01 · Dart Fundamentals

Flutter Practice Notes

Contents

01 · Dart Fundamentals

Introduction

Flutter is written in **Dart**, so fluency in Dart is non-negotiable. This module builds that fluency from first principles — not "here's the syntax," but *why the language is shaped this way* and *what the compiler and VM do with your code*. Master this and every later module (widgets, state, architecture) rests on solid ground.

Why this module exists

You cannot reason about Flutter performance, rebuilds, or memory without understanding Dart's type system, null safety, and value/reference semantics. Interviewers at top companies probe these fundamentals precisely because they predict whether you'll write correct, efficient UI code.

Module map

#	File	Topic	Level
1	01_variables_and_mutability.md	`var` / `final` / `const` / `late`, type inference	🟢
2	02_data_types.md	`num` / `int` / `double`, `String`, `bool`, `dynamic`, `Object?`	🟢
3	03_operators.md	Arithmetic, logical, null-aware, cascade, spread	🟢
4	04_control_flow.md	`if` / `for` / `while` / `switch`, collection-if/for	🟢
5	05_functions.md	Params, arrow, closures, higher-order, `typedef`	🟢
6	06_collections.md	`List` / `Set` / `Map` / `Iterable` / `Queue`, lazy ops	🟢
7	07_null_safety.md	`?` / `!` / `??` / `??=` / `late`, sound null safety, flow analysis	🟢
8	08_enums.md	Basic & enhanced enums	🟢
9	09_records_and_patterns.md	Records, pattern matching, destructuring (Dart 3)	🟢
10	10_exception_handling.md	`throw` / `try` / `catch` / `on` / `finally`, custom exceptions	🟢

> Async, `Future`, `Stream`, the event loop, isolates, generics-in-depth, mixins, extensions, and the Dart VM/AOT/JIT pipeline are covered in **02 Advanced Dart** and **03 OOP**.

Prerequisites

None. This is the entry point. If you can install the Dart SDK and run `dart run file.dart`, you're ready.

What you'll be able to do after this module

- Choose `var` / `final` / `const` / `late` correctly and explain the difference to an interviewer.
- Read and write null-safe Dart fluently, including flow analysis and promotion.
- Use collections and their lazy `Iterable` methods idiomatically.
- Model data with records, enums, and patterns (Dart 3).
- Handle errors with typed, meaningful exceptions.

Capstones

Tier	Build
Beginner	A pure-Dart CLI **contact book** (add/list/search) using collections + null safety.
Intermediate	A **CSV report parser** using records, patterns, and exception handling.
Advanced	A tiny **expression evaluator** using enums + patterns.
Enterprise	A reusable `Result<T,E>` **+ validation library** (feeds Module 38).

Summary

Module 01 gives you the language substrate. Read the files in order; each is self-contained and follows the handbook template. Finish with the Revision Notes of every file before moving to Advanced Dart.

Variables & Mutability (`var`, `final`, `const`, `late`)

> A variable is a named reference to a value; the keyword you declare it with decides *when* the value is fixed and *whether* it can change.

Introduction

Dart gives you four ways to introduce a variable — `var`, `final`, `const`, and `late` — plus explicit type annotations. They differ along two axes: **when the value is determined** (compile time vs runtime) and **whether it can be reassigned**. Choosing correctly is the first mark of an idiomatic Dart developer.

Why this concept exists

Every language needs to bind names to values. Dart adds compile-time constants (`const`) and lazy/deferred initialization (`late`) because Flutter leans heavily on **immutability** for performance: `const` widgets are canonicalized and skipped during rebuilds, and `final` fields make objects predictable. The keywords exist to let the compiler prove and optimize what your code guarantees.

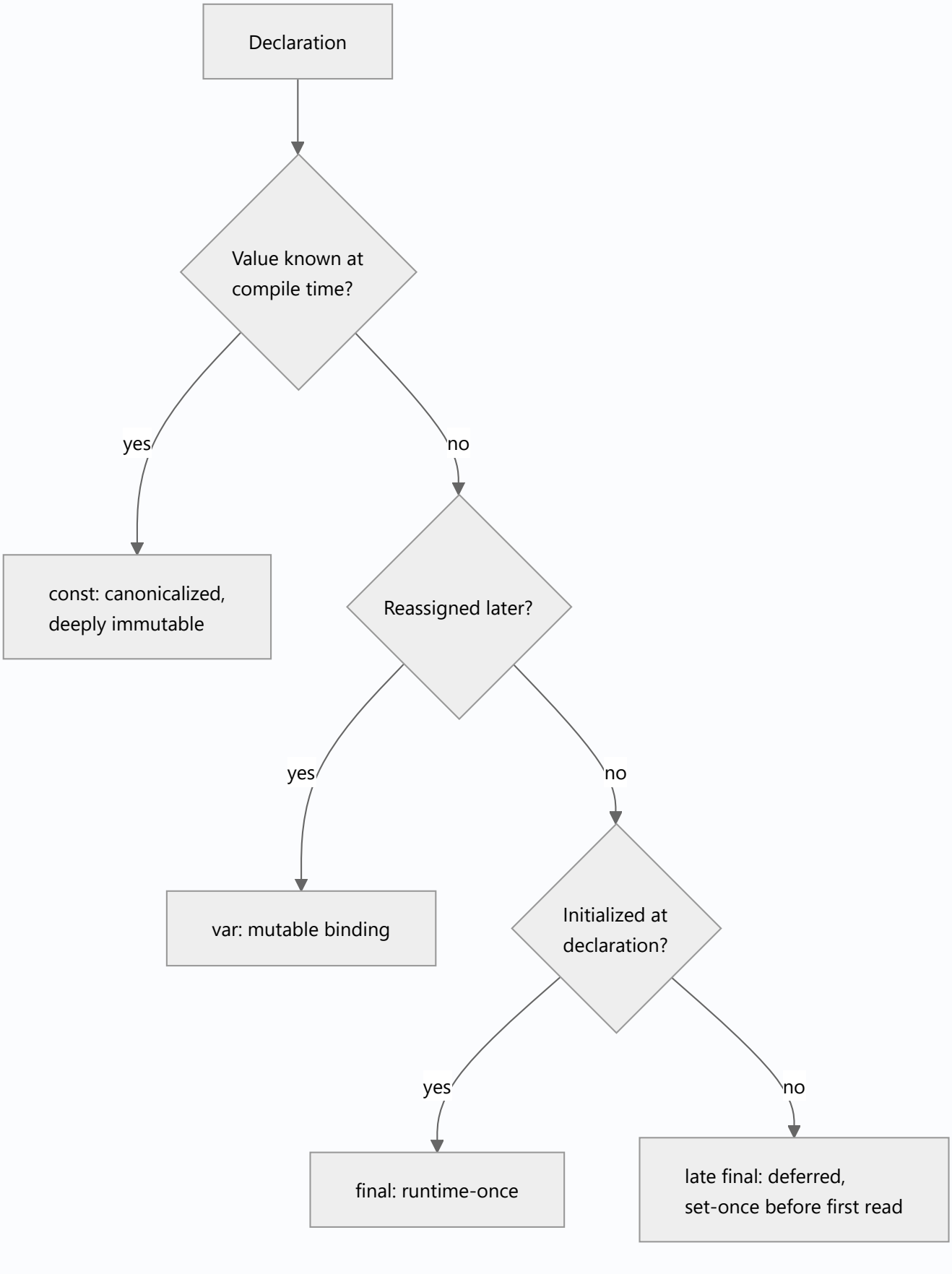
Real-world analogy

- `var` — a **whiteboard**: write a value, erase, write another (same kind of thing).
- `final` — a **signed contract**: filled in once at signing (runtime), never changed after.
- `const` — a value **carved into a printed textbook**: known before the book ships (compile time), identical copies everywhere are literally the same page.
- `late` — a **reserved parking spot**: the space exists now, the car arrives later; sit in it before the car arrives and you get an error.

Problem Statement

You must store: (a) a counter that changes, (b) a user's ID set once after login, (c) the mathematical constant `pi`, and (d) a configuration object that is expensive to build and may never be used. Which keyword for each? By the end you'll answer instantly: `var`, `final`, `const`, `late final`.

Internal Working

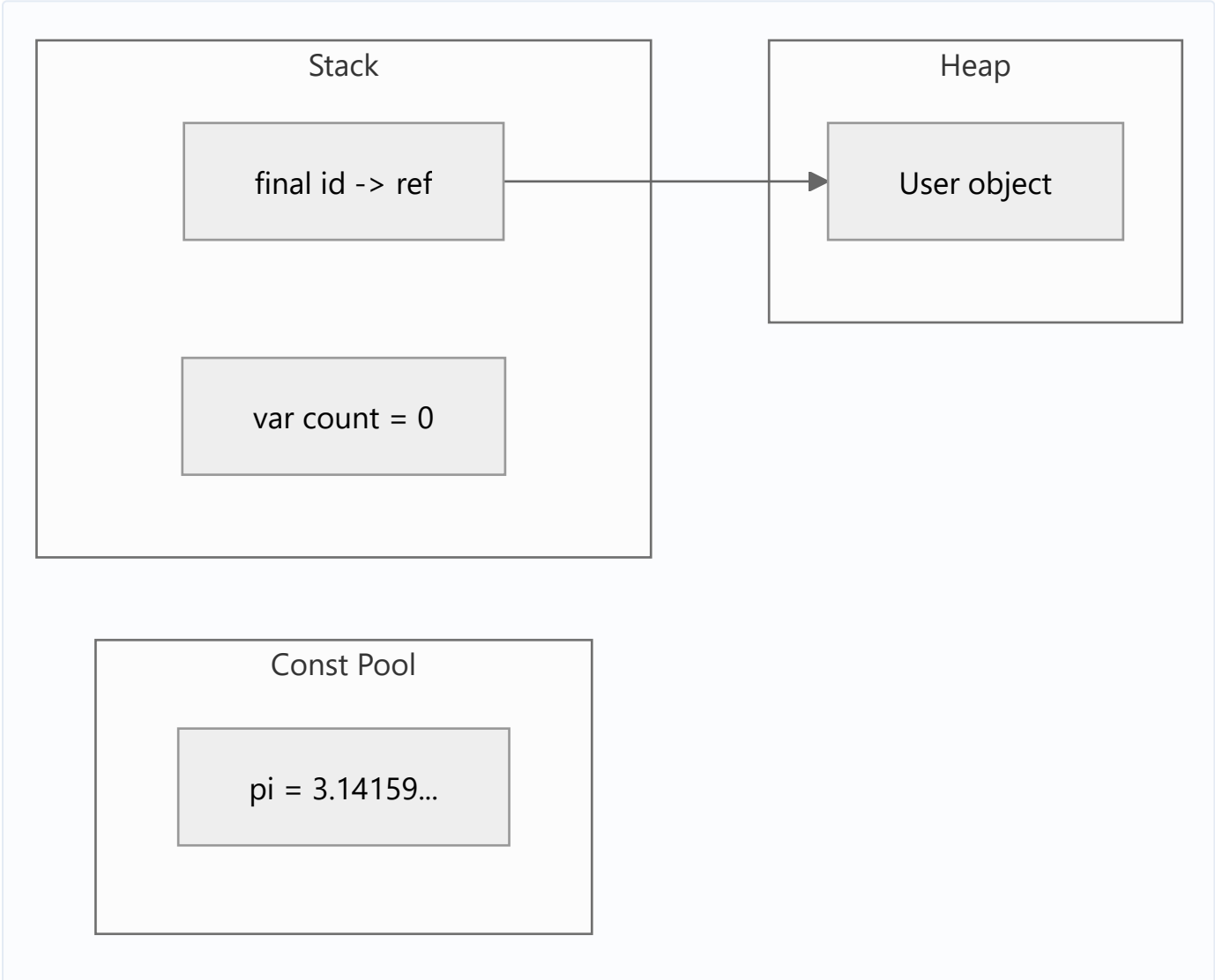


- `var` infers the type once from the initializer and locks that type. The binding is mutable.

- `final` allows exactly one assignment, at runtime. The *binding* is immutable; the *object* may still be mutable (`final list` can still `.add`).
- `const` requires a value computable at compile time. `const` objects are **canonicalized**: two structurally-equal const values are the *same instance* in memory (`identical()` is `true`).
- `late` defers initialization of a non-nullable variable, or makes a `final` field **lazily** initialized on first access.

Memory Representation

- `const` values live in a **canonicalized constant pool** — one copy shared across the whole program.
- `final` / `var` locals live on the stack (the reference); the object they point to lives on the heap.
- A `late` variable reserves its slot immediately but the backing store is only populated on first assignment; a lazy `late final x = expr()` stores a sentinel until first read, then caches the computed value.



Compiler Behavior

- `var x = 5;` → the compiler **infers** `int` and rejects `x = 'hi';`.
- `const` expressions are **evaluated by the compiler**; illegal (`const x = DateTime.now();`) is a compile-time error because the value isn't known until runtime.
- Const canonicalization is a compile-time optimization enabling Flutter to skip rebuilding `const` widget subtrees.
- Reassigning a `final` is a compile-time error, not a runtime one.

Runtime Behavior

- `final id = fetchId();` computes `fetchId()` at runtime and assigns once.
- Reading a `late` variable before assignment throws `LateInitializationError` at runtime.
- `late final config = expensiveSetup();` runs `expensiveSetup()` on **first access only**, then caches.

Flutter Engine Behavior

Not applicable directly — but `const` correctness is what lets the Flutter framework short-circuit widget rebuilds (see Module 21 Performance). A `const` widget instance is identical across builds, so the framework can skip it.

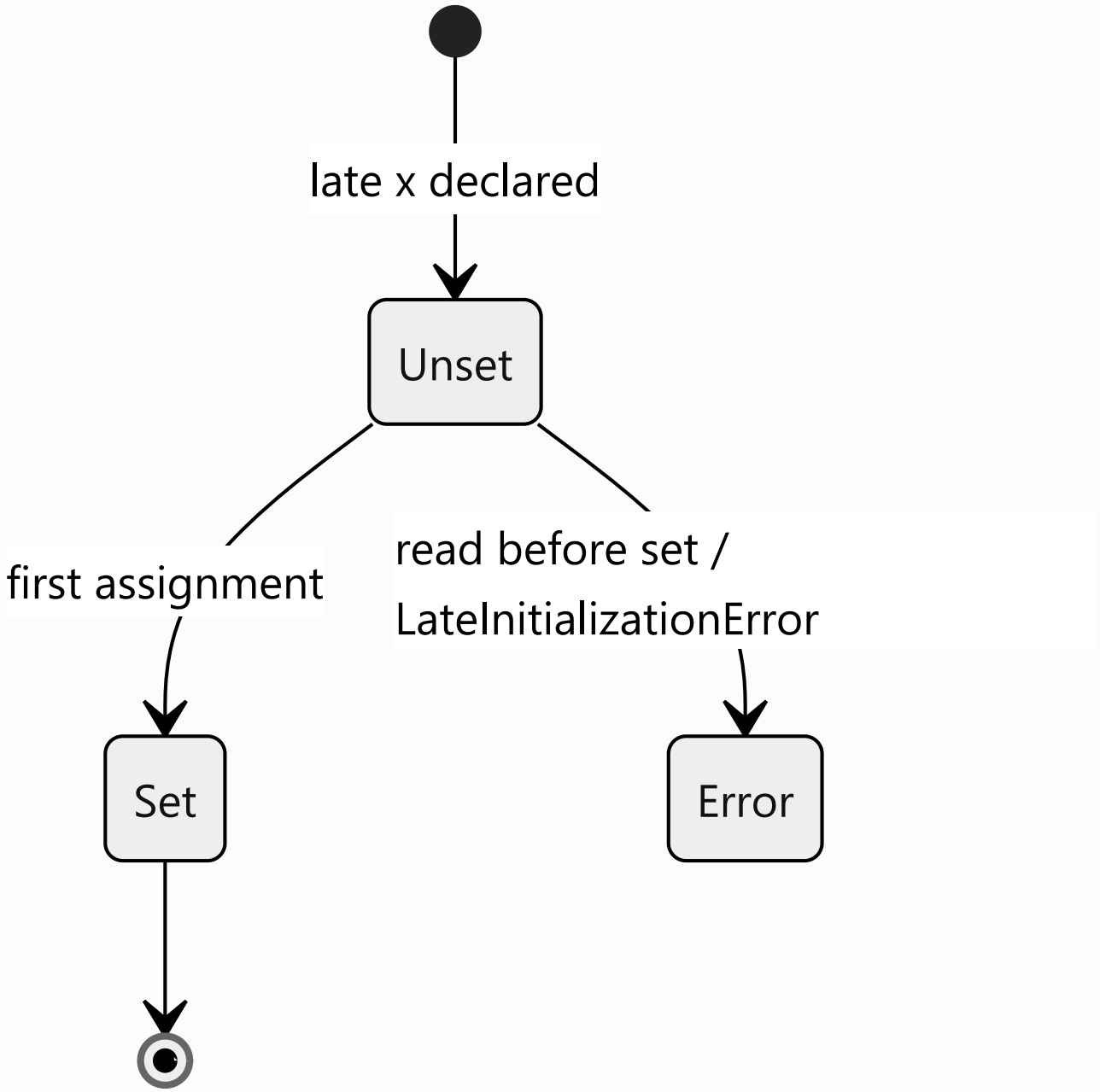
Dart VM Behavior

- In **JIT** (debug) the VM evaluates and canonicalizes consts during compilation to kernel; in **AOT** (release) consts are baked into the snapshot's constant table.
- `late` lowering: the VM generates a guard that checks an initialization flag on each read (lazy) or on the first read (deferred), throwing if unset.

Examples

```
void main() {
  // (a) changes -> var
  var count = 0;
  count = 1; // ok

  // (b) set once at runtime -> final
  final userId = _login(); // computed at runtime
  // userId = 'x'; // COMPILE ERROR: can't reassign a final

  // (c) compile-time constant -> const
  const pi = 3.1415926535;

  // (d) expensive, maybe-unused -> late final (lazy)
  late final config = _buildConfig(); // not run yet
  print(count + pi.floor()); // config still not built

  // final binding, mutable object:
  final items = <int>[1, 2];
  items.add(3); // OK — the LIST is mutable, the BINDING is fixed
  print(items); // [1, 2, 3]

  // const is deeply immutable + canonicalized:
  const a = [1, 2, 3];
  const b = [1, 2, 3];
  print(identical(a, b)); // true — same instance in memory
}

String _login() => 'user_42';
Map<String, Object> _buildConfig() => {'theme': 'dark'};
```

Diagrams


late x declared
Unset
first assignment
read before set /
LateInitializationError
Set
Error

Common Mistakes

Mistake	Why it's wrong	Fix
`const now = DateTime.now();`	Not compile-time	Use `final now = DateTime.now();`
Assuming `final list` is immutable	Only the binding is fixed	Use `const []` or `List.unmodifiable` for true immutability
Overusing `late` to silence null errors	Trades compile-time safety for runtime crashes	Prefer nullable `T?` when null is a valid state
`var` when the type should be explicit in a public API	Hurts readability	Annotate return/param types in public surfaces

Best Practices

- Prefer `final` **by default**; reach for `var` only when you genuinely reassign.
- Prefer `const` wherever the value is compile-time known (especially widgets).
- Use `late final x = ...` for expensive, lazily-computed, set-once values.
- Reserve `late` (non-final) for framework patterns like `initState`-assigned fields.

