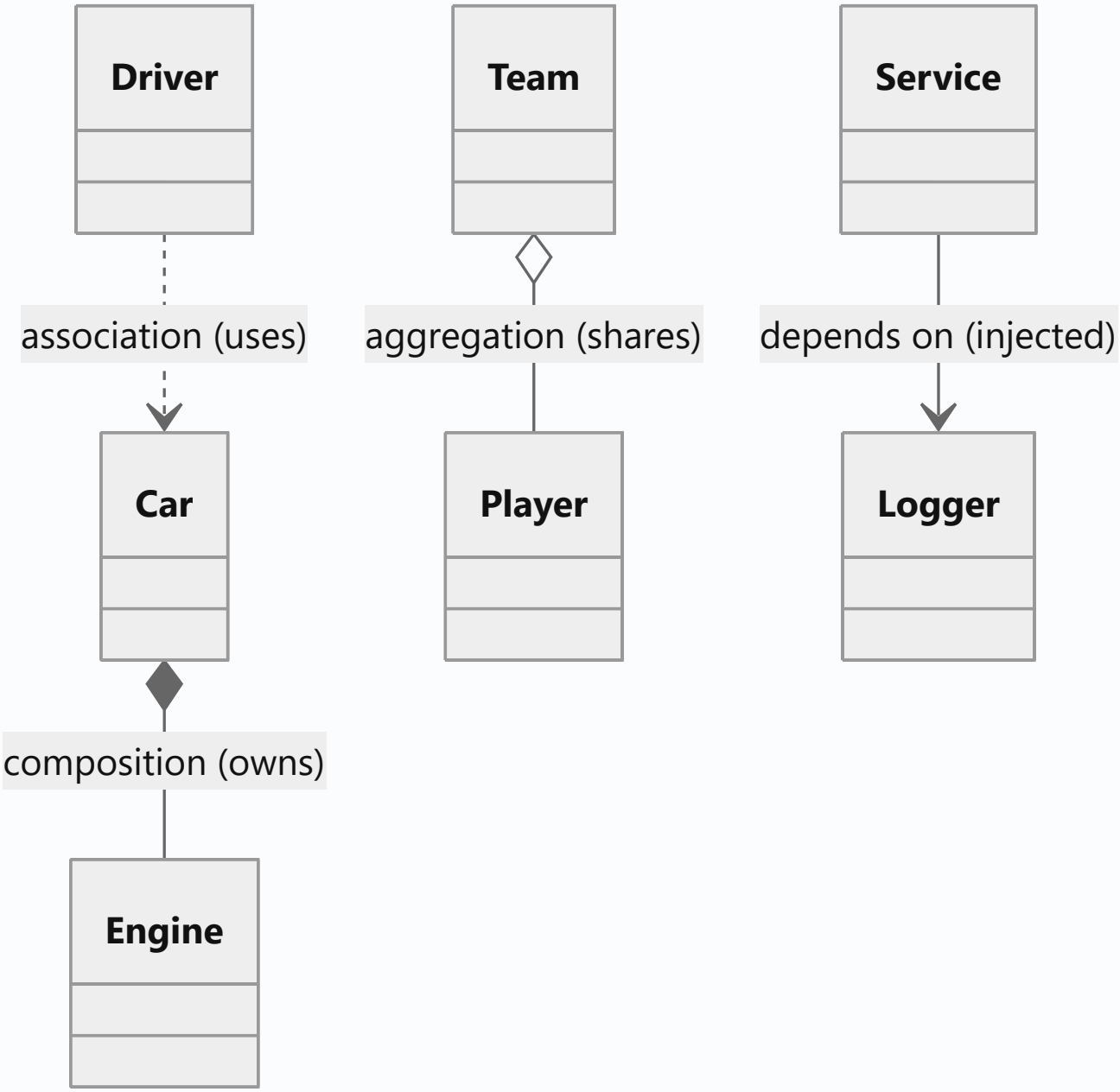
Real-world analogy

- **Inheritance (is-a):** a `Car` *is a* `Vehicle`.
- **Composition (has-a, owned):** a `Car` *has an* `Engine` that lives and dies with the car — no car, no engine.
- **Aggregation (has-a, independent):** a `Team` *has* `Players`, but players exist before/after the team; disbanding the team doesn't destroy the players.
- **Association (uses-a):** a `Driver` *uses* a `Car` — they interact, neither owns the other.

Problem Statement

Add "logging" and "caching" behavior to several services without a fragile base class, and model a `Car` / `Engine` (composition) and `PlaylistLibrary` / `Song` (aggregation). You'll compose collaborators and inject them.

