02 · Advanced Dart

Flutter Practice Notes

Contents

02 · Advanced Dart

Introduction

Module 01 gave you the language. This module gives you the **runtime model and power features** that separate a Flutter developer from a Dart *engineer*: the single-threaded event loop, `Future` / `Stream` concurrency, isolates for true parallelism, generics, mixins, extensions, immutability patterns, and the VM's JIT/AOT compilation pipeline.

Why this module exists

Every jank bug, every "why didn't my `await` block the UI," every "how do I parse a 10MB JSON without freezing the app" traces back to the concepts here. Interviewers at top companies lean *hard* on the event loop, isolates, and streams because they reveal whether you understand what actually runs your code.

Module map

#	File	Topic	Level
1	01_event_loop.md	Event loop, microtask vs event queue, scheduling	🟢
2	02_async_futures.md	`Future`, `async` / `await`, error handling, `Completer`	🟢
3	03_streams.md	`Stream`, controllers, `async*`, transformers, broadcast	🟢
4	04_isolates.md	Isolates, `Isolate.run`, `compute`, ports, true parallelism	🔵
5	05_generics.md	Generic classes/methods, bounds, variance, advanced generics	🔵
6	06_mixins.md	`mixin`, `on`, linearization, diamond resolution	🔵
7	07_extension_methods.md	Add methods to existing types	🟢
8	08_extension_types.md	Zero-cost wrappers (Dart 3.3+)	🔵
9	09_constructors_and_singletons.md	Factory/named/private constructors, singletons, callable classes	🔵
10	10_immutability.md	Immutable objects, `copyWith`, value equality	🔵
11	11_libraries_and_packages.md	`library` / `part` / `import` / `export`, pub packages	🟢
12	12_json_and_serialization.md	`jsonEncode` / `Decode`, manual + codegen, annotations	🟢
13	13_memory_and_gc.md	Memory model, generational GC, leaks	🔴
14	14_dart_compilation.md	Dart VM, JIT, AOT, snapshots, tree shaking	🔴

Prerequisites

Complete **01 Dart Fundamentals** — especially functions/closures, null safety, and collections.

What you'll be able to do after this module

- Explain, on a whiteboard, exactly what happens when you `await` — and why the UI stays responsive.
- Choose between `async`, `Stream`, `Isolate`, and `compute` for a given workload.
- Write generic, reusable APIs with correct bounds and variance intuition.
- Reach for mixins, extensions, and immutability idiomatically.
- Describe how Dart compiles (JIT for hot reload, AOT for release) and why tree shaking matters for app size.

Capstones

Tier	Build
Beginner	An async **countdown + polling** CLI using `Future` / `Stream`.
Intermediate	A **rate-limited downloader** with a `Stream` progress API.
Advanced	A **parallel image/JSON processor** using isolates + `compute`.
Enterprise	A reusable `AsyncResult` **/retry/backoff** library (feeds Modules 16 & 38).

Summary

Advanced Dart is where correctness under concurrency and performance intuition are forged. Read the async core (event loop → futures → streams → isolates) first; it underpins everything in Flutter.

The Event Loop (Microtask Queue vs Event Queue)

> Dart runs your code on a single thread driven by an event loop that drains a high-priority microtask queue to empty before taking one event from the event queue — understand this and all async behavior becomes predictable.

Introduction

Each Dart isolate has **one thread** and **one event loop**. Concurrency in Dart is *not* parallelism by default: `async` / `await`, `Future`, and `Stream` all schedule work onto two queues that the single event loop drains in a strict order. This file explains that order — the key to reasoning about why UI stays responsive and in what sequence async callbacks run.

Why this concept exists

A single-threaded model avoids data races and locks entirely, which makes UI code dramatically simpler and safer. The event loop is how one thread juggles many pending operations (timers, I/O, gestures) without blocking. Knowing the queue priority lets you predict and control execution order.