Performance

- `const` eliminates repeated allocation and enables widget-rebuild skipping — measurable in large trees.
- `late` lazy init avoids paying for values you never use, but adds a per-read init check (negligible but non-zero).

Advantages

- Explicit intent: readers know instantly whether a value can change.
- Compiler can prove immutability and optimize (canonicalization, const folding).

Disadvantages

- `late` moves a class of errors from compile time to runtime.
- `const` has strict requirements (all inputs must themselves be const).

Interview Questions

1. 🟢 **Difference between `final` and `const`?** — `final` = single assignment, value computed at **runtime**. `const` = **compile-time** constant, deeply immutable, and **canonicalized** (equal consts share one instance).

2. 🟢 **Is `final` immutability deep or shallow?** — Shallow: the binding can't be reassigned, but the referenced object can still mutate (`final list.add(x)` works).
3. 🟡 **What does `late` do, and what are its two uses?** — (1) Deferred init of a non-nullable variable (set before first read); (2) lazy init of a `final` (runs initializer on first access, then caches).
4. 🟡 **What happens if you read a `late` variable before assigning it?** — Throws `LateInitializationError` at runtime.
5. 🟡 **Why does `identical(const [1,2], const [1,2])` return `true`?** — Const canonicalization: the compiler stores one shared instance for structurally-equal const values.
6. 🔴 **How does `const` help Flutter performance?** — `const` widgets are identical across rebuilds, so the framework can skip re-creating and re-diffing that subtree.
7. 🔴 **Can a `const` constructor produce non-const instances?** — Yes; a class with a `const` constructor can be instantiated with or without `const`. Only the `const` invocation is canonicalized.

Senior Engineer Tips

- "Prefer `final`" isn't dogma — it's about **local reasoning**: fewer mutable bindings means fewer states to track when debugging.
- Watch for `const` propagation: making a leaf widget `const` often unlocks `const` on its parents, compounding rebuild savings.
- `late` on instance fields is a code smell if overused; it usually signals a constructor or nullable type would be cleaner.

Architect Perspective

Immutability is an architectural lever, not a micro-optimization. A codebase that defaults to `final` / `const` and immutable models is dramatically easier to make reactive, testable, and concurrency-safe (immutable data can cross isolates without defensive copies). Enforce it via lints (`prefer_final_locals` , `prefer_const_constructors`).

Summary

- `var` = mutable, inferred-once. `final` = runtime, set-once (shallow immutable). `const` = compile-time, deeply immutable, canonicalized. `late` = deferred/lazy non-nullable init.
- Default to `final` , upgrade to `const` when possible, use `late` deliberately.

Revision Notes

- `final` = runtime-once; `const` = compile-time + canonicalized + deeply immutable.

- `final list.add()` works (binding fixed, object mutable).
- `late` read-before-write → `LateInitializationError`.
- `const` widgets → skipped rebuilds.
- Lint: `prefer_final_locals`, `prefer_const_constructors`.

Practice Questions

1. Explain why `const config = {'t': DateTime.now()}` fails to compile.
2. Give a case where `late final` is strictly better than `final`.
3. When is `var` more readable than a type annotation, and when is it worse?

Coding Questions

1. Write a class `Circle` with a `const` constructor and a computed `area` getter; prove two `const Circle(1)` are `identical`.
2. Implement a lazily-initialized `Logger` singleton field using `late final`.
3. Given `final ids = <int>[]`, show two lines: one that compiles (mutation) and one that doesn't (reassignment), with comments.

Mini Project

Config loader: Build a pure-Dart `AppConfig` class with `const` defaults, a `late final` lazily-loaded `overrides` map (simulate an expensive load), and `final` resolved values. Print which fields came from defaults vs overrides. Acceptance: no field is `var`; `dart analyze` is clean; the expensive load runs only if overrides are accessed.

Data Types (`num`, `int`, `double`, `String`, `bool`, `dynamic`, `Object?`)

> Every value in Dart is an object with a type; the type system decides what you can do with a value and when mistakes are caught.

Introduction

Dart is **statically typed with type inference**, and **everything is an object** — including numbers, booleans, functions, and `null`. This file covers the core built-in types, the `num` / `int` / `double` hierarchy, and the crucial distinction between `dynamic` (opt-out of checking) and `Object?` (safe top type).

Why this concept exists

A type system exists to catch errors **before** the program runs and to enable the compiler to generate efficient code. Dart's "everything is an object" model gives a uniform mental model (no primitive/object split like Java's `int` vs `Integer`), while `dynamic` and `Object?` give controlled escape hatches for genuinely dynamic data (like decoded JSON).

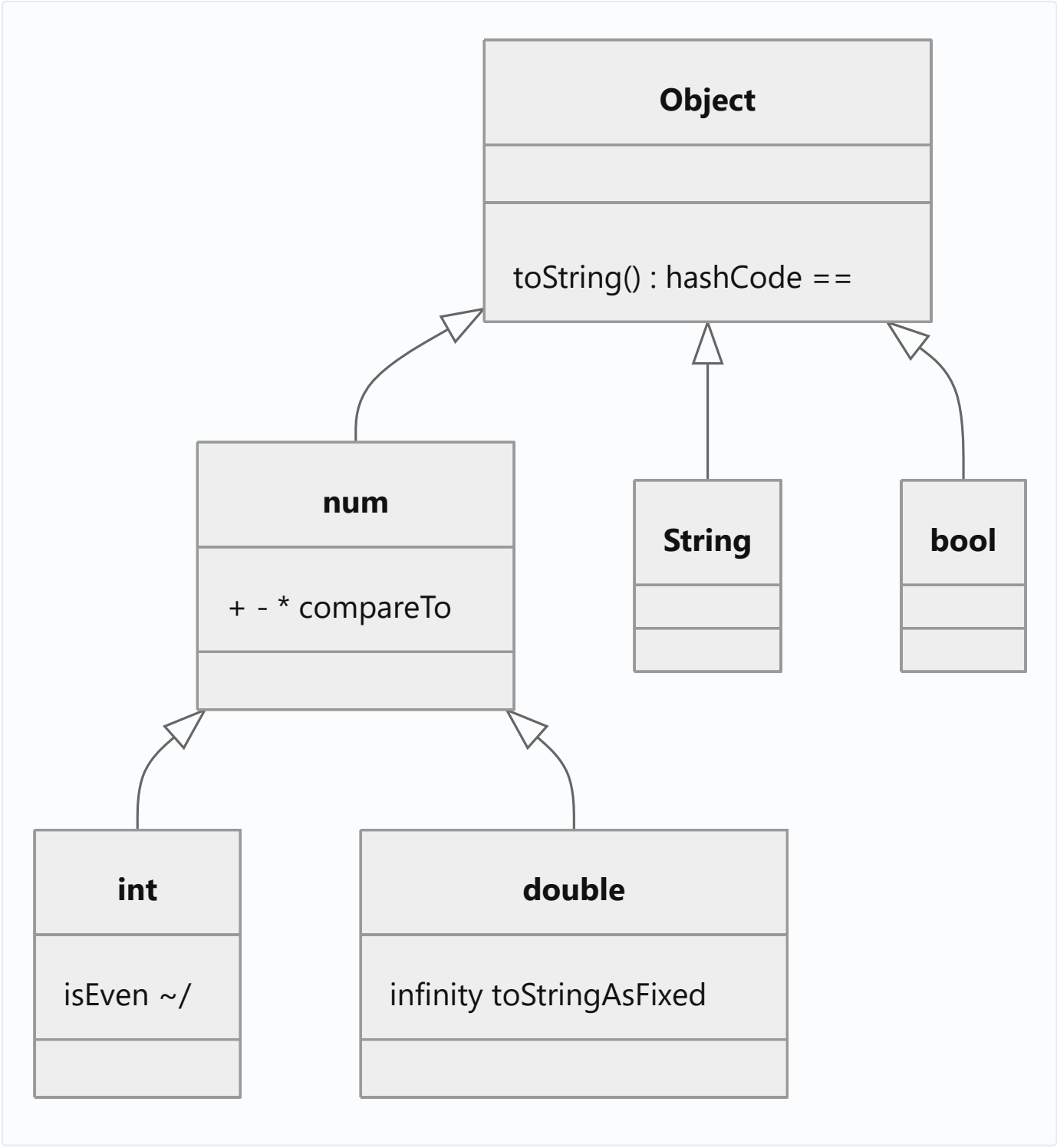
Real-world analogy

Types are **labeled shipping containers**. A container labeled `int` can only hold whole numbers; the port (compiler) rejects a mislabeled load before the ship sails (runtime). `dynamic` is an **unlabeled container** — anything fits, but nobody checks until you open it at sea (runtime crash). `Object?` is a container labeled "some object, contents unknown" — you must inspect before using the contents.

Problem Statement

You decode JSON into a `Map<String, dynamic>`. You need the `"age"` field as an `int` and the `"name"` as a `String`, safely, without runtime crashes. By the end you'll know why the map is `dynamic`-valued and how to narrow safely with `is`/casts.

Internal Working



- `num` is the supertype of `int` and `double`. `num x = 5; x = 5.5;` is legal.
- `int` is an arbitrary-precision-ish integer (64-bit on native; on web it's a JS `double`-backed integer — a portability gotcha).
- `String` is an immutable sequence of UTF-16 code units.
- `bool` has exactly two instances: `true`, `false`.
- `dynamic` disables static checking. `Object?` is the top type: everything is an `Object?`.

Memory Representation

- Small `int`s and `double`s may be represented efficiently (boxed/unboxed) by the VM; conceptually they're heap objects but the VM optimizes common cases.
- `String` is immutable — concatenation creates a new object; heavy building should use `StringBuffer`.
- `null` is a single canonical object (`Null` type).

Compiler Behavior

- Type inference: `var n = 5` → `int`; `var d = 5.0` → `double`; `var s = 'x'` → `String`.
- Calling a member the static type doesn't have is a **compile error** for `Object?` but **allowed** for `dynamic` (deferred to runtime).
- Numeric literals: `5` is `int`, `5.0` is `double`; `double d = 5;` is allowed (int literal coerced to double in a double context).

Runtime Behavior

- `dynamic value = 5; value.length;` compiles, then throws `NoSuchMethodError` at runtime.
- Type checks (`is`) and casts (`as`) are evaluated at runtime; a failing `as` throws `TypeError`.
- Integer division `~/` returns `int`; `/` always returns `double`.

Flutter Engine Behavior

Not applicable — data types are a pure language concern.

Dart VM Behavior

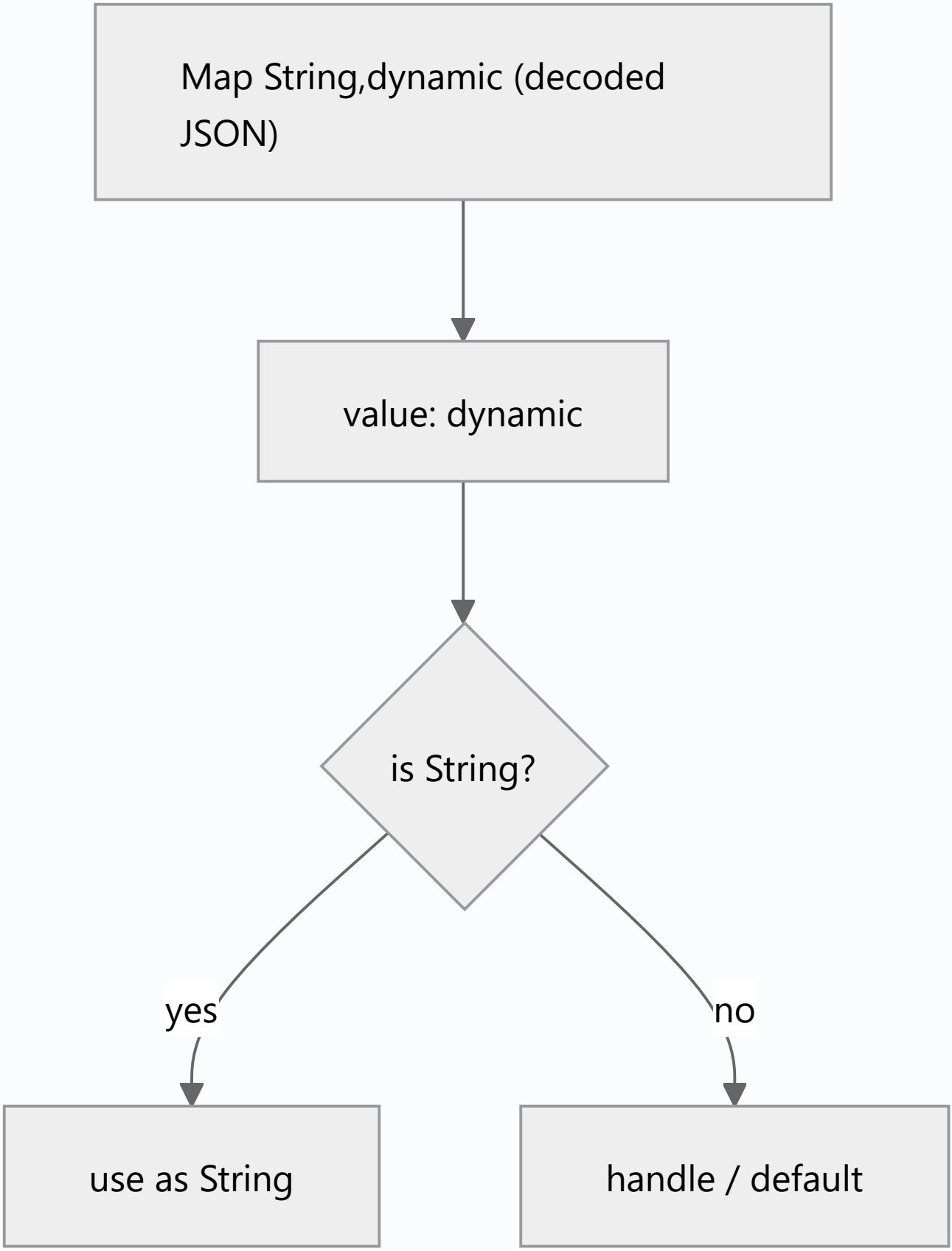
- The VM specializes numeric operations; in AOT, monomorphic numeric code is highly optimized.
- **Web caveat:** on `dart2js` / `dartdevc`, `int` and `double` share JavaScript's IEEE-754 double, so very large integers lose precision. Native (mobile/desktop) uses true 64-bit ints.

Examples

```
void main() {
  // num holds both:
  num x = 5;      // int
  x = 5.5;        // now double — legal because both are num
  print(x is double); // true
```

```dart
  // int vs double division:
  print(7 / 2);   // 3.5   (double)
  print(7 ~/ 2);  // 3     (int)
  print(7 % 2);   // 1

  // String is immutable UTF-16:
  const name = 'Dart';
  print(name.length);            // 4
  print(name.toUpperCase());     // DART  (new string)

  // dynamic vs Object? safety:
  dynamic d = 'hello';
  print(d.length);   // 5 — no compile check; would crash if d were an int

  Object? o = 'hello';
  // print(o.length); // COMPILE ERROR: 'length' not defined for Object
  if (o is String) {
    print(o.length); // 5 — promoted to String inside the check
  }

  // Safe JSON narrowing:
  final json = <String, dynamic>{'name': 'Ada', 'age': 36};
  final ageName = json['name'];         // dynamic
  final age = json['age'] as int;       // explicit cast
  final safeName = json['name'] as String? ?? 'unknown';
  print('$safeName is $age'); // Ada is 36
}
```

Diagrams


Map String,dynamic (decoded JSON)
value: dynamic
is String?
yes
no
use as String
handle / default

Common Mistakes

Mistake	Why it's wrong	Fix
Using `dynamic` for JSON everywhere	Loses all safety	Cast at the boundary; model with classes
`int` assumptions on web	Precision loss with big ints	Use `BigInt` or strings for large IDs on web
Building strings with `+=` in loops	$O(n^2)$ allocations	Use `StringBuffer`
`as` without a null check	Throws on mismatch	Use `as T?` + `??`, or `is` promotion

Best Practices

- Keep `dynamic` at the **edges** (deserialization) and convert to typed models immediately.
- Prefer `Object?` over `dynamic` when you want "anything, but safely."
- Use `num` only when a value is genuinely int-or-double; otherwise be specific.
- Format money/precision with `toStringAsFixed`, never rely on `double` for exact currency math (use integers of the smallest unit, e.g. paise/cents).

Performance

- `StringBuffer` turns repeated concatenation from $O(n^2)$ to $O(n)$.
- Monomorphic numeric code (always `int`, or always `double`) is faster than mixed `num`.

Advantages

- Uniform object model — no primitive/wrapper split.
- Strong inference keeps code concise while type-safe.
- Escape hatches (`dynamic`, `Object?`) for real-world dynamic data.

Disadvantages

- `dynamic` can silently defer bugs to runtime.
- Web numeric semantics differ from native — a portability footgun.

Interview Questions

1. 🟢 **What's the supertype of `int` and `double`?** — `num`. Both extend it.
2. 🟢 **Difference between `/` and `~/`?** — `/` returns a `double` always; `~/` is integer (truncating) division returning `int`.
3. 🟡 **`dynamic` vs `Object?` — which is safer and why?** — `Object?`. Both hold anything, but `dynamic` allows *any* member call (unchecked, may crash), while `Object?` only allows `Object`

members until you check/cast — so mistakes are caught at compile time.

4. 🟡 **Why is `String` concatenation in a loop slow?** — Strings are immutable; each `+=` allocates a new string. Use `StringBuffer`.
5. 🟡 **Is `double d = 5;` legal?** — Yes; an `int` literal is coerced to `double` in a double context. But `int i = 5.0;` is not.
6. 🔴 **What breaks about `int` on Flutter Web?** — Web ints are JS doubles (IEEE-754); integers beyond 2^53 lose precision. Use `BigInt`/strings for large values.
7. 🔴 **Why avoid `double` for currency?** — Binary floating point can't represent decimals like 0.1 exactly, causing rounding errors. Store the smallest integer unit instead.

Senior Engineer Tips

- Treat the deserialization boundary as a **type firewall**: `dynamic` in, typed models out, with validation.
- Prefer exhaustive modeling (sealed classes/enums) over `dynamic` flags — it moves errors to compile time.
- Remember web numeric semantics when building cross-platform apps with large numeric IDs.