Internal Working



Relationship	UML	Semantics	Lifecycle	Dart shape
Inheritance	▷ (hollow)	is-a	—	`extends`
Composition	◆ filled	has-a, owns the part	part dies with whole	field created/owned by the object
Aggregation	◇ hollow	has-a, shares the part	independent	field referencing an externally-owned object
Association	→	uses/knows	independent	passed-in reference / method param

- **Composition:** the object *creates and owns* its parts (`_engine = Engine()`), and is responsible for their disposal.

- **Aggregation:** the object *references* parts owned elsewhere (`Team(this.players)`), and must **not** dispose them.
- **Delegation:** composition's mechanism — forward calls to a collaborator (`void start() => _engine.ignite();`).
- **Composition vs mixins:** mixins add behavior at *compile time* into the type; composition holds a *runtime* collaborator you can swap/inject. Prefer composition when the behavior has state/lifecycle or should be swappable (02 · mixins).

Memory Representation

- Composition: the whole holds a reference to its owned part; both are heap objects, the part reachable through the whole (and typically only through it).
- Aggregation: the part is reachable through multiple owners; disposing one owner must not free a shared part.

Compiler Behavior

Not special — these are design patterns expressed with ordinary fields/constructors. The compiler enforces types of injected collaborators.

Runtime Behavior

- Delegated calls dispatch to the collaborator at runtime; swapping the collaborator (e.g., a fake logger in tests) changes behavior without touching the host.

Flutter Engine Behavior

Not applicable, but Flutter is composition-first: widgets are *composed* trees (`Padding(child: Text(...))`), not deep inheritance hierarchies — "composition over inheritance" is a core Flutter philosophy (Module 07).

Dart VM Behavior

Not applicable beyond normal dispatch.