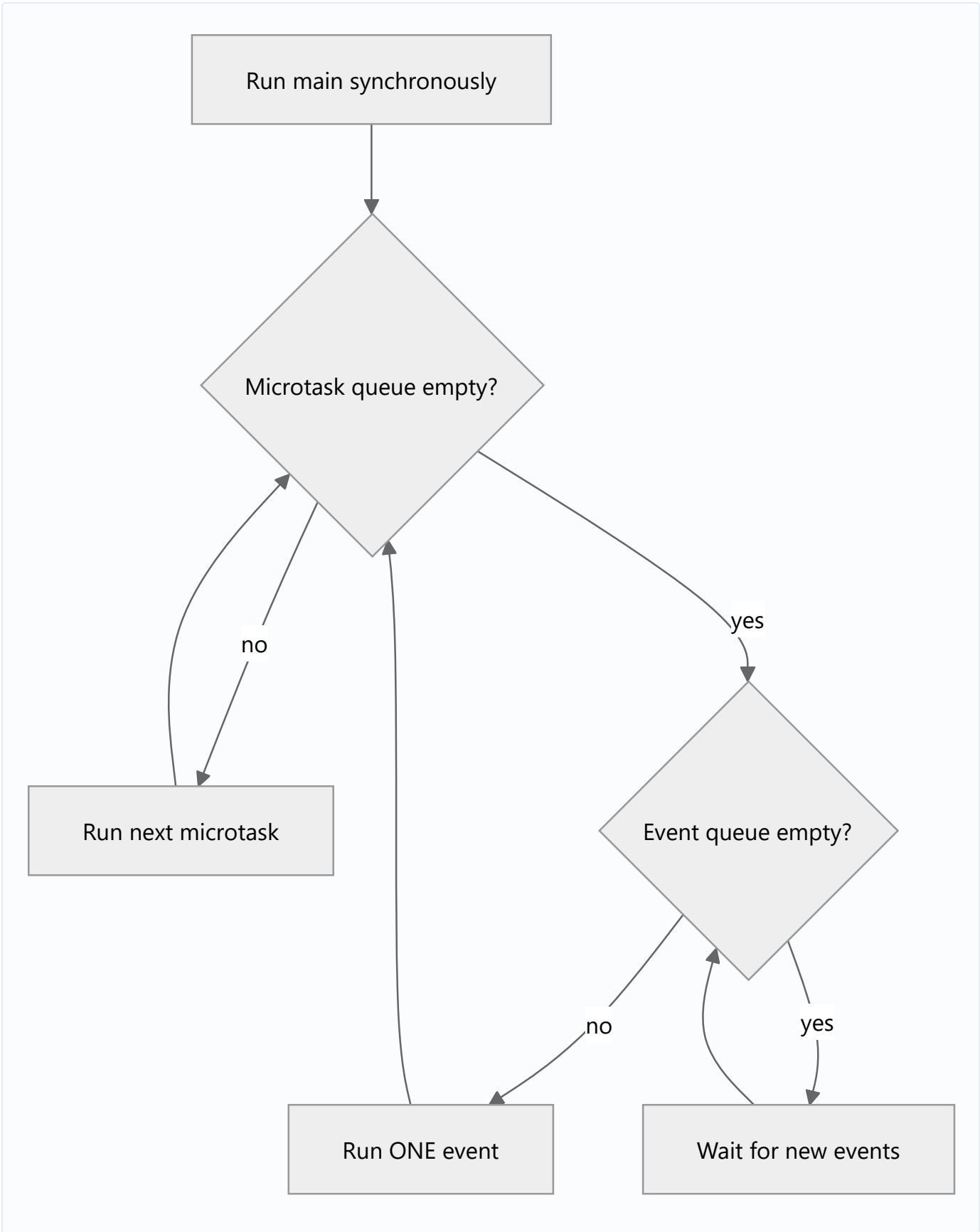
Real-world analogy

The event loop is a **barista** working alone. The **microtask queue** is a stack of "quick touch-ups I promised to finish before serving anyone new" (top up foam). The **event queue** is the **line of customers**. Rule: finish *all* pending touch-ups before calling the next customer — and after serving each customer, finish any new touch-ups before the next. If a touch-up spawns endless touch-ups, the customer line never moves (starvation).

Problem Statement

Predict the exact print order of a program mixing `print`, `Future(() => ...)`, `Future.microtask`, `scheduleMicrotask`, and `await`. By the end you'll order microtasks before events, every time.

Internal Working



Priority rules:

1. Run all **synchronous** code first (the current call stack).

2. Drain the **entire microtask queue** (scheduled via `scheduleMicrotask`, `Future.microtask`, and continuations of already-completed futures).
3. Take **one** item from the **event queue** (timers, I/O, gestures, `Future` from `Future(() => ...)` / `Future.delayed`).
4. After that one event, **drain microtasks again**, then next event. Repeat.

Microtasks always beat events. New microtasks created during microtask draining run *before* any event.

Memory Representation

- The two queues hold **closures** (callbacks) plus captured state. They live on the heap for the isolate.
- Each isolate owns its own loop + queues + heap; nothing is shared across isolates (see 04_isolates.md).

Compiler Behavior

- `async` / `await` is compiled into a **state machine**: the function body is split at each `await` into continuation callbacks scheduled as microtasks when the awaited future completes.
- `await` on an already-completed future still yields to the microtask queue (doesn't run synchronously).

Runtime Behavior

- `Future(() {})` / `Future.delayed` → **event** queue.
- `Future.microtask(() {})` / `scheduleMicrotask` → **microtask** queue.
- Continuations of `.then` / `await` on a completed future → **microtask**.
- A long synchronous computation **blocks the loop** → no events processed → UI jank/ANR.

Flutter Engine Behavior

The Flutter framework's frame scheduling (`SchedulerBinding`, `vsync`) posts frame work as events. If your synchronous work or a microtask flood blocks the loop, the engine can't produce frames on time → dropped frames (jank). This is *why* heavy work belongs in an isolate. See Module 09 Rendering Pipeline and Module 21 Performance.

Dart VM Behavior

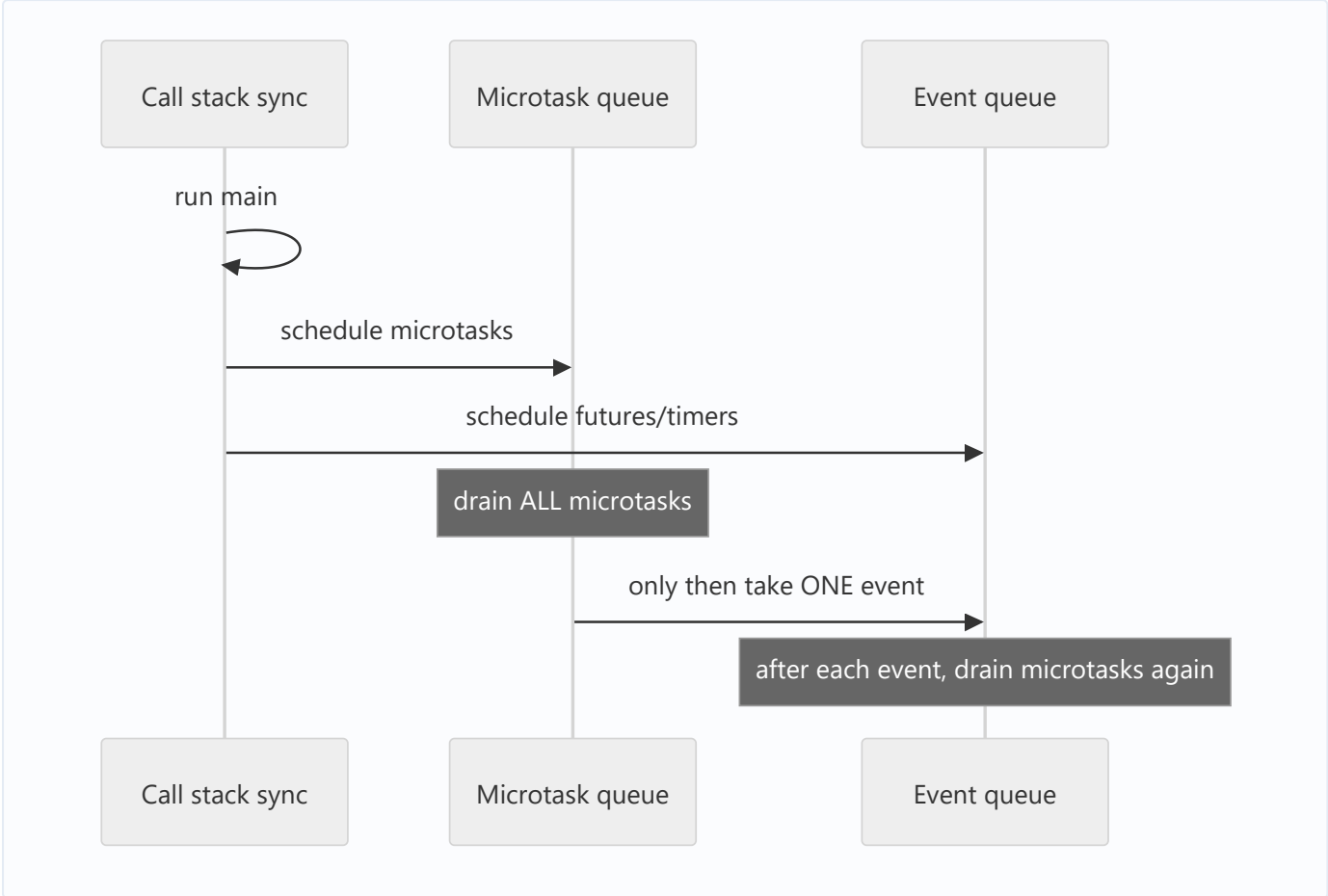
- The VM's isolate message loop implements these queues natively. Timer and I/O completions enqueue events; future continuations enqueue microtasks.