Architect Perspective

Type discipline at boundaries (API, DB, platform channels) is a system-wide reliability decision. Centralize parsing/validation so the rest of the codebase never touches `dynamic`. This is the foundation for the `Result`/failure modeling in Module 38 and DTO↔entity mapping in Clean Architecture (Module 40).

Summary

- Everything is an object; `num` → `int`/`double`; `String` is immutable UTF-16.
- `dynamic` opts out of checking (risky); `Object?` is the safe top type.
- Narrow `dynamic` at the edges; beware web int precision and `double` currency math.

Revision Notes

- `num` supertype of `int`/`double`. `/` →double, `~/` →int.
- `dynamic` = no checks (runtime crash); `Object?` = safe, must check/cast.
- `String` immutable → `StringBuffer` for building.
- Web `int` = JS double (precision loss); currency → integer smallest unit.

Practice Questions

1. Predict the type: `var a = 5; var b = 5.0; var c = 5 / 5; var d = 5 ~/ 5;`.
2. Why does `Object? o = 5; o.isEven;` fail to compile but `dynamic d = 5; d.isEven;` compile?
3. Explain the web precision issue with a concrete large-int example.

Coding Questions

1. Write `T? tryCast<T>(Object? value)` returning `value as T?` safely (null on mismatch).
2. Parse `{'price': '19.99', 'qty': '3'}` into a total **in cents** as an `int`, avoiding `double` currency errors.
3. Benchmark `+=` vs `StringBuffer` building a 100k-char string; print both durations.

Mini Project

Typed JSON firewall: Given a raw `Map<String, dynamic>` for a `User` (`id`, `name`, `email`, optional `age`), write a `User.fromJson` that casts/validates every field, throws a descriptive `FormatException` on bad data, and exposes only typed fields. Acceptance: no `dynamic` escapes the class; invalid input produces a clear error; `dart analyze` clean.

Operators (Arithmetic, Logical, Null-aware, Cascade, Spread)

> Operators are just method calls in disguise — which is why you can override most of them on your own types.

Introduction

Dart operators cover arithmetic, equality/relational, logical, type-test, assignment, null-aware, cascade (`..`), and spread (`...`). Crucially, most operators are **instance methods** you can override, and several (null-aware, cascade, spread) are ergonomics that show up constantly in Flutter widget trees.

Why this concept exists

Operators give concise, readable syntax for common operations, and making them overridable lets your value types (`Money` , `Vector`) behave like built-ins. Cascade and spread exist specifically to make **declarative object/collection construction** (Flutter's bread and butter) terse and readable.

Real-world analogy

Cascade (`..`) is a **barista making one drink**: pour, add milk, add foam — all to the *same cup*, without re-naming the cup each step. Spread (`...`) is **emptying one bag of groceries into another** in a single motion.

Problem Statement

Build a widget's children list that always includes a header, conditionally includes a banner, and expands a variable list of items — in one literal. And configure a `Paint` object with several properties without repeating its name. You'll use collection- `if` / `for` , spread, and cascade.

Internal Working

Category	Operators
Arithmetic	`+ - * / ~/ % -expr`
Relational/Equality	`== != < > <= >=`
Logical	`&&
Type test	`is is! as`
Assignment	`= += -= ??= ...`
Null-aware	`?. ?? ??= ...? ?..`
Other	`..` (cascade), `...` (spread), `[]` / `[]=` (index)

- `a + b` compiles to `a.+(b)` — operators are methods; override with `operator +`.
- `==` defaults to identity; override **with** `hashCode` for value equality.
- Cascade `..` returns the **receiver**, not the method result.
- Spread `...` inlines an iterable's elements into a collection literal; `...?` skips a null iterable.

Memory Representation

Not applicable beyond ordinary object semantics — operators don't have special storage. Cascade/spread produce ordinary objects/collections.

Compiler Behavior

- Operator calls are resolved to the receiver's method; unknown operator on a static type is a compile error.
- Collection-`if` / `for` and spread are compiled into efficient element-adding code (no intermediate temp list for spread of a known iterable).
- `a ?? b` : `b` is only evaluated if `a` is null (short-circuit).

Runtime Behavior

- `&&` / `||` short-circuit: the right side runs only if needed.
- `as` throws `TypeError` on failure at runtime; `is` returns bool.
- `~/` and `%` follow integer semantics; `/` yields double.

Flutter Engine Behavior

Not applicable.

Dart VM Behavior

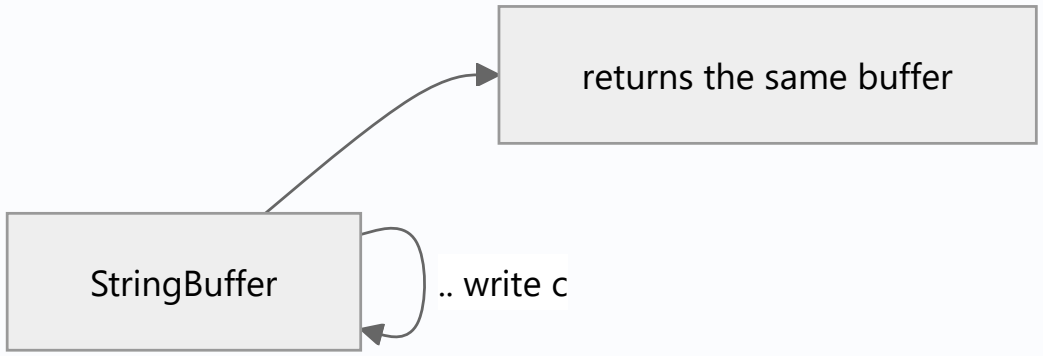
- Overloaded operators on hot value types (e.g., `Offset + Offset` in animations) are inlined by the AOT compiler when monomorphic.

Examples

```
class Money {
  final int cents;
  const Money(this.cents);
  Money operator +(Money o) => Money(cents + o.cents);
  @override
  bool operator ==(Object o) => o is Money && o.cents == cents;
  @override
  int get hashCode => cents.hashCode;
  @override
  String toString() => '\$${(cents / 100).toStringAsFixed(2)}';
}

void main() {
  print(Money(150) + Money(250)); // $4.00
  print(Money(100) == Money(100)); // true (value equality)

  // null-aware
  int? n;
  print(n ?? 0);   // 0
  print(n?.isEven); // null

  // cascade — configure one object
  final buffer = StringBuffer()
    ..write('a')
    ..write('b')
    ..write('c');
  print(buffer); // abc

  // spread + collection-if/for (very Flutter-like)
  final extra = [2, 3];
  const showZero = true;
  final list = [
    if (showZero) 0,
```

```
    1,
    ...extra,
    for (var i = 4; i < 6; i++) i,
  ];
  print(list); // [0, 1, 2, 3, 4, 5]

  // null-aware spread
  List<int>? maybeNull;
  final safe = [0, ...?maybeNull];
  print(safe); // [0]
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Override `==` without `hashCode`	Breaks Set/Map keys	Always override both together
Expecting `..` to return the method's result	It returns the receiver	Use a normal call if you need the result
`as` without handling failure	Throws	Prefer `is` promotion or `as T?` + `??`
Forgetting `...?` for nullable spreads	Crashes on null	Use `...?`

Best Practices

- Override `==` / `hashCode` (or use `records` / `equatable` / `freezed`) for value types.
- Use cascades to build/configure objects fluently.
- Use collection-`if` / `for` /spread in widget `children` for declarative UI.

Performance

- Spread of a large known list is O(n) copy; prefer building directly if you can avoid intermediate lists in hot paths.
- Short-circuit operators avoid unnecessary work — order cheap conditions first.

Advantages / Disadvantages

- **+** Concise, readable, extensible to custom types; declarative construction.
- **−** Operator overloading can obscure cost if abused (a "cheap-looking" `+` doing heavy work).

Interview Questions

1. 🟢 **Are operators methods in Dart?** — Yes; `a + b` is `a.+(b)`, and most are overridable via `operator`.
2. 🟢 **What does cascade `..` return?** — The receiver object, not the invoked method's result.
3. 🟡 **Why override `hashCode` with `==`?** — Hash-based collections use `hashCode` to bucket and `==` to confirm; overriding one without the other breaks `Set` / `Map` semantics.
4. 🟡 **`is` vs `as`?** — `is` tests type (returns bool, enables promotion); `as` casts (throws on mismatch).
5. 🔴 **How do collection-`if`/spread compile?** — Into element-adding operations within the literal, avoiding manual `add` / `addAll` boilerplate; the compiler emits efficient code.
6. 🔴 **When is operator overloading a bad idea?** — When it hides significant cost or surprises readers (non-intuitive semantics); explicitness beats cleverness.

Senior Engineer Tips

- Reach for `records` (Dart 3) or `freezed` / `equatable` instead of hand-writing `==` / `hashCode` across many models.
- Cascades shine in builders and test setup; don't overuse where a local variable reads clearer.

Architect Perspective

Value equality is foundational for reactive state: frameworks compare old/new state to decide rebuilds. Consistent `==` / `hashCode` (or immutable value types) makes state diffing correct and cheap — a prerequisite for `Bloc` / `Riverpod` selectors (Module 11).

Summary

- Operators are overridable methods; override `==` with `hashCode`.

- Null-aware (`?.` / `??` / `??=` / `...?`), cascade (`..`), and spread (`...`) power concise, declarative code.
- Short-circuiting and integer/double division semantics matter for correctness.

Revision Notes

- `a + b` == `a.+(b)` ; override `==` + `hashCode` together.
- `..` returns receiver; `...` / `...?` spread; collection- `if` / `for` in literals.
- `/` →double, `~/` →int; `&&` / `||` short-circuit; `as` throws, `is` promotes.

Practice Questions

1. Why does a `Set<Money>` misbehave if you override only `==` ?
2. Rewrite a `children` list built with `add` / `addAll` using spread + collection- `if` .
3. What's the difference between `?.` and `..` — one word each?

Coding Questions

1. Implement `Vector2(x, y)` with `+` , `-` , `*` (scalar), `==` , `hashCode` , `toString` .
2. Build a widget-like `Column` 's children list mixing header, optional error, and a spread of items.
3. Write `T pipe<T>(T seed, List<T Function(T)> steps)` and configure an object via cascade inside it.

Mini Project

Immutable geometry library: Implement `Point` , `Size` , `Rect` value types with overloaded operators, correct value equality, and cascade-friendly builders. Add tests proving equality and Set/Map key correctness. Acceptance: no mutable state; `dart analyze` clean; equality tests pass.

Control Flow (`if`, `for`, `while`, `switch`, collection-`if`/`for`)

> Control flow decides *which* code runs and *how often* — and in Dart 3, `switch` graduated from a statement into a powerful pattern-matching expression.

Introduction

This file covers conditionals (`if`/`else`), loops (`for`, `for-in`, `while`, `do-while`), `break`/`continue`, the modern `switch` (statement **and** expression), and collection-`if`/`for` used inside literals. Dart 3's pattern-enabled `switch` is a major upgrade covered here at an introductory level (full treatment in 09_records_and_patterns.md).

Why this concept exists

Programs must branch and repeat. Dart modernized `switch` into an **exhaustive expression** because exhaustiveness (the compiler forcing you to handle every case) turns whole categories of logic bugs into compile errors — especially valuable when switching over sealed types/enums that model app state.

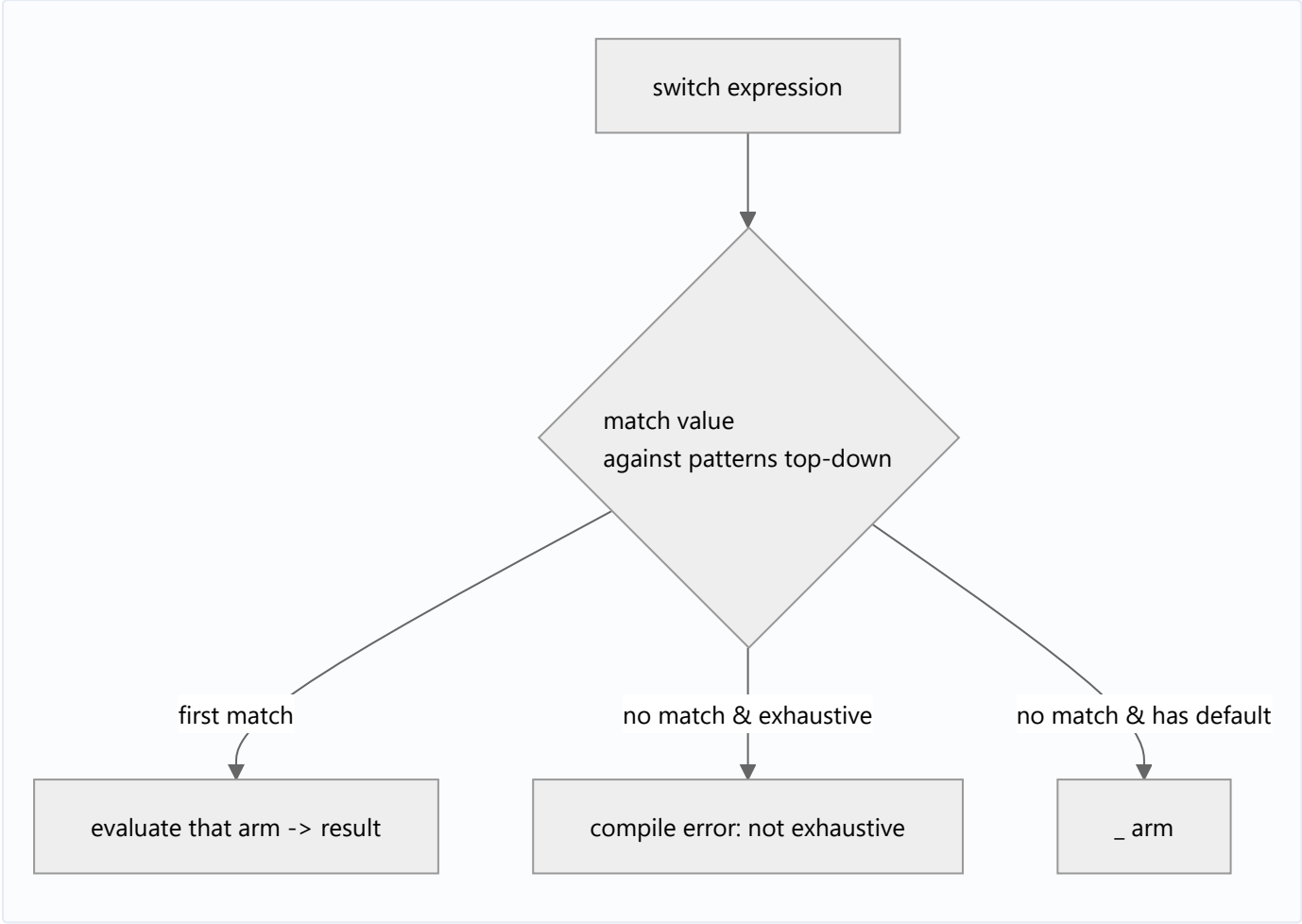
Real-world analogy

`if`/`else` is a **fork in the road**; a loop is a **lap around a track**; `switch` is a **mail sorter** dropping each letter into exactly one labeled bin — and exhaustive switch means the sorter refuses to operate until every possible label has a bin.

Problem Statement

Given an `enum Status { loading, success, error }`, render a different message for each, with the compiler guaranteeing you didn't forget a case. And build a list that conditionally includes items. You'll use a `switch` expression and collection-`if`.

Internal Working



- `if (cond) {...} else {...}` — `cond` must be `bool` (no truthy/falsy coercion).
- `for (final x in iterable)` gives a **fresh binding** per iteration (safe for closures).
- `switch` **expression** returns a value: `final msg = switch (x) { ... };`.
- **Exhaustiveness:** switching over an `enum` or sealed hierarchy without covering all cases (and no `_`) is a compile error.
- Collection- `if` / `for` build elements conditionally/repeatedly inside `[]` / `{}`.

Memory Representation

Not applicable — control flow is code structure, not data. (Loop variables follow normal stack/heap rules; `for-in` fresh bindings avoid the classic closure-capture bug.)

Compiler Behavior

- `switch` expressions are checked for **exhaustiveness**; missing cases are compile errors.
- Unreachable code after `return` / `break` is flagged.
- The old fall-through rule: statement `case`s don't fall through (each needs `break` / `return` / `continue`), unlike C.

Runtime Behavior

- Loops evaluate their condition each iteration; `break` exits, `continue` skips to next.
- A non-exhaustive `switch` *statement* with no matching case and no `default` simply does nothing (no throw), which is why expressions (exhaustive) are safer.

Flutter Engine Behavior

Not applicable. (But `switch` expressions over state enums are the idiomatic way to map state → widget in `build`.)

Dart VM Behavior

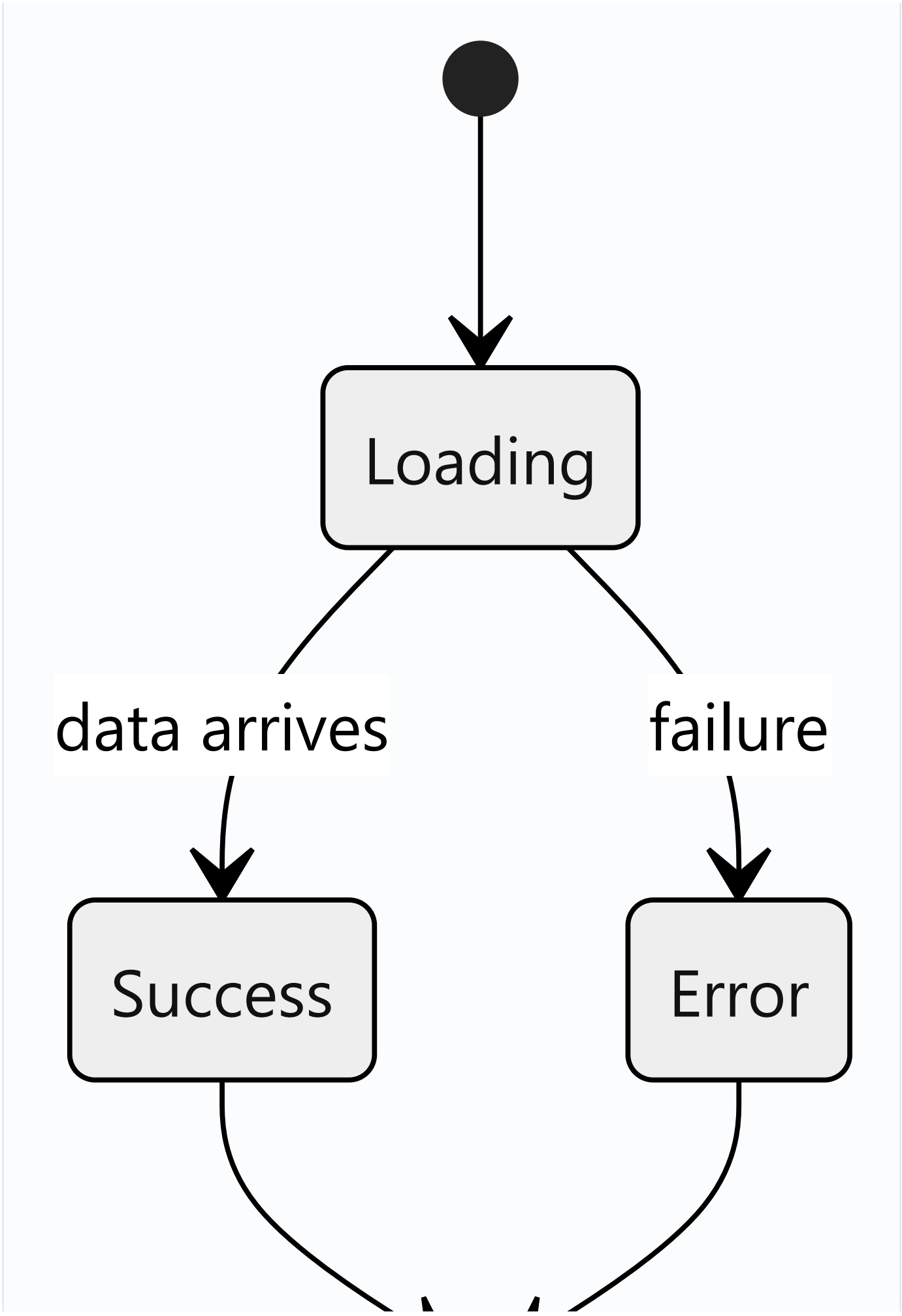
- Dense integer/enum switches may be compiled to jump tables for O(1) dispatch; sparse ones become comparisons.

Examples

```
enum Status { loading, success, error }


String message(Status s) => switch (s) {
      Status.loading => 'Please wait…',
      Status.success => 'Done!',
      Status.error => 'Something went wrong',
    }; // exhaustive — remove one arm and it won't compile


void main() {
  print(message(Status.success)); // Done!