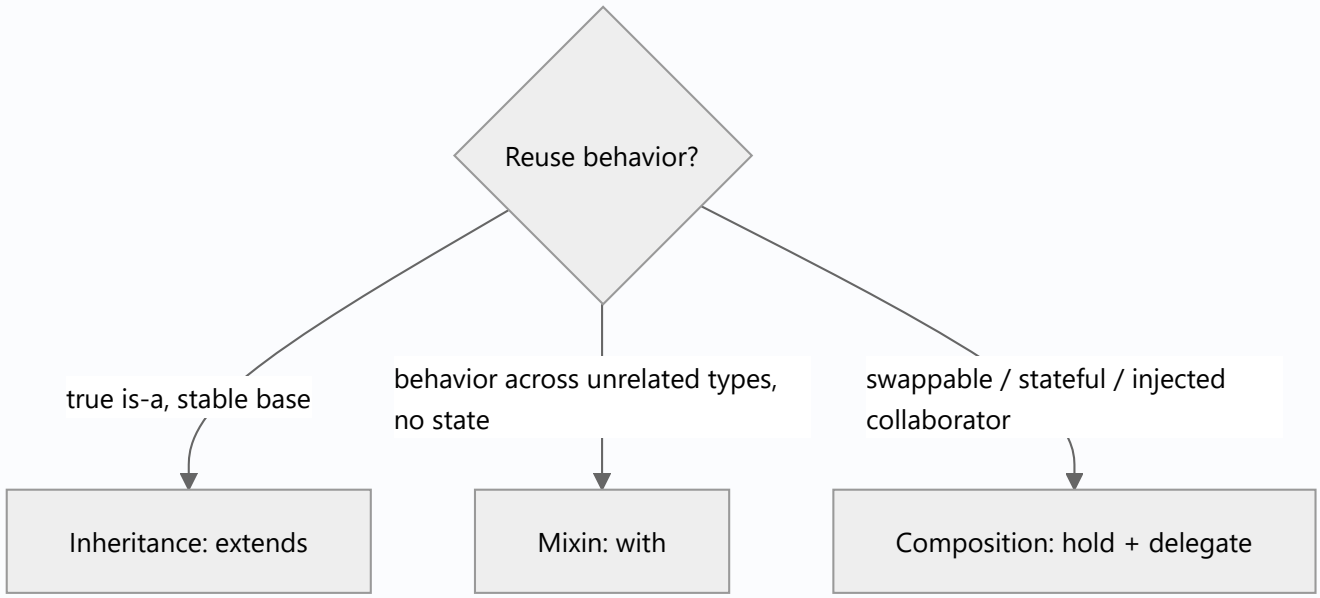
Examples

```
// COMPOSITION via delegation + dependency injection
abstract interface class Logger {
  void log(String msg);
}
```

```
class ConsoleLogger implements Logger {
  @override
  void log(String msg) => print('[LOG] $msg');
}

class PaymentService {
  final Logger _logger; // injected collaborator (composition/DI)
  PaymentService(this._logger);
  void pay(double amount) {
    _logger.log('charging $amount'); // delegate
  }
}

// COMPOSITION with ownership
class Engine {
  void ignite() => print('vroom');
  void dispose() => print('engine off');
}
class Car {
  final Engine _engine = Engine(); // Car OWNS the engine
  void start() => _engine.ignite(); // delegation
  void scrap() => _engine.dispose(); // owner disposes the part
}

// AGGREGATION: shared, independent lifecycle
class Song {
  final String title;
  Song(this.title);
}
class Playlist {
  final List<Song> songs; // referenced, NOT owned
  Playlist(this.songs);   // songs live independently
}

void main() {
  PaymentService(ConsoleLogger()).pay(100); // swap logger freely (tests: FakeLogger)
  Car().start(); // vroom

  final s1 = Song('A'), s2 = Song('B');
  final rock = Playlist([s1, s2]);
```

```
  final chill = Playlist([s1]); // s1 shared across playlists (aggregation)
  print(rock.songs.length + chill.songs.length); // 3 references, 2 songs
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Inheriting to reuse a method	Fragile, wrong is-a	Compose/delegate instead
Disposing an aggregated (shared) part	Frees something others use	Only owners (composition) dispose
Deep inheritance for cross-cutting behavior	Rigid	Composition + DI, or a mixin
God object composing everything	Low cohesion	Split responsibilities (SRP)
Confusing composition and aggregation ownership	Lifecycle/leak bugs	Decide who owns/disposes

Best Practices

- **Favor composition over inheritance**; use inheritance only for true, stable is-a.
- Inject collaborators (constructor DI) so they're swappable and testable (Module 14).