Examples

```
import 'dart:async';

void main() {
  print('1: sync start');

  Future(() => print('5: event (Future)')); // event queue

  Future.microtask(() => print('3: microtask A')); // microtask queue

  Future(() => print('6: event 2'))
      .then((_) => print('7: event 2 .then (microtask after its event)'));

  scheduleMicrotask(() => print('4: microtask B'));

  print('2: sync end');
}

// Output:
// 1: sync start
// 2: sync end
// 3: microtask A
// 4: microtask B
// 5: event (Future)
// 6: event 2
// 7: event 2 .then (microtask after its event)
```

```
// await yields even for a completed future:
Future<void> demo() async {
  print('A');
  await Future.value();      // suspends -> continuation is a microtask
  print('C');                // runs after current microtasks drain
}
void main() {
  demo();
  print('B'); // prints between A and C: A, B, C
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Expecting `Future(() {})` to run before a microtask	Events are lower priority	Use `Future.microtask` if you need microtask timing
Heavy CPU work in an `async` function	`async` ≠ another thread; still blocks the loop	Offload to an isolate/ `compute`
Microtask recursion	Starves the event queue (UI freezes)	Break work across events (`Future(() ...)`)
Assuming `await` runs synchronously on completed futures	It always yields to microtask	Don't rely on synchronous continuation

Best Practices

- Keep synchronous work per event **short** (target < a frame budget, ~16ms at 60fps).
- Use microtasks only for tiny, must-happen-before-next-event work.
- Chunk large loops across events, or move them to an isolate.

- Never busy-wait; yield with `await Future.delayed(Duration.zero)` to let the loop breathe (sparingly).

Performance

- Loop-blocking is the #1 cause of jank. The fix is architectural: isolate CPU-bound work; stream I/O results.
- Microtask floods are subtle jank sources — profile with DevTools if frames drop with no obvious cause.

Advantages / Disadvantages

- + No locks/races; simple mental model; responsive I/O without threads.
- − One blocking call freezes everything; true parallelism needs isolates.

Interview Questions

1. 🟢 **Is Dart single-threaded?** — Each isolate is single-threaded with one event loop; parallelism requires multiple isolates.
2. 🟢 **Microtask vs event queue priority?** — The loop drains the *entire* microtask queue before taking *one* event, and re-drains microtasks after every event. Microtasks always win.
3. 🟡 **What schedules a microtask vs an event?** — Microtask: `scheduleMicrotask`, `Future.microtask`, `.then` / `await` continuations. Event: `Future(() {})`, `Future.delayed`, timers, I/O, gestures.
4. 🟡 **Does `await` on a completed future run synchronously?** — No; it schedules the continuation as a microtask and yields.
5. 🟡 **Why does heavy work in `async` still cause jank?** — `async` doesn't create a thread; the CPU work runs on the same loop and blocks frame events.
6. 🔴 **How can microtasks starve the UI?** — A microtask that keeps scheduling more microtasks prevents the loop from ever reaching the event queue (where frames live).
7. 🔴 **How does `async` / `await` compile?** — Into a state machine split at each `await`; continuations are scheduled as microtasks when the awaited future completes.

Senior Engineer Tips

- When debugging ordering bugs, mentally tag each callback "microtask" or "event" — the order falls out immediately.
- Prefer `Future.delayed(Duration.zero)` over microtasks to intentionally *yield to the event queue* (let a frame render).

- Treat "UI froze for 300ms" as "something blocked the loop" — search for sync loops, big JSON parses, or crypto on the main isolate.

Architect Perspective

The event-loop model dictates your concurrency architecture: I/O-bound work → `async` / `Stream` on the main isolate; CPU-bound work → isolates. Designing this boundary correctly (a "compute offload" layer) is what keeps large apps at 60/120fps. It also shapes testability — deterministic async ordering makes `fakeAsync` tests reliable (Module 49).

Summary

- One thread, one loop, two queues: drain all microtasks, then one event, repeat.
- Microtasks (`scheduleMicrotask` , `.then` / `await` continuations) outrank events (timers, I/O, `Future(() {})`).
- `async` isn't a thread; blocking the loop blocks the UI — offload CPU work to isolates.

Revision Notes

- Sync first → drain ALL microtasks → ONE event → re-drain microtasks → repeat.