  // for-in with fresh binding (closure-safe)
  final actions = <void Function()>[];
  for (final i in [1, 2, 3]) {
    actions.add(() => print(i));
  }
  for (final a in actions) a(); // 1, 2, 3 (not 3,3,3)


  // classic for + break/continue
  for (var i = 0; i < 5; i++) {
    if (i == 1) continue;
    if (i == 4) break;
    print(i); // 0, 2, 3
```

```
  }

  // while
  var n = 3;
  while (n > 0) {
    n--;
  }

  // collection-if / collection-for
  const isAdmin = true;
  final menu = [
    'Home',
    if (isAdmin) 'Admin',
    for (final p in ['A', 'B']) 'Page $p',
  ];
  print(menu); // [Home, Admin, Page A, Page B]

  // switch with guard (when)
  final int score = 85;
  final grade = switch (score) {
    >= 90 => 'A',
    >= 80 => 'B',
    _ => 'C',
  };
  print(grade); // B
}
```

Diagrams


Loading
data arrives
failure
Success
Error

Common Mistakes

Mistake	Why	Fix
Using a non-bool condition	Dart has no truthy/falsy	Compare explicitly: `if (list.isNotEmpty)`
Capturing a C-style loop index in a closure	Shares one variable	Use `for (final x in ...)`
Relying on `switch` statement to catch all cases	Statements aren't forced exhaustive	Use `switch` **expression** over enums/sealed
Adding a `default` / `_` on an enum switch unnecessarily	Defeats exhaustiveness checking	Omit `_`; let the compiler enforce new cases

Best Practices

- Prefer `switch` **expressions** over enums/sealed types and **omit** `_` so adding a new variant forces you to handle it.
- Use `for-in` over index loops unless you need the index.
- Keep loop bodies small; extract complex bodies into named functions.
- Use collection-`if` / `for` in widget `children` instead of building lists imperatively.

Performance

- `switch` over dense ints/enums → jump table (fast).
- Avoid heavy work inside tight loops; hoist invariants out.

Advantages / Disadvantages

- **+** Exhaustive `switch` moves bugs to compile time; collection-`if` / `for` keep UI declarative.
- **−** Pattern `switch` has a learning curve; deeply nested control flow hurts readability (refactor to functions).

Interview Questions

1. 🟢 **Does Dart have truthy/falsy?** — No; conditions must be `bool`.

2. 🟢 `switch` **statement vs expression?** — Statement runs side effects; expression returns a value and is checked for exhaustiveness.
3. 🟡 **What is exhaustiveness and why does it matter?** — The compiler forces every case of an enum/sealed type to be handled; adding a new variant then becomes a compile error until you handle it — catching bugs early.
4. 🟡 **Why does `for-in` avoid the closure-capture bug?** — Each iteration gets a fresh binding, so captured closures see distinct values.
5. 🔴 **How is a `switch` compiled for dense integer cases?** — Often as a jump table for O(1) dispatch.
6. 🔴 **When should you NOT add `default` / `_` to a switch?** — Over sealed types/enums, so you lose exhaustiveness protection when new variants appear.

Senior Engineer Tips

- Model UI state as a sealed type and `switch` on it in `build` — the compiler becomes your reviewer for "did you handle every state?"
- Guards (`case x when cond`) express complex branching without nested `if` s.

Architect Perspective

Exhaustive switching over sealed state types is the backbone of predictable, self-documenting state machines — the same idea powering BLoC state handling and DDD domain events. It shifts "did we handle every case?" from code review to the compiler, which scales far better across a large team.

Summary

- Conditions are strictly `bool` ; loops favor `for-in` ; `switch` is now an exhaustive expression.
- Collection- `if` / `for` /spread build declarative literals.
- Omit `_` on enum/sealed switches to keep exhaustiveness enforcement.

Revision Notes

- No truthy/falsy — conditions must be `bool` .
- `switch` expression returns a value + exhaustive; omit `_` on enums.
- `for-in` = fresh binding (closure-safe); jump table for dense switches.
- Collection- `if` / `for` in `[]` / `{}` .

Practice Questions

1. Why does the C-style `for` closure print `3,3,3` while `for-in` prints `1,2,3`?
2. When is a `switch` expression strictly safer than an `if` / `else` chain?
3. Give a UI case where omitting `_` in a switch prevents a future bug.

Coding Questions

1. Write `String httpCategory(int code)` using a `switch` expression with guards (`2xx` →success, etc.).
2. Build a settings menu list using collection-`if` / `for` based on a `Set<Permission>`.
3. Implement a tiny state machine `TrafficLight` using an enum + exhaustive `switch`.

Mini Project

State-to-view mapper: Define a sealed `ScreenState` (`Loading`, `Empty`, `Loaded(items)`, `Failure(message)`) and a pure function mapping it to a text description via an exhaustive `switch`. Add a new state and observe the compile error until handled. Acceptance: no `_` used; adding a variant breaks compilation; tests cover each state.

Functions (Parameters, Arrow, Closures, Higher-Order, `typedef`)

> A function is a first-class value in Dart: it can be stored, passed, and returned — which is what makes callbacks, `map` / `where`, and Flutter's builder pattern possible.

Introduction

Functions in Dart are **first-class objects**. This file covers the parameter system (positional, optional, named, required), arrow syntax, closures, higher-order functions, and `typedef` — the machinery behind every Flutter callback (`onPressed`, `builder`, `onChanged`).

Why this concept exists

UI is event-driven: you must hand the framework code to run *later* ("when tapped, do this"). That requires treating functions as values you can pass around. First-class functions and closures make declarative, reactive UIs expressible without boilerplate observer classes.

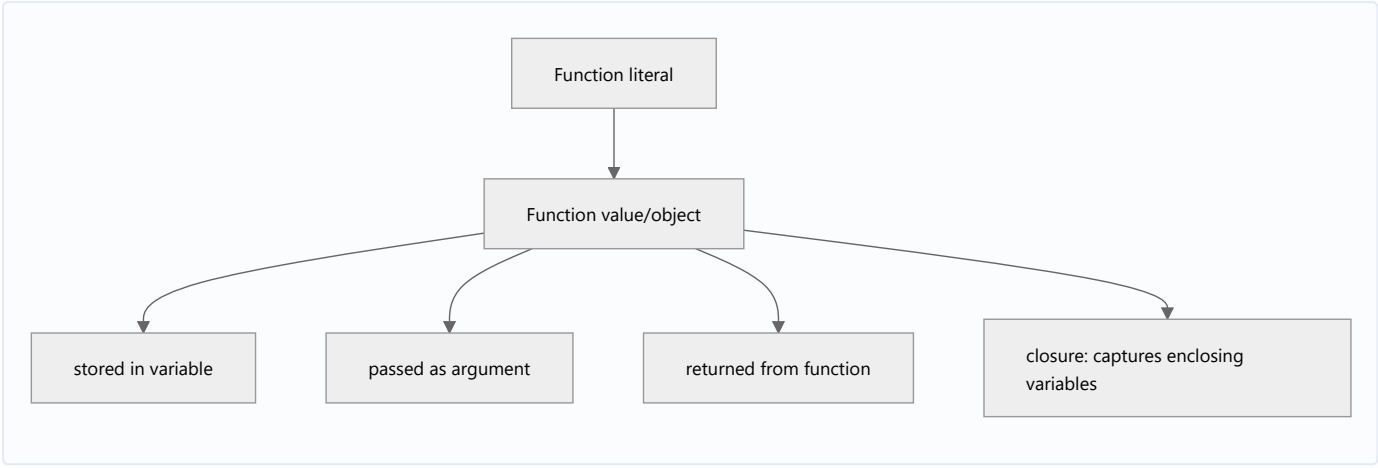
Real-world analogy

A function is a **recipe card**. A first-class function means you can hand the card to someone else (pass it), file it in a box (store it), or have a recipe that produces new recipes (return one). A closure is a recipe card with a **sticky note** attached remembering an ingredient from where it was written.

Problem Statement

You want a button that increments a counter, a list that maps `Person → name`, and a factory that builds `+N` adder functions. All three need functions-as-values. By the end you'll write each idiomatically.

Internal Working



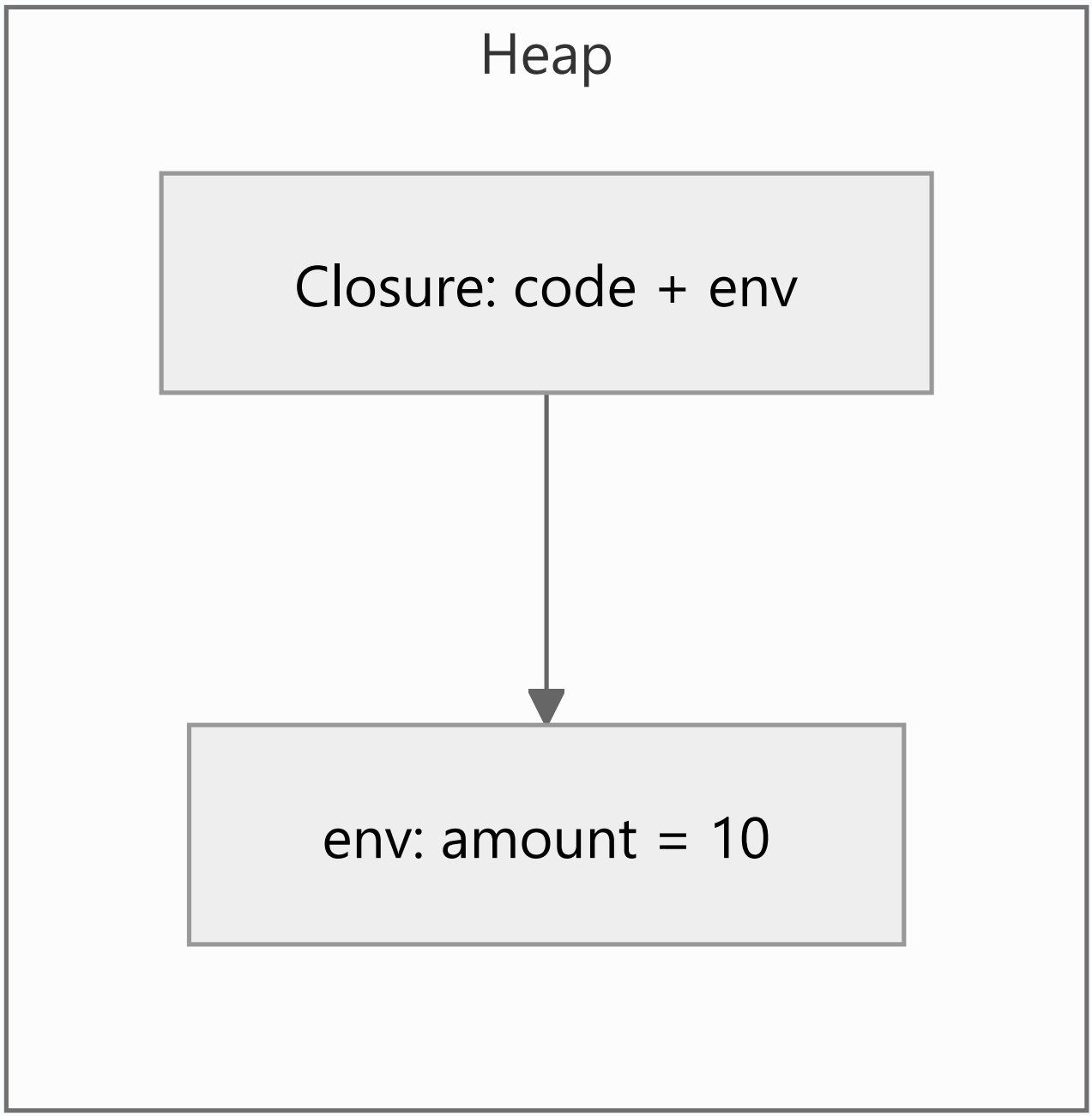
Parameter kinds:

Kind	Syntax	Notes
Positional required	`f(a, b)`	order matters
Optional positional	`f(a, [b, c])`	must be nullable or have default
Named	`f({a, b})`	optional by default, order-free
Named required	`f({required a})`	caller must pass

- **Arrow** `=> expr` is sugar for `{ return expr; }` for a single expression.
- **Closure**: an inner function that captures and retains variables from its lexical scope, keeping them alive after the outer scope returns.
- **Higher-order function**: takes and/or returns a function.
- `typedef` : a named alias for a function signature.

Memory Representation

- A closure is a heap object holding a code pointer **plus** references to the captured variables (its *environment*). Captured variables are promoted from the stack to the heap so they survive after the enclosing function returns.
- Each call to a closure factory creates a **new, independent** environment.



Compiler Behavior

- The compiler infers function types; `(p) => p.city` has inferred type `String Function(Person)` from context.
- Passing `f()` (call) where `f` (reference) is expected is a common type error the compiler catches.
- Tear-offs: `list.forEach(print)` passes `print` as a value (a method tear-off).

Runtime Behavior

- Calling a closure reads its captured environment; a counter closure mutates the *same* captured variable each call.
- Missing required named arg → compile error; a wrong-type arg to a `dynamic`-typed call → runtime error.

Flutter Engine Behavior

Not applicable directly — but the framework stores your callback/builder closures and invokes them during event dispatch and the build phase.

Dart VM Behavior

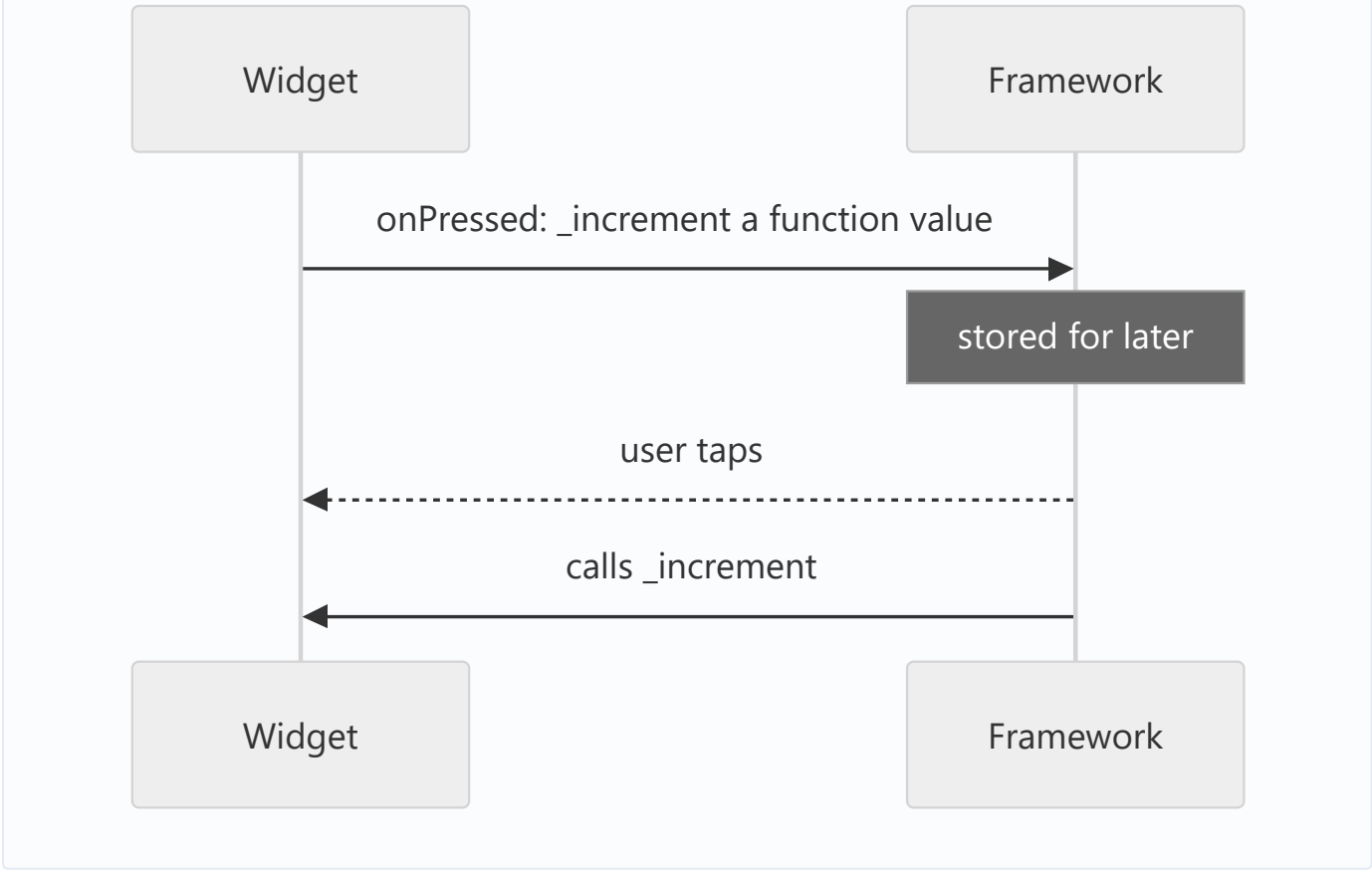
- Closures are ordinary allocations subject to GC; heavily-created short-lived closures (e.g., rebuilt every frame) add GC pressure — a real Flutter performance consideration.

Examples

```
typedef IntTransform = int Function(int value); // named signature


IntTransform makeAdder(int amount) {
  return (value) => value + amount; // closure capturing `amount`
}


int Function() makeCounter() {
  var count = 0;                 // captured, mutable
  return () => ++count;          // remembers & mutates the same count
}


String describe(String name, {int age = 0, required String city}) =>
    '$name ($age) from $city';


void main() {
  final addTen = makeAdder(10);
  print(addTen(5)); // 15


  final counter = makeCounter();
  print(counter()); // 1
  print(counter()); // 2 — same captured count
```

```
  // higher-order: map takes a function
  final people = [('Alice', 'Mumbai'), ('Bob', 'Delhi')];
  final names = people.map((p) => p.$1).toList();
  print(names); // [Alice, Bob]

  // named + required + default
  print(describe('Ada', city: 'London'));        // Ada (0) from London
  print(describe('Ada', age: 36, city: 'London')); // Ada (36) from London

  // reference vs call:
  void sayHi() => print('hi');
  final ref = sayHi; // store the function
  ref();             // hi — now it runs
}
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
`onPressed: myFn()`	Runs during build, not on tap	`onPressed: myFn` (reference)
Mixing `[optional]` and `{named}`	Illegal in one signature	Choose one style
Capturing a loop variable unexpectedly	Closures capture the variable, not its value	Use `for (final x in ...)` (fresh binding)
Rebuilding heavy closures every frame	GC pressure/jank	Hoist stable callbacks; use `const` /fields

Best Practices

- Prefer **named parameters** for 2+ optional args (self-documenting — why Flutter widgets use them).
- Use arrow syntax only for genuinely single expressions.
- Pass function **references**, not calls, to callbacks.
- Name function-type parameters clearly (`onTap` , `keyOf` , `builder`).