- Be explicit about ownership: the owner (composition) disposes; aggregators don't.
- Use mixins for stateless cross-cutting behavior; composition when state/lifecycle/swapping is involved.

Performance

- Delegation adds a cheap indirection; negligible. The win is maintainability, not speed.

Advantages / Disadvantages

- **+ Composition:** flexible, swappable, testable, runtime-configurable, avoids fragile base class.
- **− Composition:** more wiring/boilerplate (mitigated by DI); many small collaborators to track.
- **+ Inheritance:** concise reuse when is-a truly holds.
- **− Inheritance:** tight coupling, single-inheritance limit, fragility.

Interview Questions

1. 🟢 **What does "favor composition over inheritance" mean?** — Prefer building behavior by combining/delegating to collaborator objects (has-a) rather than extending a base (is-a), for flexibility and testability.
2. 🟢 **Composition vs aggregation?** — Both are has-a; composition **owns** the part (shared lifecycle, disposes it), aggregation references an **independently-owned** part (must not dispose it).
3. 🟡 **Composition vs inheritance tradeoffs?** — Inheritance couples to the base's implementation and forces is-a; composition is swappable, testable, runtime-configurable, and avoids the fragile base class problem.
4. 🟡 **Composition vs mixin?** — Mixins add behavior into the type at compile time (stateless cross-cutting); composition holds a runtime collaborator you can inject/swap (good when state/lifecycle matters).
5. 🟡 **What is delegation?** — Forwarding a call to a held collaborator object — the mechanism composition uses to reuse behavior.
6. 🔴 **How does composition enable testability?** — You inject a fake collaborator (e.g., `FakeLogger` / `FakeRepo`) in tests without changing the host class.
7. 🔴 **Why is Flutter "composition over inheritance"?** — UIs are built by nesting/composing widgets (has-a child trees) rather than subclassing, which is more flexible and reusable.

Senior Engineer Tips