- Microtask: `scheduleMicrotask` , `Future.microtask` , `.then` / `await` . Event: `Future(() {})` , `Future.delayed` , timers, I/O.
- `await` always yields (even on completed future).
- Blocking the loop = jank; CPU work → isolate.

Practice Questions

1. Order the output of a snippet mixing `Future` , `Future.microtask` , and `print` .
2. Explain why `async` can't fix a CPU-bound freeze.
3. What is microtask starvation and how would you detect it?

Coding Questions

1. Write a program that prints `A,B,C,D` in a specific order using one sync line, one microtask, and one event — and justify the order.
2. Implement `yieldToEventLoop()` and use it to chunk a 1M-iteration loop so a periodic timer still fires.
3. Demonstrate `.then` on a completed future running as a microtask before a pending event.

Mini Project

Event-loop visualizer (CLI): Build a small program that schedules a mix of sync statements, microtasks, and events with labels, and prints them in execution order with a running "queue state" comment. Add a deliberately loop-blocking function and a chunked version; show a timer that fires on time only in the chunked version. Acceptance: predicted vs actual order documented; blocking vs chunked behavior demonstrated.

Futures & `async` / `await`

> A `Future<T>` is a promise of a value (or error) that will arrive later; `async` / `await` lets you write asynchronous code that reads like synchronous code without blocking the event loop.

Introduction

A `Future<T>` represents a computation that completes **later** with either a value of type `T` or an error. `async` / `await` is syntactic sugar over `Future` that removes callback nesting. This file covers creating, chaining, awaiting, combining, and error-handling futures — the daily grammar of I/O in Flutter.

Why this concept exists

Network calls, file reads, and timers don't finish instantly. Blocking the single thread until they do would freeze the UI. Futures let you *register interest* in a later result and let the event loop keep running. `async` / `await` makes that readable, turning callback pyramids into linear code.

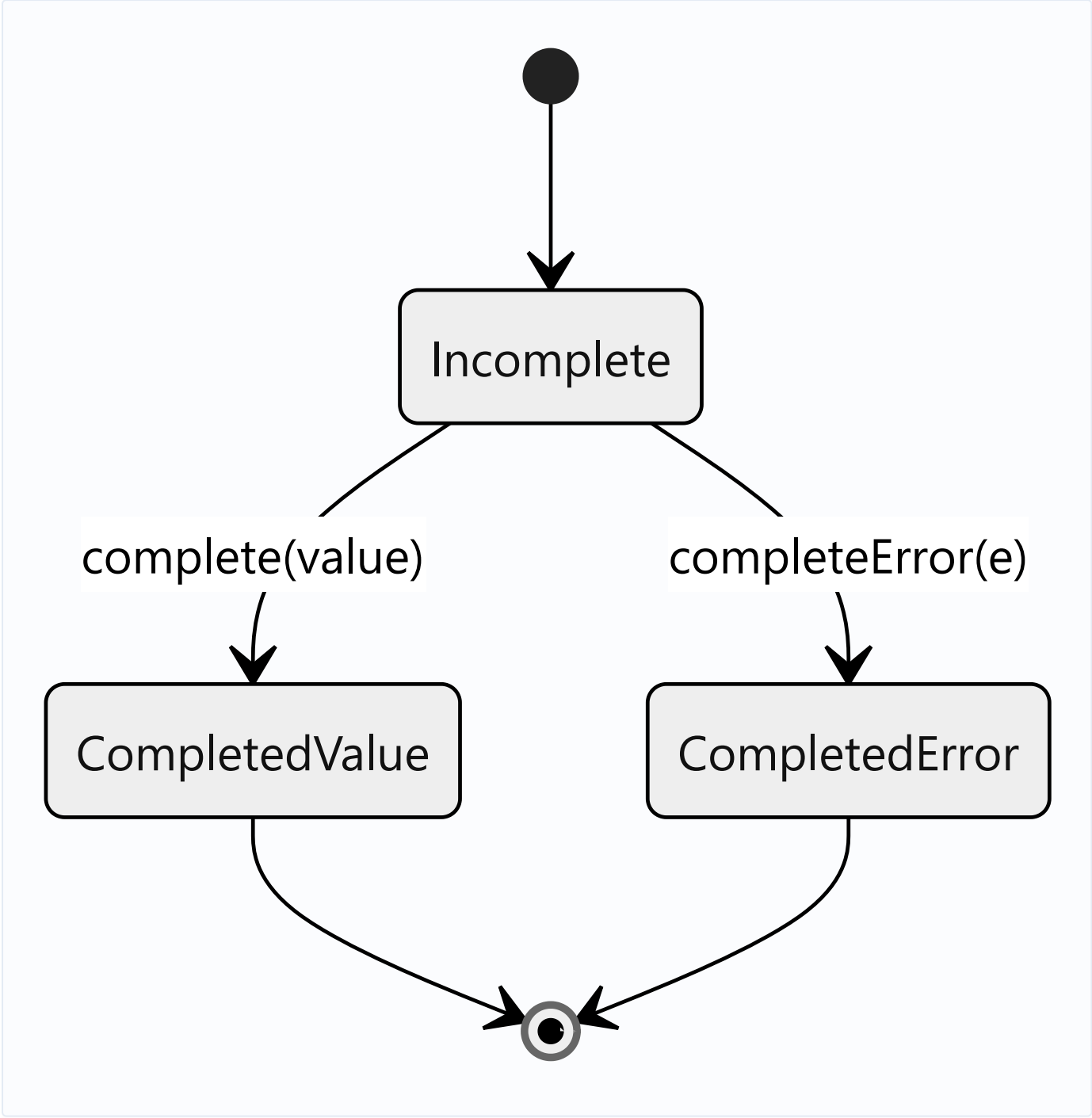
Real-world analogy

Ordering coffee with a **buzzer**: you place the order (start the future), take the buzzer, and go sit down (the thread keeps working). When it buzzes (the future completes), you collect your coffee (the value) — or the barista tells you they're out (an error). `await` is "sit and do nothing *of your own* until it buzzes," but the café (event loop) keeps serving others.

Problem Statement

Fetch a user, then their orders, handle a network error gracefully, add a timeout, and fetch three independent resources in parallel. You'll use `await`, `try` / `catch`, `.timeout`, and `Future.wait`.

Internal Working



- A `Future` is a single-shot container: it completes **once**, with a value or an error.
- `.then(onValue, onError:)` / `.catchError` / `.whenComplete` register callbacks (run as microtasks on completion).
- `async` functions **always return a** `Future`; `await` suspends the function, scheduling the rest as a continuation (microtask) when the awaited future completes.
- `Completer<T>` lets you create and complete a future manually (bridging callback APIs).

Memory Representation

- A `Future` object holds its state (pending/value/error) and a list of registered continuation closures. Those closures capture their surrounding variables until the future completes and they run.

Compiler Behavior

- `async` / `await` compiles to a **state machine** (see 01_event_loop.md); each `await` is a suspension point.
- Returning a value from an `async` function wraps it in `Future.value`; throwing inside becomes a completed-with-error future.
- The analyzer warns on **unawaited** futures (`unawaited_futures`) — a common bug source.

Runtime Behavior

- Awaiting a completed future still yields to the microtask queue (never fully synchronous).
- Errors in an awaited future are **rethrown** at the `await` point, catchable by a surrounding `try` / `catch` .
- An unhandled future error becomes an uncaught async error (goes to the zone/ `PlatformDispatcher.onError`).

Flutter Engine Behavior

Not applicable at engine level. Flutter's `FutureBuilder` subscribes to a future and rebuilds on completion; awaited I/O keeps frames flowing because the loop isn't blocked.

Dart VM Behavior

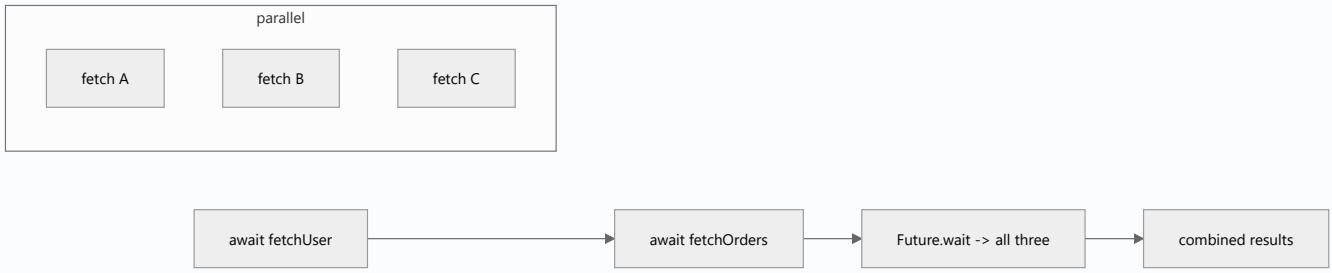
- Continuations are scheduled as microtasks; the VM's async infrastructure integrates with the isolate loop.

Examples