Performance

- Method tear-offs and hoisted callbacks avoid per-frame closure allocation.
- Closures are cheap individually but costly in hot paths (per-item in huge lists, per-frame in animations).

Advantages

- Enables declarative, reactive, callback-driven APIs.
- Closures give encapsulated, private mutable state without a class.

Disadvantages

- Over-capturing can retain memory longer than expected (a captured object can't be GC'd while the closure lives).
- Closures in hot paths add allocation/GC cost.

Interview Questions

1. 🟢 **What is a first-class function?** — Functions can be stored in variables, passed as arguments, and returned from other functions.

2. 🟢 **Difference between `=>` and `{}` bodies?** — `=>` is a single expression with an implicit return; `{}` is a block requiring an explicit `return`.
3. 🟡 **What is a closure? Give an example.** — A function that captures and retains variables from its lexical scope. `int Function() c() { var n=0; return ()=>++n; }` keeps `n` alive across calls.
4. 🟡 **Named vs optional-positional parameters — when each?** — Named for clarity/many optionals (order-free, `required`-able); optional positional for a few natural, ordered args.
5. 🟡 **Why is `onPressed: myFn()` a bug?** — It *calls* `myFn` during build and passes its return value; you want to pass the function reference `myFn` to be called on the event.
6. 🔴 **How can closures cause memory leaks in Flutter?** — A closure retains its captured variables (e.g., `this` / `BuildContext`); if stored long-lived (a stream subscription, a static), it keeps those objects alive. Cancel/dispose to release.
7. 🔴 **What is a higher-order function? Name three in the SDK.** — Takes/returns a function; e.g., `Iterable.map`, `where`, `fold`, `List.sort` (compare).

Senior Engineer Tips

- Treat callbacks as part of your API contract — document *when* they fire and on which thread.
- In lists/animations, hoist callbacks out of `build` to avoid per-frame allocations.
- `typedef` your recurring callback signatures for readability and refactorability.

Architect Perspective

Function-as-value is the backbone of dependency inversion in Dart: instead of injecting whole strategy classes, you can inject a function. This underpins clean separation between UI and logic (pass a `VoidCallback` up, a `ValueChanged<T>` down) and the builder/strategy patterns in Module 05.

Summary

- Functions are first-class values; closures capture their environment.
- Parameters: positional required, optional positional `[]`, named `{}`, `required`.
- Pass references to callbacks; prefer named params; mind closure allocation in hot paths.

Revision Notes

- First-class: store/pass/return functions. Closure: remembers outer-scope vars.
- `=>` single expr; `{}` needs `return`.
- Named `{}` optional by default; add `required`. Can't mix `[]` and `{}`.
- Callback = pass `fn` not `fn()`.
- Closures in hot paths → GC pressure; hoist them.

Practice Questions

1. Explain why each `makeCounter()` call has an independent count.
2. When would you `typedef` a function signature instead of inlining it?
3. Why do Flutter widgets favor named parameters?

Coding Questions

1. Implement `List<R> mapList<T, R>(List<T> xs, R Function(T) f)` from scratch.
2. Write `Function debounce(void Function() fn, Duration d)` using a closure + `Timer`.
3. Build `makeMultiplier(int factor)` and demonstrate two independent multipliers.

Mini Project

Mini functional toolkit: Implement `pipe`, `compose`, `memoize`, and `once` as pure-Dart higher-order functions with tests. `memoize` caches results in a captured `Map`; `once` runs a function at most one time. Acceptance: all four are generic, closure-based, and covered by simple assertions.

Collections (`List`, `Set`, `Map`, `Iterable`, `Queue`) & Lazy Ops

> `Iterable` is the lazy sequence you compute on demand; `List` / `Set` / `Map` are the concrete, materialized structures you store data in.

Introduction

Collections are how you group data. Dart's core collections are `List` (ordered, indexed, duplicates), `Set` (unique, fast membership), and `Map` (key→value). They all implement or relate to `Iterable`, whose methods (`map`, `where`, `fold`, `reduce`, `expand`) are the daily tools of every Dart developer — and several are **lazy**, which surprises people.

Why this concept exists

Different access patterns need different structures: ordered access (`List`), deduplication/membership (`Set`), keyed lookup (`Map`), FIFO/deque (`Queue`). Laziness in `Iterable` avoids building intermediate collections you never fully consume — important for performance in pipelines and large data.

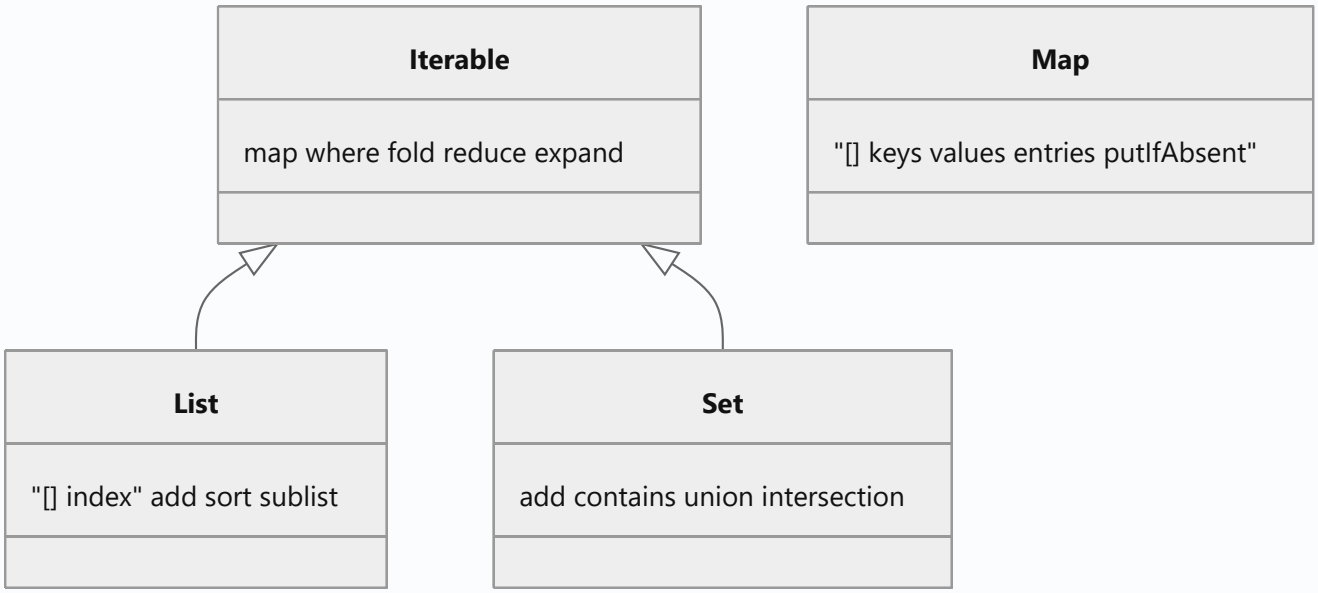
Real-world analogy

- `List` = a **numbered shelf** (position matters, duplicates allowed).
- `Set` = a **guest list** (each name once; "is X invited?" is instant).
- `Map` = a **dictionary** (look up a definition by word).
- Lazy `Iterable` = a **recipe you haven't cooked yet** — nothing happens until you actually plate it (`toList()`/iterate).

Problem Statement

You have a list of orders. You need: unique customer IDs, orders grouped by status, the total revenue, and the first order over ₹1000. You'll pick `Set` for uniqueness, a `Map<K,List<T>>` for grouping, `fold` for the total, and `firstWhere` for the search — and understand which of these are lazy.

Internal Working



Structure	Order	Duplicates	Lookup	Backing (default)
`List`	insertion/index	yes	O(1) by index	growable array
`Set`	insertion (LinkedHashSet)	no	O(1) contains	hash set
`Map`	insertion (LinkedHashMap)	unique keys	O(1) by key	hash map
`Queue`	FIFO/deque	yes	O(1) ends	doubly linked / list

Lazy vs eager `Iterable` methods:

Lazy (return an `Iterable`, run on iteration)	Eager (run immediately)
`map`, `where`, `expand`, `take`, `skip`, `followedBy`	`toList`, `toSet`, `forEach`, `fold`, `reduce`, `any`, `every`, `join`, `length`

Memory Representation

- `List` is a contiguous growable array (amortized O(1) append, occasional realloc + copy on growth).
- `Set` / `Map` are hash tables; iteration order preserved via a linked structure (`LinkedHashSet` / `LinkedHashMap`).
- Lazy `Iterable`s store the source + transformation, not the results — minimal memory until materialized.

Compiler Behavior

- Literal inference: `[1,2]` → `List<int>`, `{1,2}` → `Set<int>`, `{'a':1}` → `Map<String,int>`. **`{}` alone is a `Map`, not a `Set`** — annotate `<int>{}` for an empty set.
- Type args flow through: `map((x)=>x.toString())` yields `Iterable<String>`.

Runtime Behavior

- `reduce` on an empty iterable **throws**; `fold` returns the seed.
- Modifying a collection while iterating throws `ConcurrentModificationError`.
- Lazy chains re-execute on each iteration — iterating a lazy `where().map()` twice runs the transforms twice.

Flutter Engine Behavior

Not applicable. (But `ListView.builder` relies on lazy, index-driven construction — the same "compute on demand" philosophy; see Module 07 Widgets.)

Dart VM Behavior

- Growable list growth is amortized; large known sizes benefit from `List.filled` / `List.generate` to avoid repeated reallocs.
- Hash collections rely on correct `hashCode` / `==` — custom keys with bad hashing degrade to O(n).

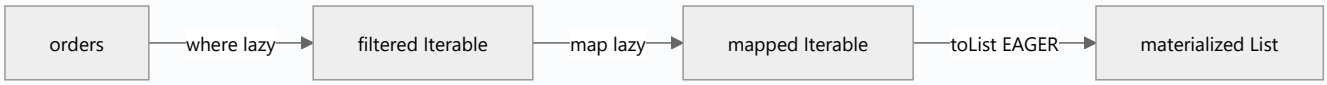
Examples

```
class Order {
  final String customer;
  final String status; // 'paid' | 'pending'
  final int amount;
  const Order(this.customer, this.status, this.amount);
}

Map<K, List<T>> groupBy<T, K>(Iterable<T> items, K Function(T) keyOf) {
  final result = <K, List<T>>{};
  for (final item in items) {
    result.putIfAbsent(keyOf(item), () => <T>[]).add(item);
  }
  return result;
```

```
}

void main() {
  final orders = [
    const Order('A', 'paid', 1200),
    const Order('B', 'pending', 300),
    const Order('A', 'paid', 800),
  ];

  // unique customers -> Set
  final customers = orders.map((o) => o.customer).toSet();
  print(customers); // {A, B}

  // group by status -> Map<K, List<T>>
  final byStatus = groupBy(orders, (o) => o.status);
  print(byStatus.keys); // (paid, pending)

  // total revenue -> fold (eager, empty-safe)
  final total = orders.fold<int>(0, (sum, o) => sum + o.amount);
  print(total); // 2300

  // first over 1000 -> firstWhere with orElse
  final big = orders.firstWhere((o) => o.amount > 1000,
      orElse: () => const Order('none', 'n/a', 0));
  print(big.amount); // 1200

  // LAZY gotcha:
  var evaluations = 0;
  final lazy = orders.where((o) {
    evaluations++;
    return o.status == 'paid';
  });
  print('evaluations so far: $evaluations'); // 0 — nothing ran yet!
  final paid = lazy.toList();                 // NOW it runs
  print('after toList: $evaluations, paid: ${paid.length}'); // 3, 2

  // empty set vs empty map:
  final emptySet = <int>{}; // {} alone would be a Map
```

```
  print(emptySet.runtimeType);
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`{}` for an empty Set	It's a `Map`	Use `<T>{}`
`reduce` on possibly-empty list	Throws	Use `fold(seed, ...)`
Assuming `map` / `where` ran	They're lazy	Materialize with `toList()` / `toSet()`
Mutating during iteration	`ConcurrentModificationError`	Iterate a copy or build a new collection
Custom map keys without `hashCode` / `==`	Lookups fail / O(n)	Implement both (or use records/ `equatable`)

Best Practices

- Pick the structure by access pattern: `List` ordered, `Set` unique/contains, `Map` lookups, `Queue` FIFO/deque.
- Prefer immutable/ `const` collections where possible; `List.unmodifiable` for defensive exposure.
- Materialize lazy chains once if you'll iterate multiple times.
- Use `fold` (empty-safe, any result type) over `reduce` unless the first element is a natural seed.

Performance

- Lazy pipelines avoid intermediate lists — great for large data, but re-run on each iteration.
- `List.generate` / `filled` for known sizes; `StringBuffer` for string joins in hot loops.
- Hash-collection performance hinges on good `hashCode` .

Advantages / Disadvantages

- + Rich, uniform `Iterable` API; laziness saves memory/time; hash structures are O(1).
- − Laziness surprises (side effects, repeated evaluation); wrong structure choice costs performance.

Interview Questions

1. 🟢 **`List` vs `Set` vs `Map`?** — Ordered+indexed+dups; unique+fast-contains; key→value+unique-keys.
2. 🟢 **How to dedupe a list?** — `list.toSet().toList()` (order preserved via LinkedHashSet).
3. 🟡 **Are `map` / `where` eager or lazy?** — Lazy; they return an `Iterable` computed on iteration. Materialize with `toList()` / `toSet()`.
4. 🟡 **`fold` vs `reduce`?** — `reduce` uses the first element as seed and throws on empty (result type = element type); `fold` takes an explicit seed (empty-safe, result type can differ).
5. 🟡 **Why is `{}` a Map, not a Set?** — Historical/literal-ambiguity resolution; empty braces default to `Map`. Use `<T>{}` for a Set.
6. 🔴 **What breaks hash-based lookups for custom keys?** — Not overriding `hashCode` / `==`; equal-by-value keys hash differently, so lookups miss.
7. 🔴 **What is `expand`?** — flatMap: maps each element to an iterable, then flattens — 1→many.

Senior Engineer Tips

- Beware side effects inside lazy `map` / `where`; they run at iteration time, possibly more than once.
- For grouping, `putIfAbsent(key, () => []).add(x)` (or the `collection` package's `groupBy`) is the idiom.
- Expose collections as unmodifiable views from APIs to prevent external mutation.

Architect Perspective

Choice of collection and immutability policy ripples into state management correctness (diffing relies on equality), memory (large lists vs lazy streams), and API safety (unmodifiable views). Establish conventions early: immutable domain collections, lazy pipelines for large transforms, and value-equality on keys.

Summary

- `Iterable` is lazy; `List` / `Set` / `Map` / `Queue` are concrete with distinct tradeoffs.
- `map` / `where` / `expand` / `take` / `skip` are lazy; `toList` / `fold` / `forEach` are eager.
- `{}` = Map; use `<T>{}` for a Set; override `hashCode` / `==` for custom keys.

Revision Notes

- `List`=ordered/dups, `Set`=unique/contains, `Map`=lookup, `Queue`=FIFO/deque.
- Lazy: map/where/expand/take/skip. Eager: toList/toSet/fold/reduce/forEach/join.
- `reduce` throws on empty; `fold` empty-safe + any result type.

- `{}` →Map; `<T>{}` →Set. Custom keys → `hashCode` + `==`.
- Dedupe: `toSet().toList()`.

Practice Questions

1. Predict the printed `evaluations` count in the lazy example and explain.
2. When does `firstWhere` throw, and how do you make it safe?
3. Why does a `Set` of a class with only `==` overridden still allow "duplicates"?

Coding Questions

1. Implement `groupBy<T,K>` and group a list of words by first letter.
2. Write `Map<T,int> frequency<T>(Iterable<T> xs)` counting occurrences.
3. Implement a bounded LRU-ish cache using `LinkedHashMap` (evict oldest at capacity).

Mini Project

Order analytics (pure Dart): From a list of `Order`s, compute unique customers, revenue per status (grouped map), top customer by spend, and a lazily-filtered "large orders" pipeline you materialize once. Print a small report. Acceptance: correct use of `Set` / `Map` / `fold`; a comment proving you understand where laziness kicks in; `dart analyze` clean.

Null Safety (`?`, `!`, `??`, `??=`, `late`, flow analysis)

> Sound null safety means a non-nullable type is *guaranteed* non-null at runtime — the compiler proves it, so a whole class of crashes becomes impossible.

Introduction

Dart has **sound null safety**: types are non-nullable by default, and you opt into nullability with `?`. This file covers the operators (`?.`, `??`, `??=`, `!`), the `late` keyword, and **flow analysis / type promotion** — the compiler feature that makes null safety ergonomic.

Why this concept exists

The "billion-dollar mistake" — null dereferences — was the single largest source of mobile crashes. Null safety moves those errors from **runtime** (crash in the user's hands) to **compile time** (red squiggle in your editor). "Sound" means the guarantee can be *trusted*: there are no hidden nulls sneaking through typed code.

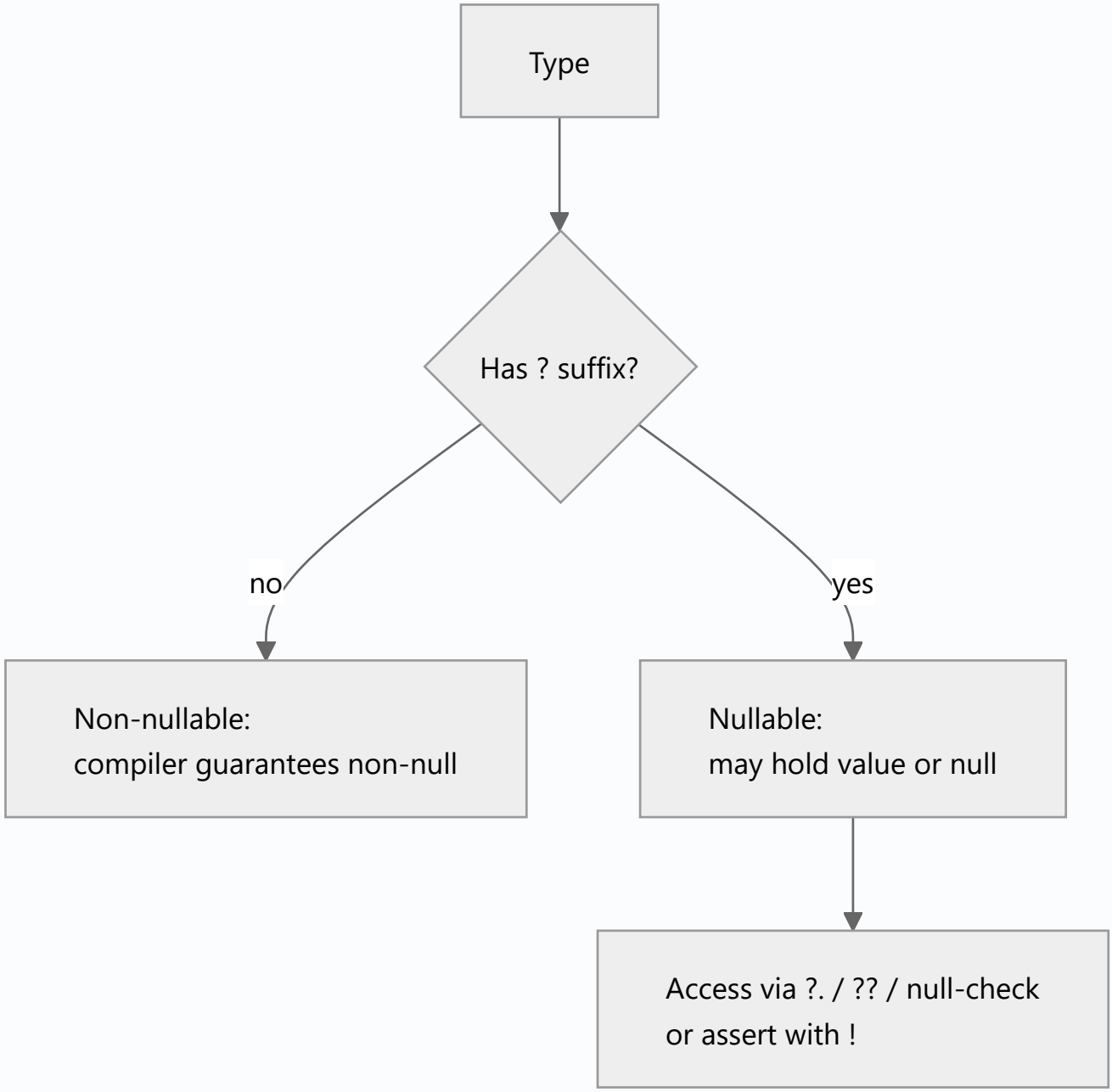
Real-world analogy

A non-nullable `String` is a **sealed box guaranteed to contain a letter**. `String?` is a box that **might be empty**. `?.` = "open only if there's a letter." `??` = "if empty, use this spare letter." `!` = "I swear this box has a letter" — and if you're wrong, it explodes immediately (fail-fast).

Problem Statement

A user profile has an optional `middleName` and a required `id`. You must display the middle name or a fallback, never crash, and assert the `id` is present after login. By the end you'll pick `?.` / `??` for the optional field and `!` (or better) for the guaranteed one.

Internal Working



Operators:

Operator	Meaning
`T?`	nullable type
`?.`	null-aware access — whole expression is `null` if receiver is `null`
`??`	if-null — `a ?? b` returns `b` when `a` is `null`
`??=`	null-aware assign — assign only if currently `null`
`!`	null assertion — asserts non-null; **throws** if actually `null`
`...?`	null-aware spread; `?..` null-aware cascade

Flow analysis / promotion: after `if (x != null)`, the compiler *promotes* `x` to non-nullable inside that block, so you can use it without `!`.

Memory Representation

- `null` is a single canonical `Null` instance shared program-wide.
- Nullability is a **static** property (compile-time type), not a runtime tag — there's no per-value "nullable flag"; the type system tracks it.

Compiler Behavior

- A non-nullable variable **must** be definitely assigned before use.
- Promotion works on **local** variables and `final` fields, but **not** on non-final instance fields (another method could mutate them between check and use).
- Using `!` compiles to a runtime null-check that throws on failure.

Runtime Behavior

- `x!` when `x` is `null` → throws `_TypeError`: "*Null check operator used on a null value.*"
- `late` variable read before assignment → `LateInitializationError`.
- `?.`, `??`, `??=` never throw due to null; they gracefully short-circuit.

Flutter Engine Behavior

Not applicable — pure language feature. (But null safety eliminates a large share of Flutter runtime crashes, improving perceived stability.)

Dart VM Behavior

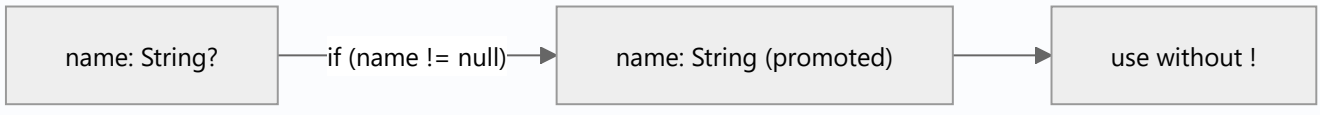
- Sound null safety lets the AOT compiler **omit** many null checks it would otherwise insert, and enables tighter code generation — a modest performance win alongside the correctness win.

Examples