- Default to composition + constructor injection; introduce inheritance only when you can defend the is-a and the base is stable.
- Draw the relationship (◆ owns vs ◇ shares vs → uses) to decide disposal responsibility — most leak/lifecycle bugs are ownership confusion.
- Keep composed collaborators behind interfaces so they're mockable and replaceable (05_abstraction_and_interfaces.md).

Architect Perspective

A composition-first design yields loosely coupled, independently testable, swappable modules — the practical enablement of Dependency Inversion and Clean Architecture. Ownership clarity (composition vs aggregation) governs resource lifecycle and prevents leaks at scale. This philosophy is why Flutter and modern Dart apps compose small pieces rather than build deep hierarchies (Modules 14, 40, 07).

Summary

- Relationships: is-a (inheritance), has-a-owned (composition), has-a-shared (aggregation), uses-a (association).
- Favor composition + DI over inheritance; delegate to swappable collaborators.
- Be explicit about ownership/disposal; use mixins for stateless cross-cutting behavior.

Revision Notes

- Composition = owns part (disposes it); aggregation = shares part (don't dispose); association = uses.
- Favor composition over inheritance; delegate + inject collaborators.
- Mixin = compile-time behavior (stateless); composition = runtime, swappable, stateful.
- Flutter = compose widgets, not subclass.

Practice Questions

1. Should a `Team` dispose its `Player`s? Why (which relationship)?
2. When is a mixin better than composition, and vice versa?
3. How does composition make a class easier to unit test?

Coding Questions

1. Refactor a `LoggingRepository extends Repository` into composition (repo *has-a* logger).
2. Model `Car` (owns `Engine`) vs `Garage` (aggregates `Car`s) and show correct disposal.
3. Build a `NotificationSender` composed of a swappable `Channel` (email/sms), injected via constructor.

Mini Project