```
import 'dart:async';


Future<String> fetchUser() async =>
    Future.delayed(const Duration(milliseconds: 100), () => 'Ada');


Future<List<String>> fetchOrders(String user) async =>
    Future.delayed(const Duration(milliseconds: 100), () => ['o1', 'o2']);
```

```
Future<void> main() async {
  // sequential dependency
  try {
    final user = await fetchUser();
    final orders = await fetchOrders(user);
    print('$user has ${orders.length} orders'); // Ada has 2 orders
  } on TimeoutException {
    print('timed out');
  } catch (e) {
    print('failed: $e');
  }

  // timeout
  final slow = Future.delayed(const Duration(seconds: 5), () => 'late');
  try {
    await slow.timeout(const Duration(milliseconds: 200));
  } on TimeoutException {
    print('gave up waiting'); // this prints
  }

  // parallel independent work (fan-out)
  final results = await Future.wait([
    Future.delayed(const Duration(milliseconds: 50), () => 1),
    Future.delayed(const Duration(milliseconds: 80), () => 2),
    Future.delayed(const Duration(milliseconds: 30), () => 3),
  ]);
  print(results); // [1, 2, 3] — waits for all, ~80ms not 160ms

  // Completer: bridge a callback API to a Future
  final completer = Completer<int>();
  Timer(const Duration(milliseconds: 10), () => completer.complete(42));
  print(await completer.future); // 42
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Awaiting independent calls sequentially	Wastes time (sum of durations)	`Future.wait([...])` for parallel
Forgetting to `await` (fire-and-forget)	Errors vanish, ordering bugs	`await` or `unawaited(...)` deliberately
`try` / `catch` around a non-awaited future	Won't catch its error	Await inside the try, or use `.catchError`
CPU-bound work in `async`	Blocks the loop	Use isolates/ `compute`
`Future.wait` with one failure	Rejects whole batch	Use `eagerError:false` or wrap each with `catchError`

Best Practices

- Parallelize independent awaits with `Future.wait`.
- Always handle errors: `try` / `catch` around awaits, or `.catchError`.
- Add `.timeout` to network calls.
- Mark intentional fire-and-forget with `unawaited(...)`.
- Return `Future<void>` (not `void`) from async APIs so callers can await.

Performance

- `Future.wait` turns N sequential I/O waits into one parallel wait (~max, not sum).
- Avoid creating unnecessary futures in hot paths; each schedules a microtask.

Advantages / Disadvantages

- + Readable async, composition (`wait` / `any`), integrated error handling, timeouts.
- − Single-shot only (use `Stream` for multiple values); easy to forget `await`; doesn't parallelize CPU work.

Interview Questions

1. 🟢 **What is a `Future`?** — A container for a value/error that becomes available later; completes exactly once.
2. 🟢 **Does an `async` function always return a Future?** — Yes; a returned value is wrapped in `Future.value`, a throw becomes a completed-with-error future.
3. 🟡 **How do you run two independent async calls in parallel?** — Start both (don't await individually), then `await Future.wait([a, b])`.
4. 🟡 **Where is an awaited future's error caught?** — It's rethrown at the `await`; a surrounding `try`/`catch` handles it.
5. 🟡 **`Future` vs `Stream`?** — `Future` = one async value; `Stream` = zero or more async values over time.
6. 🔴 **What is a `Completer` for?** — To create a future you complete manually — bridging callback-based APIs into the future world.
7. 🔴 **Why can awaiting still not run synchronously on a completed future?** — Continuations are scheduled as microtasks; `await` always yields to the loop.