```
class User {
  final String id;
  final String? middleName; // optional
  User(this.id, {this.middleName});
}
```

```
String display(User u) {
  // optional -> graceful fallback
  final mid = u.middleName ?? '(none)';
  return '${u.id}: $mid';
}

void main() {
  final a = User('u1', middleName: 'Quorra');
  final b = User('u2');
  print(display(a)); // u1: Quorra
  print(display(b)); // u2: (none)

  String? maybe;
  print(maybe?.length);      // null — null-aware, no crash
  print(maybe ?? 'default'); // default
  maybe ??= 'assigned';      // assigns because it was null
  print(maybe);              // assigned

  // flow promotion:
  String? name = _fetch();
  if (name != null) {
    print(name.length); // OK — promoted to non-null String here
  }

  // ! asserts an invariant (fail-fast if wrong):
  final id = _loggedInUserId()!; // we KNOW we're logged in
  print(id.length);
}

String? _fetch() => 'Ada';
String? _loggedInUserId() => 'u42';
```

Diagrams



Common Mistakes

Mistake	Why it's wrong	Fix
Sprinkling `!` to silence the analyzer	Reintroduces crashes	Use `?.` / `??` /null-check; reserve `!` for true invariants
Expecting promotion on instance fields	Non-final fields don't promote	Copy to a local: `final v = obj.field; if (v != null) ...`
`late` to dodge nullability when null is valid	Turns a valid state into a crash	Use `T?` when absence is legitimate
`a ?? b` vs `a ??= b` confusion	Different semantics	`??` returns; `??=` assigns-if-null

Best Practices

- Ask: **"Can null happen in normal use?"** → yes: `?.` / `??`. **"Would null mean a bug?"** → yes: `!` (fail-fast).
- Prefer nullable types over `late` unless you have a real deferred-init pattern.
- Copy nullable instance fields to locals to enable promotion.
- Provide sensible defaults with `??` at the UI boundary.

Performance

- Null safety enables the compiler to drop redundant null checks (minor speedup).
- `!` inserts a check; negligible individually, but don't scatter it needlessly.

Advantages

- Eliminates null-dereference crashes from typed code.
- Self-documents intent (which values can be absent).
- Enables compiler optimizations.

Disadvantages

- `late` and `!` can move errors back to runtime if misused.
- Migration friction for legacy code (historical).

Interview Questions

1. 🟢 **What does "sound" null safety mean?** — The non-null guarantee holds at runtime; the compiler statically proves non-nullable values are never null, so no hidden null crashes from typed code.

2. 🟢 **Difference between `?` and `!`?** — `?` makes a type nullable (declaration); `!` asserts non-null at a use site (may throw).
3. 🟡 **When does `!` throw?** — When applied to a value that is actually `null`, throwing "Null check operator used on a null value."
4. 🟡 **`??` vs `??=`?** — `a ?? b` is an expression returning `a` or `b`; `a ??= b` assigns `b` to `a` only if `a` is `null`.
5. 🟡 **Why might promotion fail on a class field?** — Non-final fields can be mutated between the check and use (even concurrently), so the compiler can't guarantee it stays non-null. Use a local or `final`.
6. 🔴 **Why does `!` even exist if `?.` is safe?** — `!` asserts an *invariant the compiler can't prove* (e.g., "we're logged in"). If violated, you want a loud, immediate crash (fail-fast) rather than silently corrupt state.
7. 🔴 **The two legitimate uses of `late`?** — Deferred init of a non-nullable (set before first read, e.g., in `initState`); and lazy init of a `final` (computed once on first access).

Senior Engineer Tips

- `!` is a *claim about your system*, not a null handler. Put it only where a layer above genuinely guarantees non-null (post-login, required ctor arg, validated input).
- Treat every `late` as a small debt: it's a runtime check you accepted in exchange for non-null typing.
- In reactive code, model "loading/absent/present" with a sealed type rather than nullable-plus-bool flags.

Architect Perspective

Null safety is a system-wide invariant strategy. Decide *where* nullability is allowed (edges/DTOs) and *where* it's forbidden (domain entities). Domain models should be non-nullable and validated at construction, so the rest of the system never null-checks business data. This dovetails with DDD value objects (Module 46) and typed failures (Module 38).

Summary

- Non-nullable by default; opt in with `?`. Sound = trustworthy at runtime.
- `?.` / `??` / `??=` handle null gracefully; `!` asserts non-null and fails fast.
- Flow analysis promotes locals/finals after a null check; instance fields don't promote.

Revision Notes

- `?` declares nullable; `!` asserts non-null (may throw).

- `??` fallback (expression); `??=` assign-if-null.
- Promotion: locals/finals yes, non-final fields no → copy to local.
- Null valid → `?.` / `??`; null = bug → `!` (fail fast).
- `late`: deferred non-null init OR lazy final; read-before-set → `LateInitializationError`.

Practice Questions

1. Rewrite `if (u.middleName != null) print(u.middleName!.length);` without `!` (hint: local).
2. Give one case each where `!` is correct and where it's an abuse.
3. Why can't the compiler promote a `var` instance field?

Coding Questions

1. Implement `T orElse<T>(T? value, T fallback) => value ?? fallback;` and test with null/non-null.
2. Write a `Cache` with `late final Map<String,int> _store` initialized lazily; prove it initializes on first access only.
3. Model a `RemoteData` sealed type (`Loading` / `Error` / `Data<T>`) that replaces `T? data + bool loading + String? error`.

Mini Project

Null-safe settings resolver: Build `Settings` where each getter resolves user override → remote default → hardcoded default using `??` chains, exposes only non-nullable resolved values, and uses `!` nowhere. Add one `late final` expensive computed value. Acceptance: no runtime null crash possible; `dart analyze` clean; a test proves fallbacks resolve correctly.

Enums (Basic & Enhanced)

> An enum is a type with a fixed, named set of instances — the safest way to model "one of a known set of options."

Introduction

Enums model a closed set of constants (`Status.loading` , `Status.error`). Dart 2.17+ added **enhanced enums**: enums can now have fields, constructors, methods, and implement interfaces — closing much of the gap with full classes while keeping exhaustiveness.

Why this concept exists

Using raw strings/ints for states (`"loading"` , `0`) is error-prone: typos compile, invalid values slip through, and `switch` can't be exhaustive. Enums give the compiler a **closed set** to check, enabling exhaustive `switch` and eliminating "magic value" bugs.

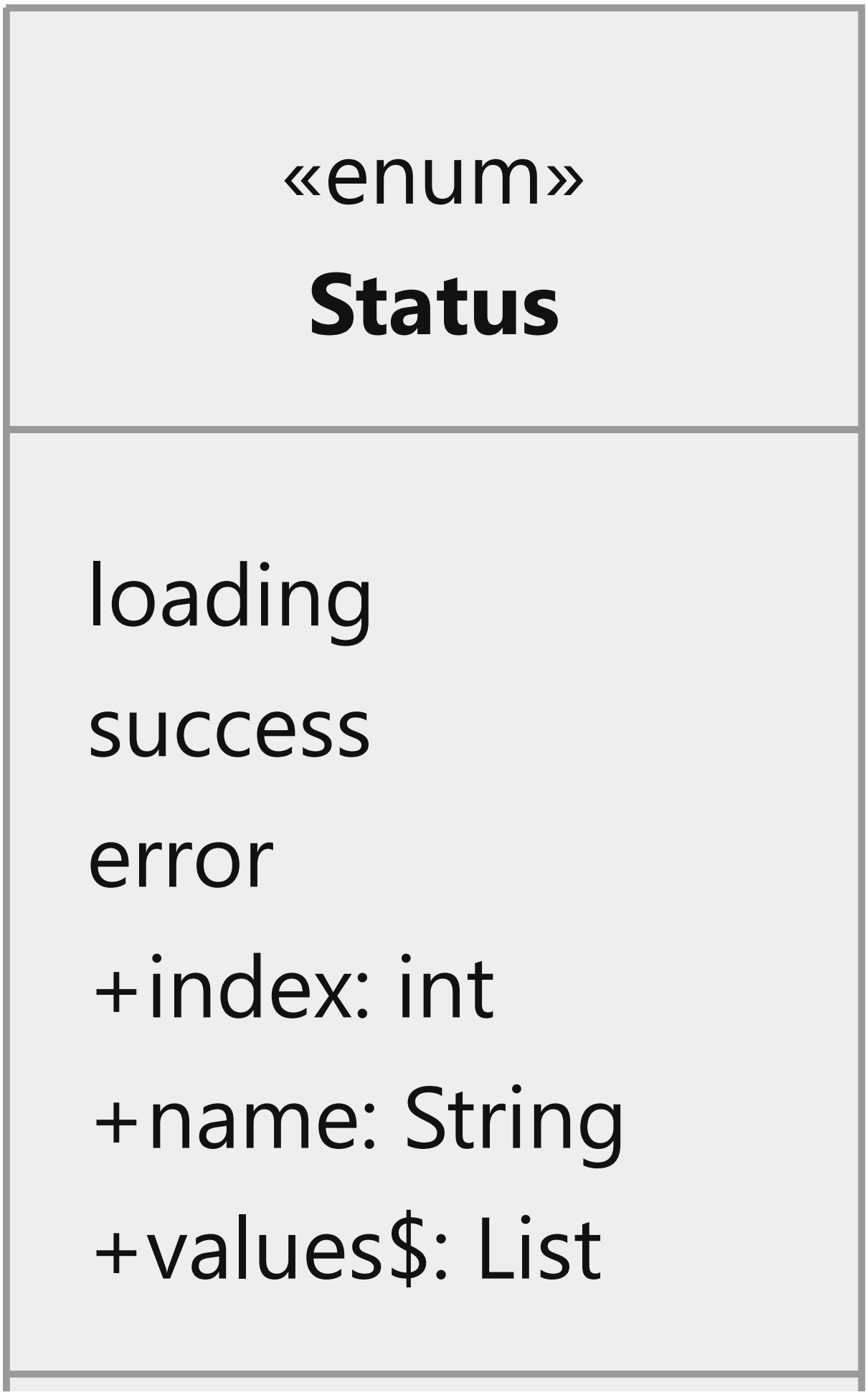
Real-world analogy

An enum is a **gear stick**: P, R, N, D — a fixed set of positions. You can't shift into "banana." Enhanced enums add a **label and behavior** to each position (e.g., each gear knows its display name and whether the car can move).

Problem Statement

Model an HTTP request state and a set of user roles where each role carries a permission level and a display label, then exhaustively map state → UI. You'll use a basic enum for state and an enhanced enum for roles.

Internal Working


«enum»
Status
loading
success
error
+index: int
+name: String
+values$: List

- Every enum value has `index` (declaration order) and `name` (its identifier as a string).
- `Status.values` is the list of all values (great for iteration/dropdowns).
- **Enhanced enums** declare a `const` constructor and `final` fields; each value calls the constructor with its arguments.
- Enums can implement interfaces and define methods/getters, but cannot be `extended`.

Memory Representation

- Enum values are `const`, canonicalized singletons — one instance per value, shared everywhere. Comparisons are identity-fast.

Compiler Behavior

- `switch` over an enum without covering all values (and no `_`) is a **compile error** — exhaustiveness.
- Enhanced enum constructors must be `const`; fields must be `final`.

Runtime Behavior

- `Status.values`, `.name`, `.index` available at runtime.
- Parse from string via `Status.values.byName('error')` (throws if absent) or `firstWhere` with `orElse`.

Flutter Engine Behavior

Not applicable. (Enums are the idiomatic backing for state in `build` switches and for typed configuration like `MainAxisAlignment`.)

Dart VM Behavior

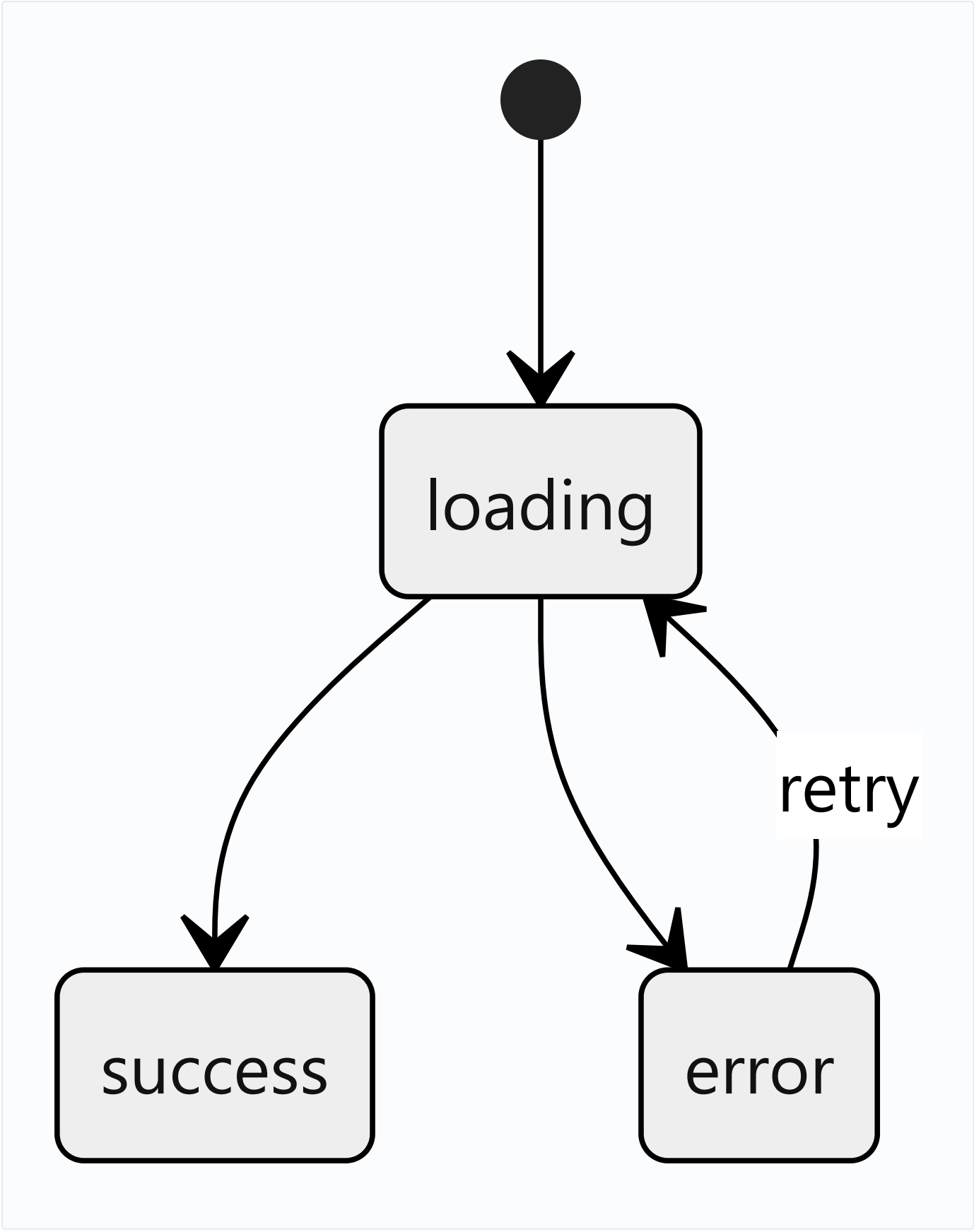
- Enum switches over dense indices may compile to jump tables (fast dispatch).

Examples