Composable service layer (pure Dart): Build a `CheckoutService` composed of injected collaborators (`PaymentGateway` , `Logger` , `InventoryClient`), each behind an interface, plus a `Cart` (owns line items) and a `Catalog` (aggregates products). Swap real collaborators for fakes in tests. Acceptance: no inheritance-for-reuse; collaborators injected + mockable; correct ownership/disposal documented; `dart analyze` clean.

Equality & Copying (`==` / `hashCode`, Identity, Deep vs Shallow Copy)

> By default `==` means "same object" (identity); to make it mean "same value" you must override `==` **and** `hashCode` together — and know whether your copies are shallow (share nested objects) or deep (fully independent).

Introduction

This file covers Dart's two notions of equality — **identity** (`identical`) vs **value** (`==`), how to implement value equality correctly (`==` + `hashCode`), and **deep vs shallow copying**. These underpin correct `Set` / `Map` behavior, reactive-state diffing, and avoiding shared-mutation bugs.

Why this concept exists

Sometimes two distinct objects should be considered "equal" (two `Money(100)` are the same value); sometimes you care they're literally the same instance. Hash-based collections and Flutter's rebuild-diffing rely on consistent equality. Copying matters because Dart objects are references — a naive copy can secretly share mutable inner objects, causing "I changed A and B changed too" bugs.

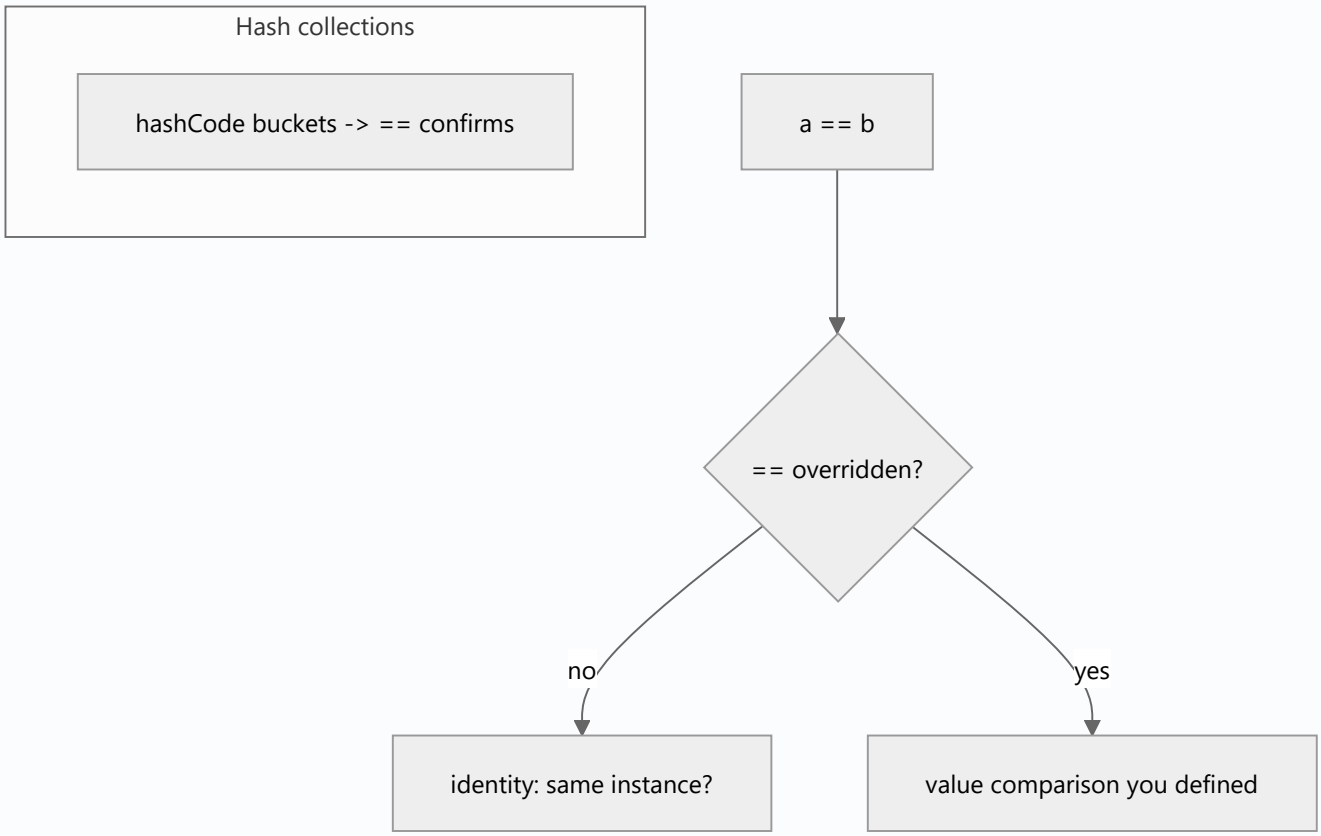
Real-world analogy

- **Identity vs value:** two identical ₹100 notes are *equal in value* but are *different physical notes* (different identity). `==` can be taught to treat them as equal; `identical` never will.
- **Shallow vs deep copy:** photocopying a folder's cover but keeping the *same* documents inside (shallow — edits leak) vs photocopying the cover *and every document* (deep — fully independent).

Problem Statement

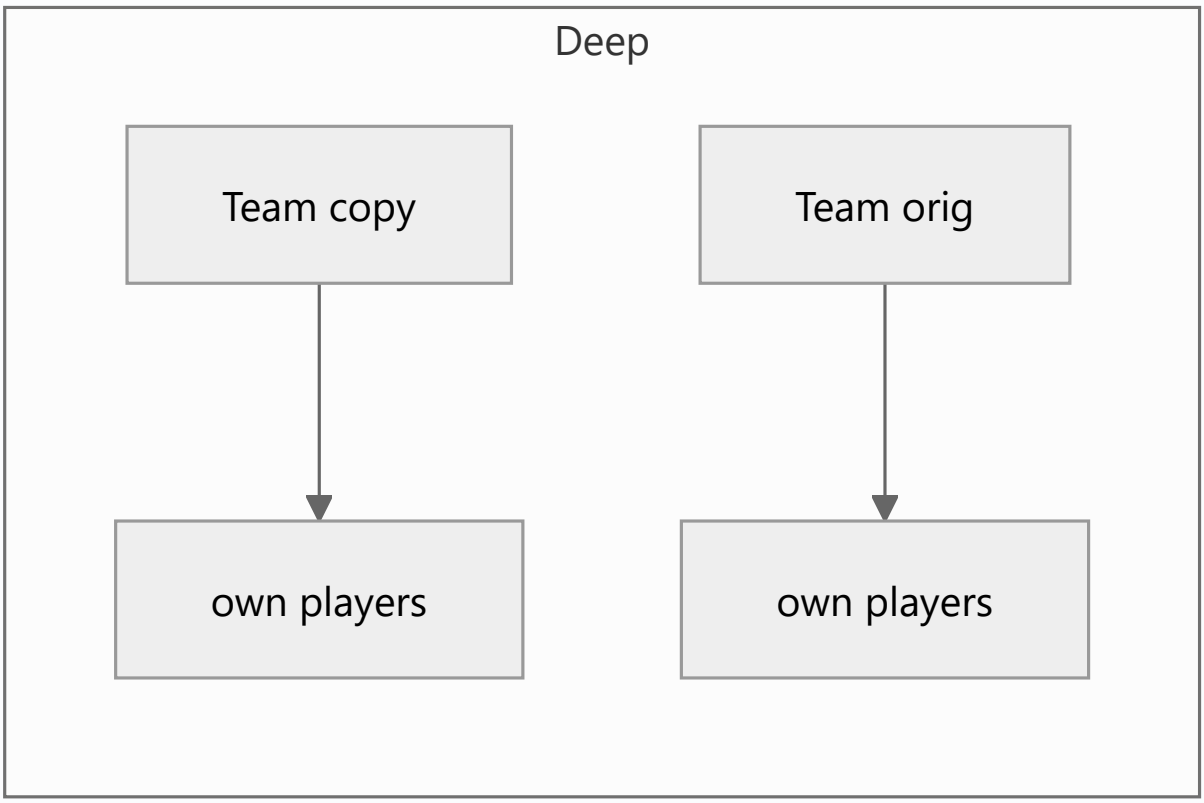
Make `Money(100) == Money(100)` true and safe as a `Map` key, then copy a `Team` with a list of players such that editing the copy's list doesn't affect the original. You'll override `==` / `hashCode` and implement deep vs shallow copies.