Senior Engineer Tips

- `Future.wait` fails fast by default; for "collect all results and errors," map each to a result-capturing wrapper before `wait`.
- Prefer returning typed failures (`Result`) over throwing across async boundaries for expected errors (Module 38).
- Beware `await` in loops — it serializes; use `Future.wait(list.map(...))` for concurrency (bounded, to avoid overload).

Architect Perspective

Async design shapes latency and resilience: fan-out with `Future.wait`, add timeouts/retries/backoff, and convert exceptions to typed failures at the repository boundary. A consistent async policy (timeouts everywhere, bounded concurrency, cancellation strategy) is a hallmark of production-grade apps and directly affects perceived speed.

Summary

- `Future<T>` = one later value/error; `async`/`await` = readable, non-blocking async.
- Parallelize with `Future.wait`; guard with `try`/`catch` and `.timeout`.
- `async` doesn't parallelize CPU work — that's isolates.

Revision Notes

- `async` fn always returns a `Future`; `await` suspends (continuation = microtask).
- Parallel: `Future.wait`; timeout: `.timeout`; manual: `Completer`.
- Await errors caught by surrounding try/catch; unawaited errors go to zone.
- CPU work → isolate, not `async`.

Practice Questions

1. Rewrite three sequential awaits that are independent to run in parallel; estimate the time saved.
2. Why does a `try/catch` around an un-awaited future fail to catch its error?
3. When do you need a `Completer`?

Coding Questions

1. Implement `Future<T> retry<T>(Future<T> Function() op, {int times, Duration delay})`.
2. Implement `Future<T> withTimeout<T>(Future<T> f, Duration d, {T Function()? onTimeout})`.
3. Write `Future<List<R>> mapConcurrent<T,R>(Iterable<T> xs, Future<R> Function(T) f, {int concurrency})` (bounded parallelism).

Mini Project

Resilient fetch layer (pure Dart): Build a `fetch` helper with timeout, exponential-backoff retry, and `Future.wait`-based parallel fetching of multiple simulated endpoints, returning a combined typed result and collecting per-endpoint errors. Acceptance: no unawaited futures; timeouts and retries covered by tests using injected fake delays; `dart analyze` clean.

Streams (`Stream`, controllers, `async*`, transformers, broadcast)

> A `Stream<T>` is a `Future` that can deliver many values over time — the backbone of reactive Flutter (BLoC, `StreamBuilder`, gesture/sensor/websocket feeds).

Introduction

Where a `Future` yields one value, a `Stream<T>` yields **zero or more** values (and possibly an error, and a done signal) over time. This file covers consuming streams (`await for`, `listen`), producing them (`async*` / `yield`, `StreamController`), transforming them (`map` / `where` / `transform`), and the crucial **single-subscription vs broadcast** distinction.

Why this concept exists

Many data sources are *ongoing*: user input, sensor readings, websocket messages, database change feeds, timers. Modeling them as a pull-once `Future` doesn't fit. Streams provide a uniform, composable abstraction for asynchronous *sequences*, with backpressure and cancellation built in.

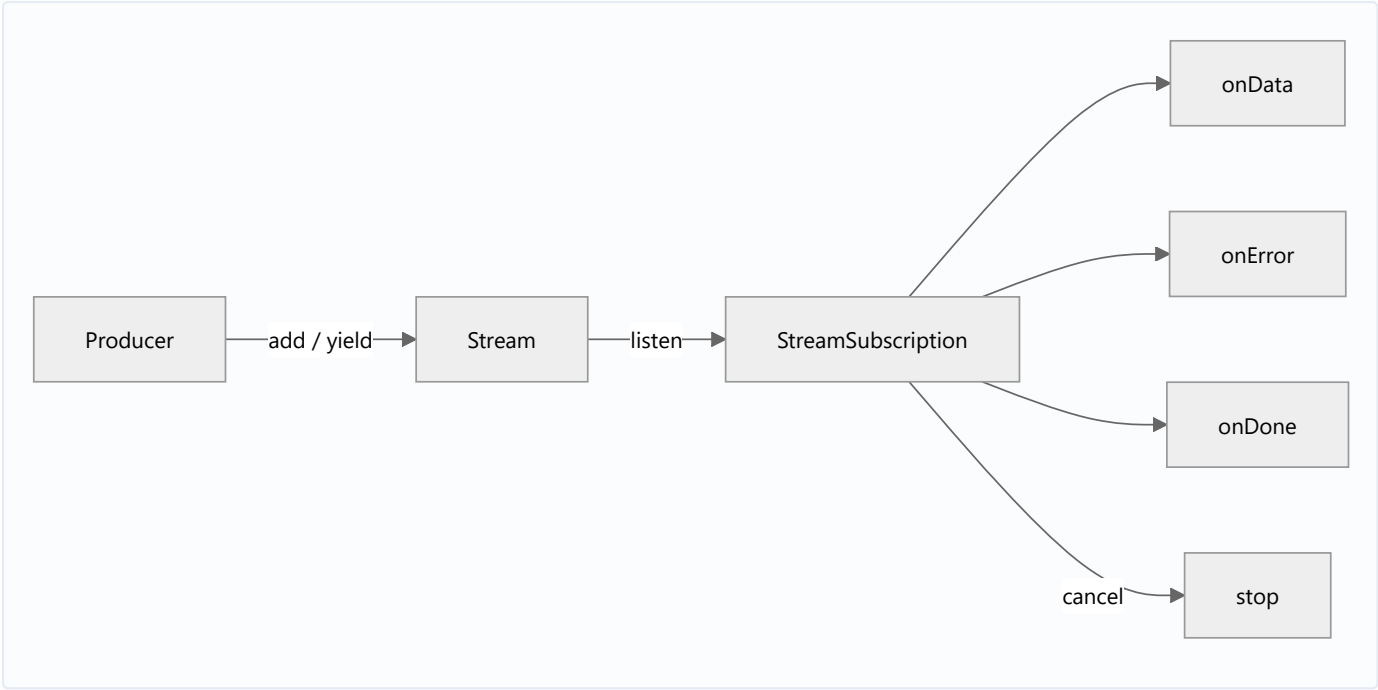
Real-world analogy

A `Future` is a **letter** (arrives once). A `Stream` is a **subscription to a magazine**: issues arrive over time until you cancel. A **single-subscription** stream is a magazine with exactly one subscriber slot; a **broadcast** stream is a radio station many listeners can tune into simultaneously.