```
// Basic enum
enum Status { loading, success, error }

// Enhanced enum: fields + const ctor + method + interface
enum Role implements Comparable<Role> {
  guest(0, 'Guest'),
  member(1, 'Member'),
  admin(2, 'Administrator');

  const Role(this.level, this.label);
  final int level;
  final String label;

  bool get canModerate => level >= member.level;

  @override
  int compareTo(Role other) => level.compareTo(other.level);
}

String view(Status s) => switch (s) {
      Status.loading => 'Spinner',
      Status.success => 'Content',
      Status.error => 'Retry',
    };

void main() {
  print(view(Status.success)); // Content

  print(Status.error.name);    // error
  print(Status.error.index);   // 2
  print(Status.values);        // [Status.loading, Status.success, Status.error]

  final r = Role.admin;
  print('${r.label} lvl=${r.level} mod=${r.canModerate}'); // Administrator lvl=2 mod=true

  // parse from string safely
  final parsed = Role.values.byName('member');
  print(parsed.label); // Member
```

```
  // sort using Comparable
  final roles = [Role.admin, Role.guest, Role.member]..sort();
  print(roles.map((e) => e.label).toList()); // [Guest, Member, Administrator]
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using `String` / `int` constants instead of enums	Typos compile, no exhaustiveness	Use an enum
Adding `_` / `default` to enum `switch`	Kills exhaustiveness protection	Handle each case explicitly
`byName` on untrusted input without catch	Throws `ArgumentError`	`values.firstWhere(..., orElse:)` or try/catch
Persisting `index` to storage	Reordering values corrupts data	Persist `name` (stable string)

Best Practices

- Prefer enums (basic or enhanced) over magic values.
- Persist/serialize by `name`, not `index`.
- Use enhanced enums when each value needs data/behavior; keep them small and cohesive.
- Omit `_` in switches to keep the compiler enforcing new variants.

Performance

- Enum values are canonical singletons — equality/switch is very cheap; safe to use in hot paths.

Advantages / Disadvantages

- **+** Type-safe closed set, exhaustive switching, self-documenting; enhanced enums carry data/behavior.
- **−** Can't be extended; not suitable for open/extensible sets (use sealed classes for open hierarchies with payloads).

Interview Questions

1. 🟢 **What is an enum and why prefer it over string constants?** — A type with a fixed set of named instances; it gives compile-time safety and exhaustive switching that string constants can't.
2. 🟢 **What do `.name`, `.index`, and `.values` give you?** — The value's identifier string, its zero-based declaration position, and the list of all values.
3. 🟡 **What are enhanced enums?** — Enums with `const` constructors, `final` fields, methods/getters, and interface implementation (Dart 2.17+).
4. 🟡 **Why persist `name` instead of `index`?** — `index` shifts if you reorder/insert values, corrupting stored data; `name` is stable.

5. 🔴 **Enum vs sealed class — when each?** — Enum for a fixed set of simple singletons; sealed class when variants carry different payloads/shapes (e.g., `Success(data)` vs `Failure(error)`).
6. 🔴 **Can enums implement interfaces or extend classes?** — Implement interfaces: yes. Extend classes: no.

Senior Engineer Tips

- Reach for enhanced enums to attach display labels, API codes, or capability flags to each value — avoids parallel maps.
- For state with data, prefer **sealed classes** (Dart 3) over enums; use enums for pure options.
- Add a `fromApi` / `byName`-with-fallback factory for robust deserialization.

Architect Perspective

Enums (and sealed types) are the vocabulary of your domain's finite states and options. Modeling them precisely makes state machines, feature flags, and permissions type-safe end to end, and keeps serialization stable across app versions — a real production concern for stored/enum'd data.

Summary

- Enums model a closed set of named singletons with `name` / `index` / `values`.
- Enhanced enums add fields, `const` ctors, methods, and interfaces.
- Prefer them over magic values; serialize by `name`; use sealed classes when variants carry data.

Revision Notes

- Enum = fixed named singletons; canonical + cheap equality.
- `.name` `.index` `.values`; parse via `values.byName` / `firstWhere`.
- Enhanced enum = fields + `const` ctor + methods + `implements`.
- Persist `name` not `index`. Sealed class for payload-carrying variants.

Practice Questions

1. Why does adding a new enum value to an exhaustive switch cause a helpful compile error?
2. When would you migrate an enum to a sealed class?
3. What goes wrong if you stored `index` and later alphabetized the enum?

Coding Questions

1. Model `enum Weekday` with an enhanced `isWeekend` getter and a `today`-style parser.
2. Build `enum HttpStatus` carrying the numeric code + category, with a `fromCode` factory.
3. Implement a role-based `canAccess(Role, Feature)` using an enhanced enum's `level`.

Mini Project

Permissions engine (pure Dart): Define an enhanced `Role` enum (level + label) and a `Feature` enum, plus a function gating features by role level. Add safe parsing from strings and an exhaustive `switch` mapping roles to a home-screen label. Acceptance: no magic strings; serialization by name; `dart analyze` clean; tests cover access rules.

Records & Patterns (Dart 3)

> Records are lightweight anonymous tuples for bundling values; patterns are a syntax for destructuring and matching them — together they make data flow concise and type-safe.

Introduction

Records (Dart 3) are immutable, anonymous aggregates: `(int, String)` or `(id: 1, name: 'A')`. **Patterns** let you *destructure* records, lists, maps, and objects, and *match* them in `switch`, `if-case`, and variable declarations. This is one of the biggest ergonomic leaps in modern Dart.

Why this concept exists

Before records, returning multiple values meant creating a throwaway class or abusing `List` / `Map`. Records solve that with zero boilerplate. Patterns solve the inverse problem: pulling structured data apart safely and exhaustively, replacing towers of `if` / `is` /cast with declarative matching.

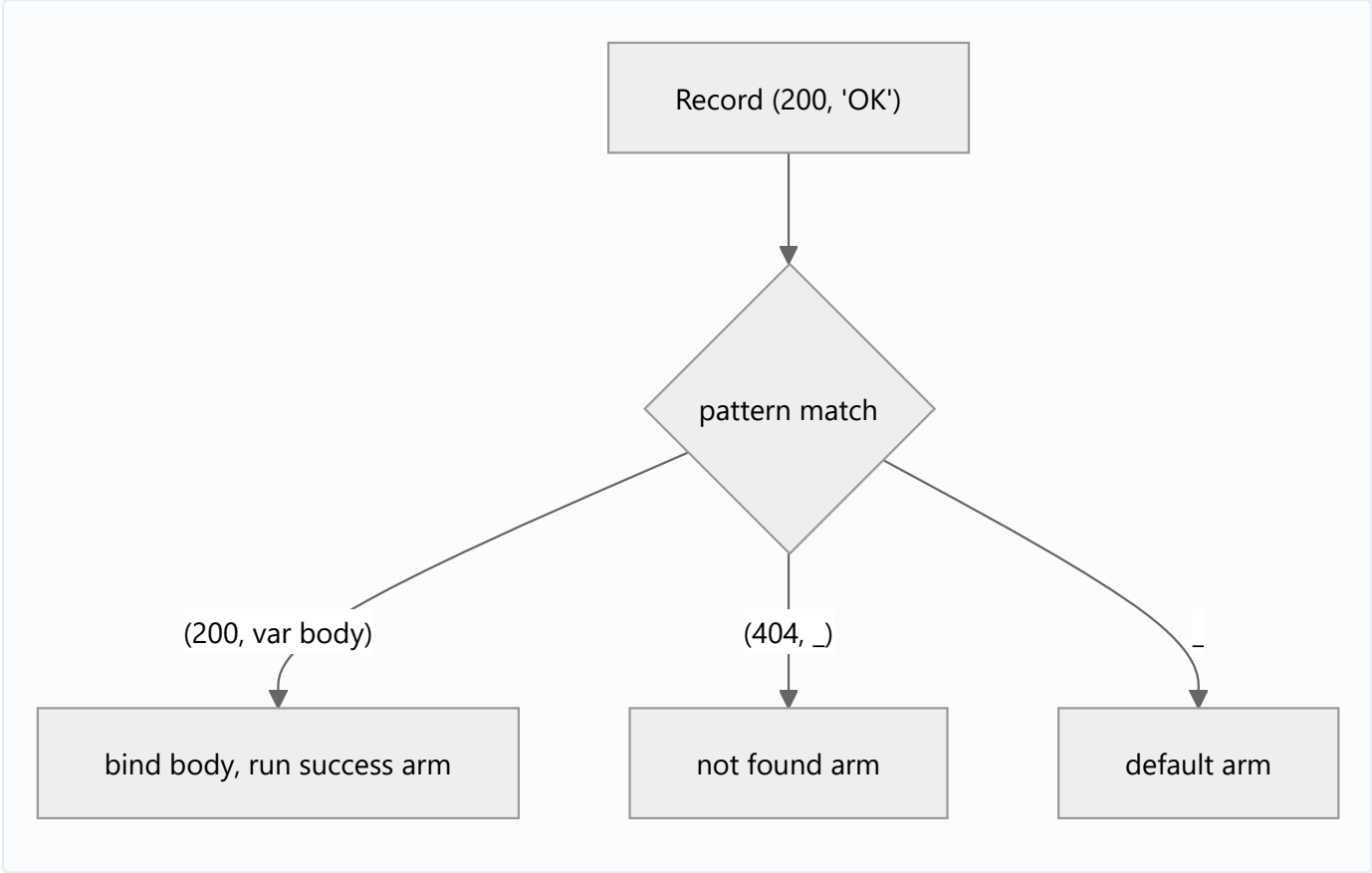
Real-world analogy

A record is a **labeled lunchbox** with compartments (`(main: rice, side: salad)`) — no need to design a "LunchBox class." A pattern is a **cookie cutter**: press it onto dough (your data) and it both *checks the shape* and *cuts out the pieces* you named.

Problem Statement

A function must return both a status code and a body. A caller must handle `(200, body)` vs `(404, _)` vs anything else, and destructure a `{'lat':.., 'lng':..}` map. You'll return a record and match with patterns.

Internal Working



Records:

- Positional: `(1, 'a')` with fields `$1`, `$2`. Named: `(id: 1, name: 'a')` with fields `.id`, `.name`.
- Immutable; **structural equality** (two records with equal fields are `==`) and a sensible `hashCode` — for free.
- The type is the shape: `(int, String)` or `({int id, String name})`.

Patterns appear in:

- Destructuring declarations: `final (a, b) = (1, 2);`
- `switch` expressions/statements (with exhaustiveness).
- `if-case`: `if (json case {'lat': double lat}) ...`
- Guards: `case (var x, var y) when x > y`.

Pattern kinds: record, list `[a, b, ...rest]`, map `{'k': v}`, object `User(:name)`, variable `var x`, constant `200`, wildcard `_`, logical-or `A || B`, cast `x as T`, null-check `x?`.

Memory Representation

- Records are immutable heap objects holding their fields; the compiler generates their equality/hashCode. Small records are cheap; they're value-like in behavior but reference objects in memory.

Compiler Behavior

- Record types are structural: `(int, String)` from two places are the same type.
- `switch` with patterns is checked for **exhaustiveness** (over sealed types/enums/bool).
- Refutable patterns in `if-case` / `switch` compile to efficient tests + binds; irrefutable ones in declarations always match.

Runtime Behavior

- Destructuring binds new variables; `if-case` runs the branch only if the pattern matches.
- Record equality compares fields structurally at runtime.

Flutter Engine Behavior

Not applicable. (Records are handy for returning e.g. `(Widget, VoidCallback)` from helpers, and patterns clean up `build` logic.)

Dart VM Behavior

- Pattern matching lowers to comparisons/type-tests; dense constant patterns can use jump tables like `switch`.

Examples

```
// return multiple values with a record
(int code, String body) fetch(bool ok) =>
    ok ? (200, 'OK') : (404, 'Not Found');

// named record
({double lat, double lng}) origin() => (lat: 0.0, lng: 0.0);

void main() {
  // destructure a record
  final (code, body) = fetch(true);
  print('$code $body'); // 200 OK

  // switch with record patterns + guard
  final result = fetch(false);
  final message = switch (result) {
    (200, final b) => 'Success: $b',
    (404, _) => 'Missing',
```

```
    (final c, _) when c >= 500 => 'Server error $c',
    _ => 'Other',
  };
  print(message); // Missing

  // named record fields
  final o = origin();
  print('${o.lat}, ${o.lng}'); // 0.0, 0.0

  // structural equality — no class, still equal
  print((1, 'a') == (1, 'a')); // true

  // if-case: destructure a map safely
  final json = {'lat': 12.9, 'lng': 77.6};
  if (json case {'lat': final double lat, 'lng': final double lng}) {
    print('coords $lat,$lng'); // coords 12.9,77.6
  }

  // object pattern destructuring
  const user = (name: 'Ada', age: 36);
  final (name: n, age: a) = user;
  print('$n $a'); // Ada 36
}
```

Diagrams

Common Mistakes

Mistake	Why	Fix
Overusing records for domain models	Loses names/behavior/validation	Use a class for real domain concepts
Forgetting records are immutable	Can't reassign fields	Build a new record
Assuming positional and named interchange	Different types	Match the declared shape
Non-exhaustive pattern switch expecting a throw	Compile error (expression) or no-op (statement)	Cover cases or add `_`

Best Practices

- Use records for **transient, local** multi-value returns; graduate to a class when the shape gains meaning, invariants, or methods.
- Prefer **named** record fields when positions are ambiguous.
- Use `if-case` to validate+destructure JSON at boundaries.
- Keep pattern arms readable; extract complex logic into functions.

Performance

- Records avoid allocating bespoke classes and give free equality — efficient for local data flow. Don't churn huge records in tight loops unnecessarily.

Advantages / Disadvantages

- **+** Zero-boilerplate multi-returns, structural equality, powerful destructuring/matching, exhaustiveness.
- **−** No named type/behavior (records), can reduce self-documentation if overused; patterns have a learning curve.

Interview Questions

1. 🟢 **What is a record?** — An immutable, anonymous aggregate of values (positional and/or named) with structural equality and hashCode for free.
2. 🟢 **How do you return two values without a class?** — Return a record: `(int, String) f() => (1, 'a');`.
3. 🟡 **Records vs a class?** — Records for transient, unnamed bundles; classes for named domain types with invariants/behavior.

4. 🟡 **Where can patterns appear?** — Destructuring declarations, `switch` expressions/statements, `if-case`, and `for-in` destructuring.
5. 🟡 **Are records equal by value?** — Yes; structural equality — equal fields ⇒ `==` true.
6. 🔴 **How do patterns improve JSON handling?** — `if (json case {'k': int v})` validates presence+type and binds in one expression, replacing nested `containsKey` / `is` /cast.
7. 🔴 **What is a refutable vs irrefutable pattern?** — Refutable can fail to match (used in `switch` / `if-case`); irrefutable always matches (used in declarations).

Senior Engineer Tips

- Records are perfect for `Iterable` transforms carrying an index/key alongside a value, and for returning `(value, error)` in simple internal APIs.
- Combine sealed classes + patterns for exhaustive, payload-carrying state machines — the modern replacement for enum-plus-nullable-fields.

Architect Perspective

Patterns + sealed classes give Dart algebraic-data-type ergonomics: model domain states precisely and let the compiler enforce total handling. This raises correctness across large teams and pairs naturally with BLoC states, `Result` types (Module 38), and DDD (Module 46).

Summary

- Records: immutable anonymous tuples (positional/named) with free equality — great for multi-returns.
- Patterns: destructure + match across records/lists/maps/objects in `switch` / `if-case` /declarations.
- Prefer classes for domain types; use records for transient data.

Revision Notes

- Record: `(1,'a')` / `(id:1)`; fields `$1` / `.id`; immutable; structural `==`.
- Patterns: destructure + match; `switch` / `if-case` /declaration; guards via `when`.
- `if (json case {'k': int v})` = validate + bind in one step.
- Sealed class + patterns = exhaustive ADT-style state.