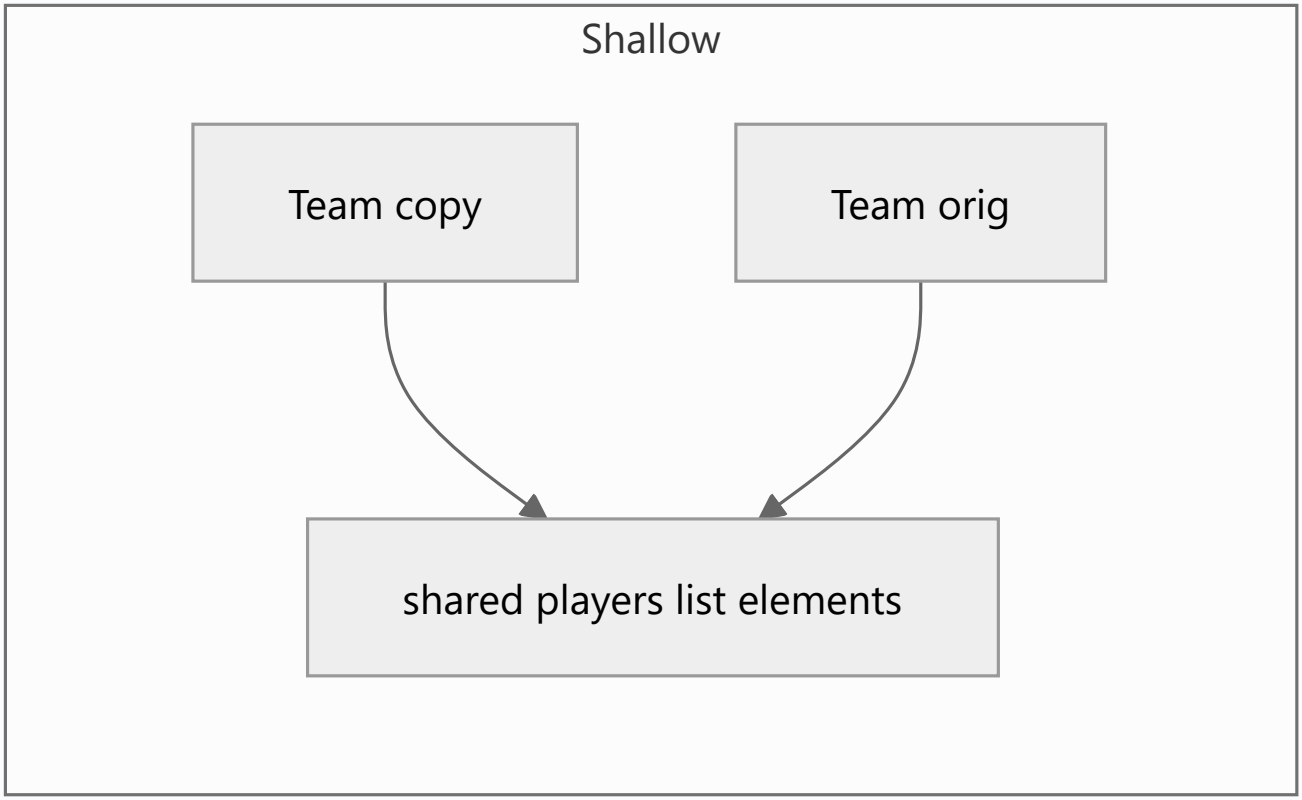
Internal Working



- **Identity:** `identical(a, b)` — same object in memory. Default `==` falls back to identity.
- **Value equality:** override `bool operator ==(Object other)` to compare fields, and override `int get hashCode` consistently (**equal objects must have equal hash codes**).
- **The contract:** if `a == b` then `a.hashCode == b.hashCode`; equality must be reflexive, symmetric, transitive, and consistent.
- `Object.hash` / `Object.hashAll` build good hash codes from fields.
- **Shallow copy:** new outer object, **same** references to nested objects (`List.of` copies the list but shares elements).
- **Deep copy:** new outer object **and** new copies of nested mutable objects (recursively).

Memory Representation





- Immutable objects make copying moot (share freely — they can't change). Deep copies are only needed for **mutable** nested state.

Compiler Behavior

- Overriding `==` but not `hashCode` triggers a lint (`hash_and_equals`) — a common bug source.
- `const` canonicalization makes structurally-equal `const` instances `identical` (so `==` is trivially true and hashing consistent).

Runtime Behavior

- `Set` / `Map` use `hashCode` to bucket, then `==` to confirm. Wrong/absent `hashCode` → lookups miss, duplicates sneak in.
- Mutating a field used in `hashCode` **after** inserting into a hash set corrupts the set (the object is in the wrong bucket) — keys should be immutable.

Flutter Engine Behavior

Not applicable at engine level, but value equality drives rebuild-skipping and selector correctness in state management (Modules 11, 21); `Key` equality affects element reuse (Module 08).

Dart VM Behavior

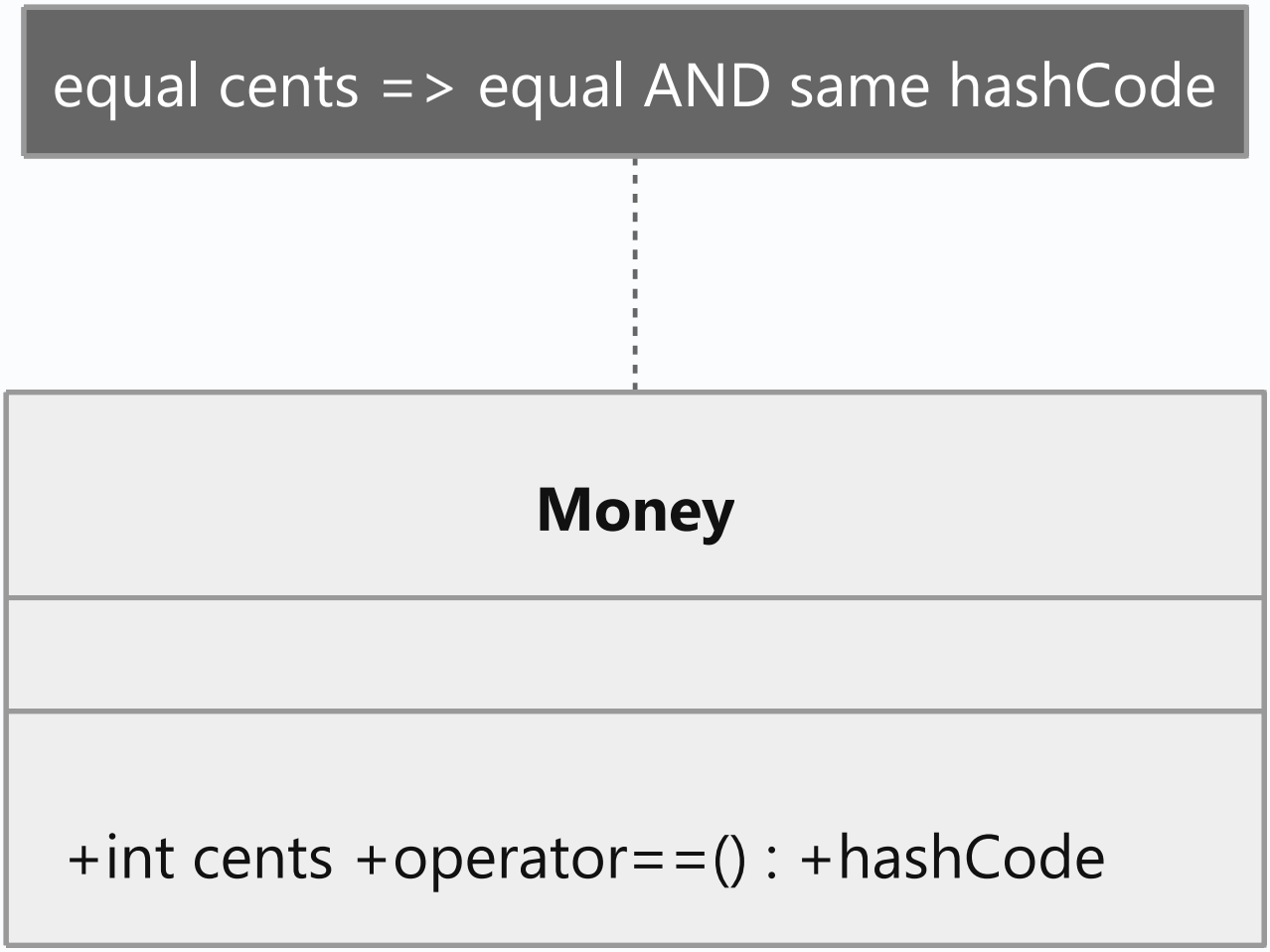
- Default `hashCode` is an identity hash assigned lazily. Custom `hashCode` runs your computation each call (cache it for expensive cases if profiling warrants).

Examples