Problem Statement

Emit a countdown, transform sensor values, multiplex one event source to several widgets, and always cancel subscriptions to avoid leaks. You'll use `async*`, `StreamController.broadcast`, transformers, and `StreamSubscription.cancel`.

Internal Working



- **Single-subscription** (default): one listener; buffers events until listened; errors if listened twice. For results of a single async sequence (file read, HTTP body).
- **Broadcast**: many listeners; events not buffered for late listeners. For shared event sources (user events, app-wide notifications).
- Produce with `async*` + `yield` / `yield*`, or imperatively with a `StreamController`.
- Consume with `await for` (in an `async` function) or `.listen(onData, onError, onDone, cancelOnError)`.
- Transform lazily: `map`, `where`, `expand`, `take`, `distinct`, `asyncMap`, `transform(StreamTransformer)`.

Memory Representation

- A `StreamController` holds a buffer (single-sub) or a listener list (broadcast) plus callbacks. Subscriptions are heap objects; **not cancelling them leaks** the subscription and anything it captures.

Compiler Behavior

- `async*` compiles to a stream-generating state machine; `yield` emits a value, `yield*` delegates to another stream, execution pauses when the consumer pauses (backpressure).
- `await for` desugars into listen + pause/resume handling.

Runtime Behavior

- Single-subscription streams **buffer** events until first `listen`; broadcast streams **drop** events that occur before a listener subscribes.
- Pausing a subscription pauses an `async*` producer (backpressure); resuming continues it.
- `cancelOnError` controls whether the subscription ends on the first error.

Flutter Engine Behavior

Not applicable at engine level. `StreamBuilder` listens and rebuilds on each event; framework input (gestures) and platform channels surface as streams.

Dart VM Behavior

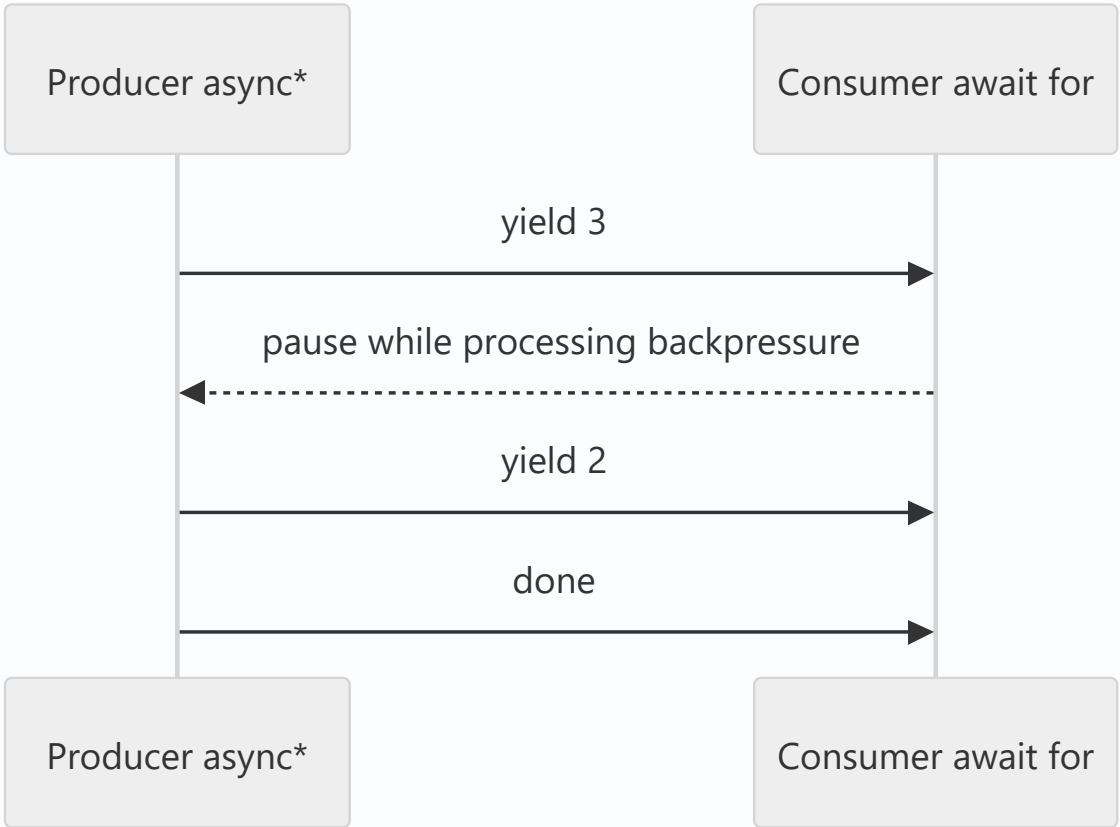
- Stream events are delivered via microtasks/events on the isolate loop; `async*` producers integrate with the scheduler for pause/resume.