Practice Questions

1. Rewrite a "return a Map with 'ok' and 'data'" function to return a named record.
2. Convert a nested `if/is/cast` JSON parse into a single `if-case`.

3. When is a record the wrong choice, and what do you use instead?

Coding Questions

1. Write `(int min, int max) bounds(List<int> xs)` and destructure the result.
2. Parse `{'type':'circle','r':2}` / `{'type':'rect','w':..,'h':..}` via `switch` map patterns into an area.
3. Implement a `divmod(int a, int b)` returning `(quotient, remainder)` as a record.

Mini Project

Tiny JSON shape validator: Given raw maps for `Circle` / `Rectangle` / `Triangle`, use map + constant patterns in a `switch` to validate the shape and compute area, returning a `(bool ok, String message, double? area)` record. Acceptance: no manual `containsKey` chains; invalid shapes handled by a `_` arm; tests cover valid and invalid inputs.

Exception Handling (`throw`, `try` / `catch` / `on` / `finally`, custom exceptions)

> An exception is an object that signals "the normal flow can't continue"; handling it well means failing loudly where it's a bug and gracefully where it's expected.

Introduction

Dart separates `Error` (programming bugs you should *fix*, e.g., `RangeError`, `AssertionError`) from `Exception` (recoverable conditions you should *handle*, e.g., `FormatException`, `IOException`). This file covers `throw`, `try` / `on` / `catch` / `finally`, custom exception types, and how the readable message actually comes from `toString()`.

Why this concept exists

Real programs meet bad input, network failures, and missing files. Exceptions let you *interrupt* a failing computation and transfer control to a handler, instead of threading error codes through every return value. The `Error` vs `Exception` split encodes intent: fix the bug vs handle the situation.

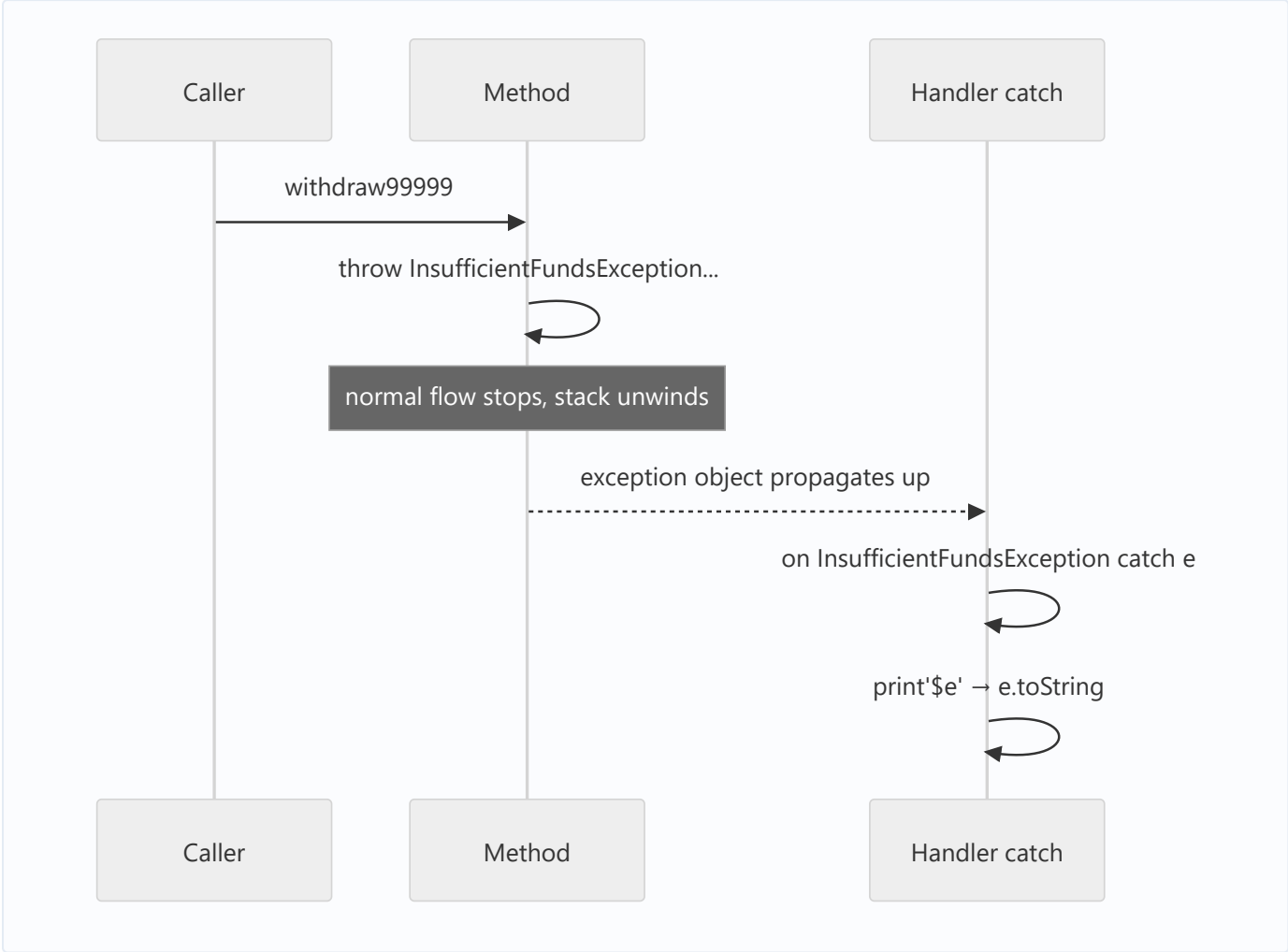
Real-world analogy

`throw` is pulling a **fire alarm**: it stops what you're doing and hands control to whoever is trained to respond (`catch`). `finally` is the **"turn off the stove no matter what"** step that runs whether or not there was a fire. A custom exception is a **specific alarm** ("kitchen fire") rather than a generic one.

Problem Statement

A `withdraw` must reject over-limit amounts with a *meaningful* domain error, callers must catch that specific type and show a friendly message, and a file handle must always close. You'll define a custom exception, catch it with `on`, and clean up in `finally`.

Internal Working



- `throw obj;` sends **any object** up the call stack (idiomatically an `Exception` / `Error`).
- `try { } on T catch (e) { } catch (e, st) { } finally { }`:
  - `on T` catches a specific type; `catch (e)` catches anything; `catch (e, st)` also binds the `StackTrace`.
  - `finally` always runs (success, throw, or return).
- `rethrow` re-throws the current exception preserving its stack trace.
- The human-readable text comes from the exception's `toString()`, invoked automatically by string interpolation.

Memory Representation

- An exception is an ordinary heap object. The captured `StackTrace` references frames at throw time. Uncaught exceptions propagate to the zone's/Isolate's error handler.

Compiler Behavior

- Dart has **no checked exceptions** — methods don't declare what they throw; the compiler won't force `try` / `catch`.
- `on` / `catch` ordering matters: put specific types before general ones (unreachable clauses are flagged).

Runtime Behavior

- Throwing unwinds the stack until a matching handler is found; if none, the isolate reports an uncaught error (in Flutter, `FlutterError.onError` /zone handlers catch these).
- `finally` runs during unwinding; a `throw` / `return` in `finally` overrides the original.

Flutter Engine Behavior

Not applicable at engine level, but Flutter installs global handlers (`FlutterError.onError` , `PlatformDispatcher.onError` , `runZonedGuarded`) to catch and report uncaught errors — see Module 38.

Dart VM Behavior

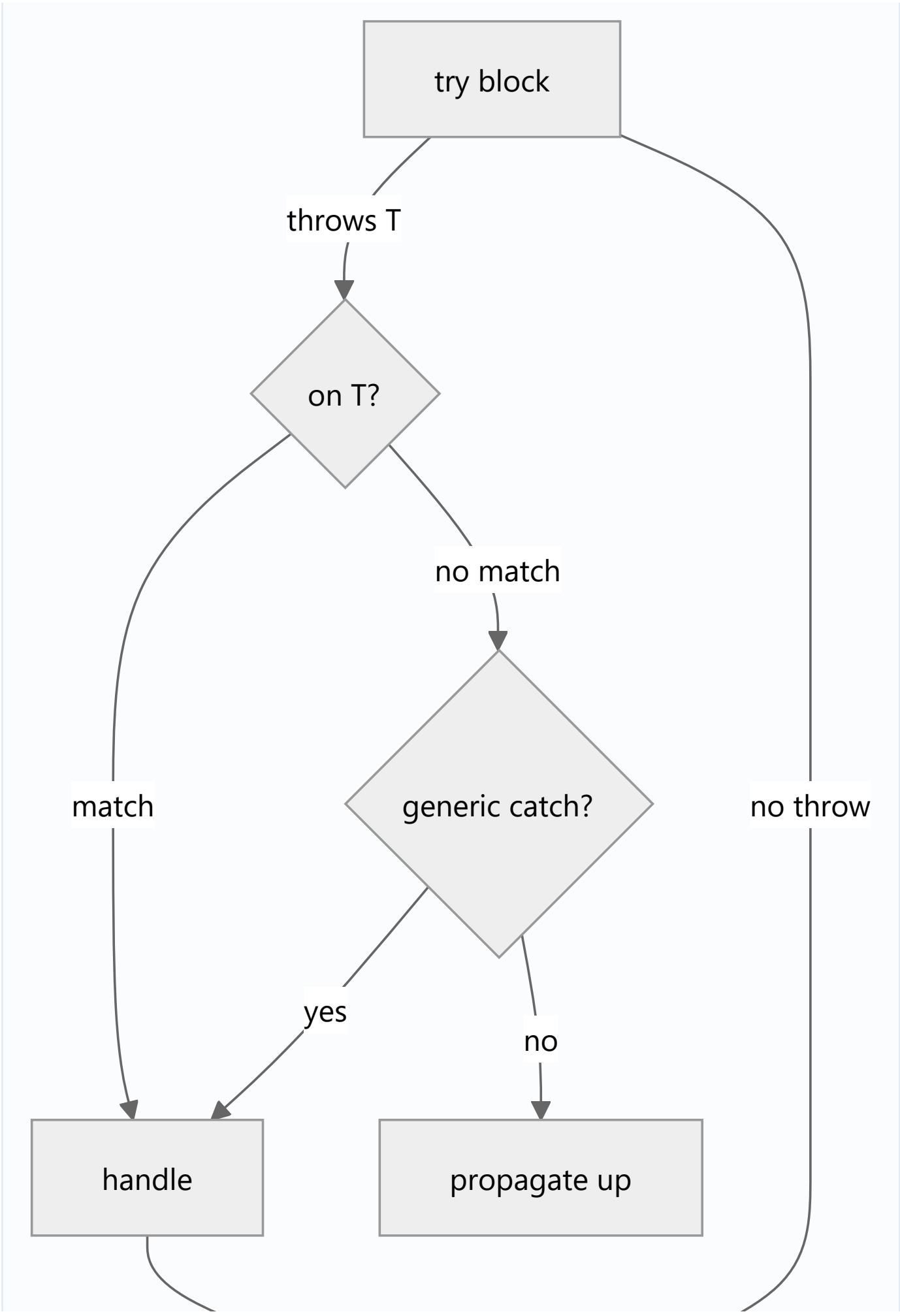
- Zones (`runZonedGuarded`) intercept uncaught async errors. Async errors propagate through `Future` / `await` and are caught by surrounding `try` / `catch` when `await` ed.

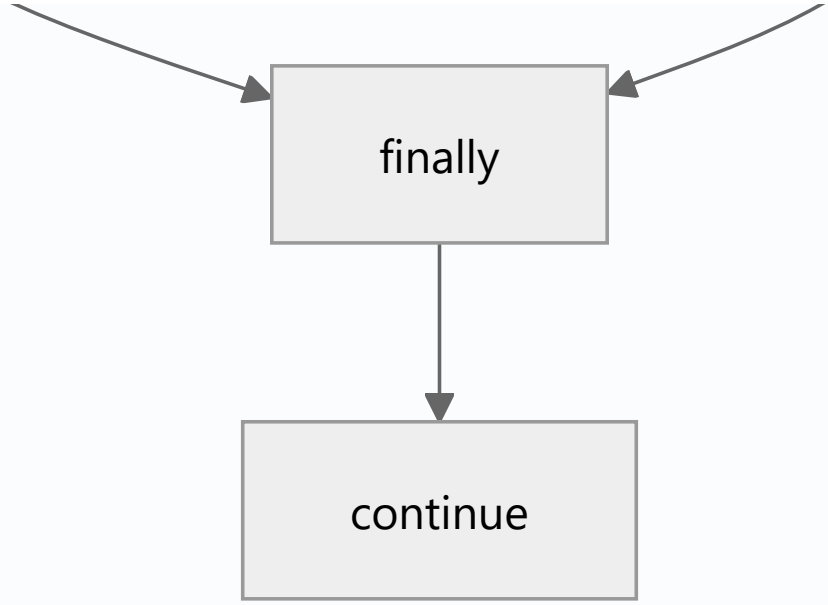
Examples

```
class InsufficientFundsException implements Exception {
  final double requested;
  final double available;
  final String? reason;
  InsufficientFundsException(
      {required this.requested, required this.available, this.reason});

  @override
  String toString() {
    final base = 'InsufficientFunds: requested '
        '${requested.toStringAsFixed(2)}, available '
        '${available.toStringAsFixed(2)}';
    return reason == null ? base : '$base ($reason)';
  }
}
```

```
class Account {
  double _balance;
  Account(this._balance);

  void withdraw(double amount) {
    if (amount <= 0) {
      throw ArgumentError.value(amount, 'amount', 'must be positive');
    }
    if (amount > _balance) {
      throw InsufficientFundsException(
          requested: amount, available: _balance);
    }
    _balance -= amount;
  }
}

void main() {
  final acc = Account(1000);
  try {
    acc.withdraw(99999);
  } on InsufficientFundsException catch (e) {
    print('Caught: $e'); // toString() runs automatically
  } on ArgumentError catch (e) {
    print('Bad arg: ${e.message}');
  } catch (e, st) {
    print('Unexpected: $e');
    print(st);
  } finally {
    print('done'); // always runs
  }
}
```

Diagrams


try block
throws T
on T?
match
no match
generic catch?
yes
no
no throw
handle
propagate up



Common Mistakes

Mistake	Why	Fix
`catch (_) {}` (swallow)	Hides bugs	Log + handle or `rethrow`
Catching `Error` types to "recover"	Errors are bugs, not conditions	Fix the bug; only catch `Exception`s to recover
General `catch` before `on T`	`on T` becomes unreachable	Order specific → general
Losing the stack trace	Harder debugging	Use `catch (e, st)` / `rethrow`
No `toString()` on custom exception	Prints `Instance of '...'`	Override `toString()`

Best Practices

- Model domain failures as **custom exceptions** (`implements Exception`) with a helpful `toString()` .
- Catch the **narrowest** type you can handle; let the rest propagate.
- Use `rethrow` to add context/log without erasing the original trace.
- Always release resources in `finally` (or use patterns that auto-close).
- Distinguish `Error` (fix) from `Exception` (handle); don't catch to hide programmer errors.

Performance

- Throwing is comparatively expensive (stack capture) — don't use exceptions for normal control flow in hot paths. Prefer returning a `Result` /sealed type for *expected* branching (see Module 38).

Advantages / Disadvantages

- + Cleanly separates happy path from error handling; carries rich context; centralizes recovery.
- − No compile-time enforcement (uncaught exceptions surface at runtime); overuse for control flow is slow and unclear.

Interview Questions

1. 🟢 **`Error` vs `Exception` in Dart?** — `Error` = programming bug you should fix (e.g., `RangeError`), generally not caught to recover; `Exception` = recoverable condition you handle (e.g., `FormatException`).
2. 🟢 **What does `finally` guarantee?** — It runs whether the `try` completes normally, throws, or returns.
3. 🟡 **`on` vs `catch`?** — `on T` filters by type; `catch (e)` catches anything; `catch (e, st)` also binds the stack trace. Order specific before general.
4. 🟡 **Where does the readable error message come from?** — The exception's `toString()`, called automatically by string interpolation (`'$e'`); `throw` only sends the object.
5. 🟡 **What is `rethrow` and why prefer it over `throw e`?** — Re-throws the caught exception **preserving the original stack trace**; `throw e` resets it.
6. 🔴 **Does Dart have checked exceptions?** — No; methods don't declare thrown types, so handling isn't compiler-enforced. Discipline/documentation fills the gap.
7. 🔴 **How are uncaught async errors handled?** — Via `runZonedGuarded`, `PlatformDispatcher.onError`, and `FlutterError.onError` in Flutter; awaited async errors surface in surrounding `try`/`catch`.

Senior Engineer Tips

- For *expected* failures (validation, not-found), prefer a typed `Result`/`Either` over throwing — it makes handling explicit and testable, and avoids exception cost.
- Throw only for *exceptional* conditions; reserve custom exceptions for domain-meaningful errors.
- Always attach context on `rethrow` (log the operation), and never let `catch` erase stack traces.

Architect Perspective

An error strategy is an architectural decision: define a failure taxonomy (domain, infrastructure, unexpected), decide where exceptions convert to `Result` types (usually the data/repository boundary), and install global handlers + logging/monitoring (Modules 38, 39, 52). Consistency here determines how debuggable and resilient the whole system is.

Summary

- `throw` sends an object; `try` / `on` / `catch` / `finally` handle and clean up; `rethrow` preserves the trace.
- `Error` = fix the bug; `Exception` = handle the condition.
- The readable message comes from `toString()`; model domain failures as custom exceptions; prefer `Result` types for expected branching.

Revision Notes

- `throw obj` sends object; message from `toString()` (auto via `$e`).
- `on T catch (e, st)`; order specific→general; `finally` always runs.
- `rethrow` keeps stack trace; no checked exceptions in Dart.
- `Error`=bug (don't recover), `Exception`=handle. Hot path → prefer `Result`.

Practice Questions

1. Why is `catch (_) {}` dangerous, and what should you do instead?
2. Explain why `throw e` inside a catch is worse than `rethrow`.
3. When should a failure be a `Result` value rather than a thrown exception?

Coding Questions

1. Write `int parsePositive(String s)` that throws `FormatException` for non-numbers and `ArgumentError` for negatives; test both.
2. Implement `T withResource<T>(Resource r, T Function() body)` that always closes `r` in `finally`.
3. Convert a throwing `fetchUser` into one returning `Result<User, AppFailure>` (sealed type).

Mini Project

Robust CSV importer: Parse a CSV of transactions; on a malformed row, throw a custom `RowParseException` carrying the row number and reason, catch per-row to collect errors (continue importing valid rows), and always close the reader in `finally`. Return `(imported, List<RowParseException> errors)`. Acceptance: meaningful `toString()`; no swallowed errors; valid rows still import; `dart analyze` clean.