```
class Money {
  final int cents;
  const Money(this.cents);

  @override
  bool operator ==(Object other) => other is Money && other.cents == cents;
  @override
  int get hashCode => cents.hashCode;
}

class Player {
  String name; // mutable
```

```
  Player(this.name);
}

class Team {
  final String name;
  final List<Player> players;
  Team(this.name, this.players);

  // SHALLOW: new list, SAME Player objects
  Team shallowCopy() => Team(name, List.of(players));

  // DEEP: new list AND new Player objects
  Team deepCopy() => Team(name, players.map((p) => Player(p.name)).toList());
}

void main() {
  // value equality:
  print(Money(100) == Money(100));      // true
  print(identical(Money(100), Money(100))); // false (distinct instances)
  print({Money(100), Money(100)}.length);   // 1 — dedup via == + hashCode
  final prices = {Money(100): 'cheap'};
  print(prices[Money(100)]);            // cheap — works as a key

  // shallow vs deep copy:
  final orig = Team('A', [Player('Ada')]);
  final shallow = orig.shallowCopy();
  final deep = orig.deepCopy();

  shallow.players.add(Player('Bob')); // list is independent (shallow copied list)
  print(orig.players.length);         // 1 — orig list unaffected

  shallow.players[0].name = 'CHANGED'; // SAME Player object as orig!
  print(orig.players[0].name);         // CHANGED — shared nested object (shallow)

  deep.players[0].name = 'X';          // deep copy: independent
  print(orig.players[0].name);         // CHANGED (unchanged by deep copy)
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Override `==` without `hashCode`	Breaks Set/Map	Override both (or use equatable/freezed/records)
Inconsistent `==` / `hashCode` (different fields)	Contract violated → collection bugs	Use the same fields in both
Mutating a hash-key field after insertion	Object lands in wrong bucket	Use immutable keys
Assuming a copy is deep	Shared nested mutables leak edits	Deep-copy mutable nesting (or use immutables)
Hand-rolling equality for many models	Error-prone	Use `equatable` / `freezed` /records (02 · immutability)

Best Practices

- Implement value equality with **the same fields** in `==` and `hashCode`; use `Object.hash` / `hashAll`.
- Keep hash-key objects **immutable**.
- Prefer **immutable models** so copying is unnecessary; when copying mutable state, be explicit about shallow vs deep.
- Prefer `records` / `equatable` / `freezed` to generate correct equality/ `copyWith`.

Performance

- `hashCode` runs per lookup; keep it cheap (combine field hashes). Cache only if profiling shows a hotspot.
- Deep copies cost O(size); immutability + structural sharing usually avoids the need.

Advantages / Disadvantages

- **+ Value equality:** correct dedup/keys, reliable state diffing.
- **− Value equality:** boilerplate (mitigated by tooling); must maintain the contract.
- **+ Deep copy:** true independence. **− Deep copy:** cost; often unnecessary with immutability.

Interview Questions

1. 🟢 **`==` vs `identical`?** — `==` is value/logical equality (overridable, default identity); `identical(a,b)` is reference equality (same instance), never overridable.
2. 🟢 **Why override `hashCode` with `==`?** — Hash collections bucket by `hashCode` then confirm with `==`; equal objects must share a hash code or Set/Map break.
3. 🟡 **What is the equality contract?** — Reflexive, symmetric, transitive, consistent; and `a == b ⇒ a.hashCode == b.hashCode`.
4. 🟡 **Shallow vs deep copy?** — Shallow copies the outer object but shares nested references; deep copies nested mutable objects too, giving full independence.
5. 🟡 **Why keep hash keys immutable?** — Mutating a field used in `hashCode` after insertion misplaces the object in its bucket, corrupting lookups.
6. 🔴 **How does value equality affect Flutter rebuilds?** — State diffing compares old/new via `==`; correct value equality enables rebuild-skipping and correct selectors, avoiding missed/extra rebuilds.
7. 🔴 **How do `const` objects interact with equality?** — Structurally-equal `const` instances are canonicalized (`identical` true), so equality and hashing are trivially consistent.

Senior Engineer Tips

- Reach for `freezed` / `equatable` / `records` immediately for models — hand-written equality across many classes is a recurring bug source.
- Make copies unnecessary by defaulting to immutability; when you must copy mutable graphs, document/test the depth.
- Beware equality that includes volatile fields (timestamps, caches) — it breaks memoization and diffing.

Architect Perspective

Equality and copy semantics are load-bearing for reactive state, caching, and messaging (immutable value-equal objects can be shared/compared safely, even across isolates). Standardize on immutable models with generated value equality; it eliminates a whole class of aliasing and diffing bugs and makes the system's data flow predictable (Modules 02, 11, 46).

Summary

- Default `==` is identity; override `==` **and** `hashCode` (same fields) for value equality.
- Keep hash keys immutable; honor the equality contract.
- Shallow copies share nested objects; deep copies don't. Prefer immutability to make copying moot; use tooling for equality.

Revision Notes

- `==` value (overridable, default identity); `identical` = same instance.
- Override `==` + `hashCode` together, same fields; `a==b ⇒ hashCode==`.
- Immutable hash keys; shallow shares nested, deep copies nested.
- Prefer immutability + `equatable` / `freezed` /records.

Practice Questions

1. Why does a `Set<Money>` need `hashCode`, not just `==`?
2. Explain why `shallow.players[0].name = 'X'` changed the original.
3. When is a deep copy unnecessary?

Coding Questions

1. Implement value equality for a `Point3D(x,y,z)` using `Object.hash`; prove Set dedup.

2. Given a mutable nested model, write `shallowCopy` and `deepCopy` and demonstrate the difference.
3. Show a bug where mutating a key after `Set.add` breaks `contains`, then fix with immutability.

Mini Project

Equality & copy lab (pure Dart): Build an immutable `Money` value object (value equality + Map key), and a mutable `Document` graph with correct `shallowCopy` / `deepCopy`. Write tests demonstrating: value equality dedup, hash-key correctness, shallow sharing, and deep independence. Acceptance: `==` / `hashCode` consistent; documented shallow vs deep behavior verified by tests; `dart analyze` clean.