Examples

```
import 'dart:async';

// produce with async* + backpressure
Stream<int> countdown(int from) async* {
  for (var i = from; i >= 0; i--) {
    await Future.delayed(const Duration(milliseconds: 50));
    yield i;
  }
}

Future<void> main() async {
  // consume with await for
  await for (final n in countdown(3)) {
    print('tick $n'); // 3,2,1,0
  }

  // transform lazily
  final evens = countdown(5).where((n) => n.isEven).map((n) => 'E$n');
  print(await evens.toList()); // [E4, E2, E0]

  // broadcast: multiple listeners
```

```
  final ctrl = StreamController<String>.broadcast();
  final s1 = ctrl.stream.listen((e) => print('L1: $e'));
  final s2 = ctrl.stream.listen((e) => print('L2: $e'));
  ctrl.add('hello'); // both L1 and L2 receive it
  await Future.delayed(Duration.zero);

  // ALWAYS cancel to avoid leaks
  await s1.cancel();
  await s2.cancel();
  await ctrl.close();

  // single-subscription buffers until listened:
  final single = StreamController<int>();
  single.add(1);
  single.add(2);
  single.stream.listen((e) => print('buffered $e')); // 1, 2
  await single.close();
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not cancelling subscriptions	Memory leaks, callbacks after dispose	`cancel()` in `dispose` / `onDone`
Listening to a single-sub stream twice	Throws	Use `.broadcast()` or `.asBroadcastStream()`
Expecting late broadcast listeners to get past events	Broadcast doesn't buffer	Use a `BehaviorSubject` (rxdart) or replay
Not closing `StreamController`	Leak; `done` never fires	`close()` when finished
Heavy sync work in `onData`	Blocks the loop	Offload/throttle

Best Practices

- Choose single-subscription for one-shot sequences, broadcast for shared events.
- Cancel subscriptions in `dispose`; close controllers you own.
- Prefer `async*` for producing derived streams; use transformers for reusable pipelines.
- Use `distinct`, `debounce` / `throttle` (rxdart) to tame chatty sources.
- Expose `Stream<T>` (read-only) from APIs; keep the `StreamController` private.

Performance

- Broadcast to many listeners is cheap per-event but each listener's `onData` runs on the loop — keep handlers light.
- Backpressure (pause/resume) prevents unbounded buffering from fast producers.

Advantages / Disadvantages

- **+** Composable async sequences, backpressure, cancellation, reactive UI.
- **−** More moving parts than futures; leak-prone if subscriptions aren't cancelled; broadcast doesn't replay.

Interview Questions

1. 🟢 **`Future` vs `Stream`?** — Future: one async value. Stream: many async values over time (plus errors/done).
2. 🟢 **Single-subscription vs broadcast?** — Single: exactly one listener, buffers until listened. Broadcast: many listeners, no buffering for late subscribers.

3. 🟡 **How do you produce a stream?** — `async*` with `yield` / `yield*`, or imperatively via a `StreamController`.
4. 🟡 **What is backpressure and how do streams handle it?** — When the consumer is slower than the producer; pausing a subscription pauses an `async*` producer so it doesn't overproduce.
5. 🟡 **Why must you cancel subscriptions?** — An active subscription keeps the callback (and its captures, e.g., `State` / `context`) alive → memory leak and post-dispose callbacks.
6. 🔴 **How do you make late broadcast listeners see the latest value?** — Use a replay/behavior subject (rxdart) or cache the last value and re-emit on subscribe; core Dart broadcast doesn't buffer.
7. 🔴 `await for` **vs** `.listen`**?** — `await for` consumes sequentially inside an `async` function (auto pause between iterations); `.listen` is callback-based and non-blocking with explicit lifecycle handlers.

Senior Engineer Tips

- Treat every `.listen` as a resource with an owner responsible for `cancel()`. In Flutter, store subscriptions and cancel in `dispose`.
- For UI, `StreamBuilder` handles subscribe/cancel for you — but mind that it rebuilds per event; select/throttle upstream.
- rxdart's `BehaviorSubject` / `debounceTime` / `switchMap` cover most real-world reactive needs; reach for them over hand-rolling.

Architect Perspective

Streams are the substrate of reactive architectures (BLoC, MVVM with reactive bindings). Deciding what is a stream (ongoing state/events) vs a future (one-shot command) shapes your state-management design (Module 11). Enforce subscription-lifecycle discipline app-wide to prevent leaks at scale.

Summary

- `Stream<T>` delivers many async values; consume via `await for` / `.listen`, produce via `async*` / `StreamController`.
- Single-subscription buffers for one listener; broadcast serves many without buffering.
- Always cancel subscriptions and close controllers; use backpressure and transformers.

Revision Notes

- Stream = many values over time; Future = one.

- Produce: `async*` + `yield`, or `StreamController(.broadcast)`. Consume: `await for` / `.listen`.
- Single-sub buffers + one listener; broadcast many + no replay.
- Cancel subscriptions + close controllers (leaks!). Backpressure via pause/resume.

Practice Questions

1. When would you pick broadcast over single-subscription?
2. Why can a forgotten subscription cause a `setState after dispose` error?
3. What's the difference between `map` and `asyncMap` on a stream?

Coding Questions

1. Implement `Stream<int> range(int start, int end)` with `async*`.
2. Build a `Debouncer` that exposes a debounced `Stream<String>` from raw input events.
3. Implement `Stream<T> merge<T>(List<Stream<T>> streams)` using a broadcast controller.

Mini Project

Live search pipeline (pure Dart): Simulate keystrokes into a `StreamController`, apply `debounce` + `distinct` + `switchMap`-style cancellation so only the latest query's results emit, and print results. Ensure all subscriptions/controllers are cancelled/closed. Acceptance: stale queries never emit; no leaks (all cancelled); tests drive fake keystrokes and assert only latest results appear.

Isolates (True Parallelism)

> An isolate is Dart's unit of concurrency with its *own* memory and event loop; because isolates share nothing, they achieve real parallelism without locks — by passing messages, not memory.

Introduction

`async` / `Future` / `Stream` give **concurrency** on one thread; they do *not* run CPU work in parallel. **Isolates** do: each is an independent worker with its own heap and event loop. This file covers why isolates exist, `Isolate.run`, Flutter's `compute`, ports for messaging, and the "share nothing" model.

Why this concept exists

The event loop is single-threaded, so a heavy computation (parse a 20MB JSON, resize an image, run crypto) **blocks the UI**. Isolates move that work to another thread. Dart chose "share nothing + message passing" over shared-memory threads to eliminate data races and locks entirely — safety by design.

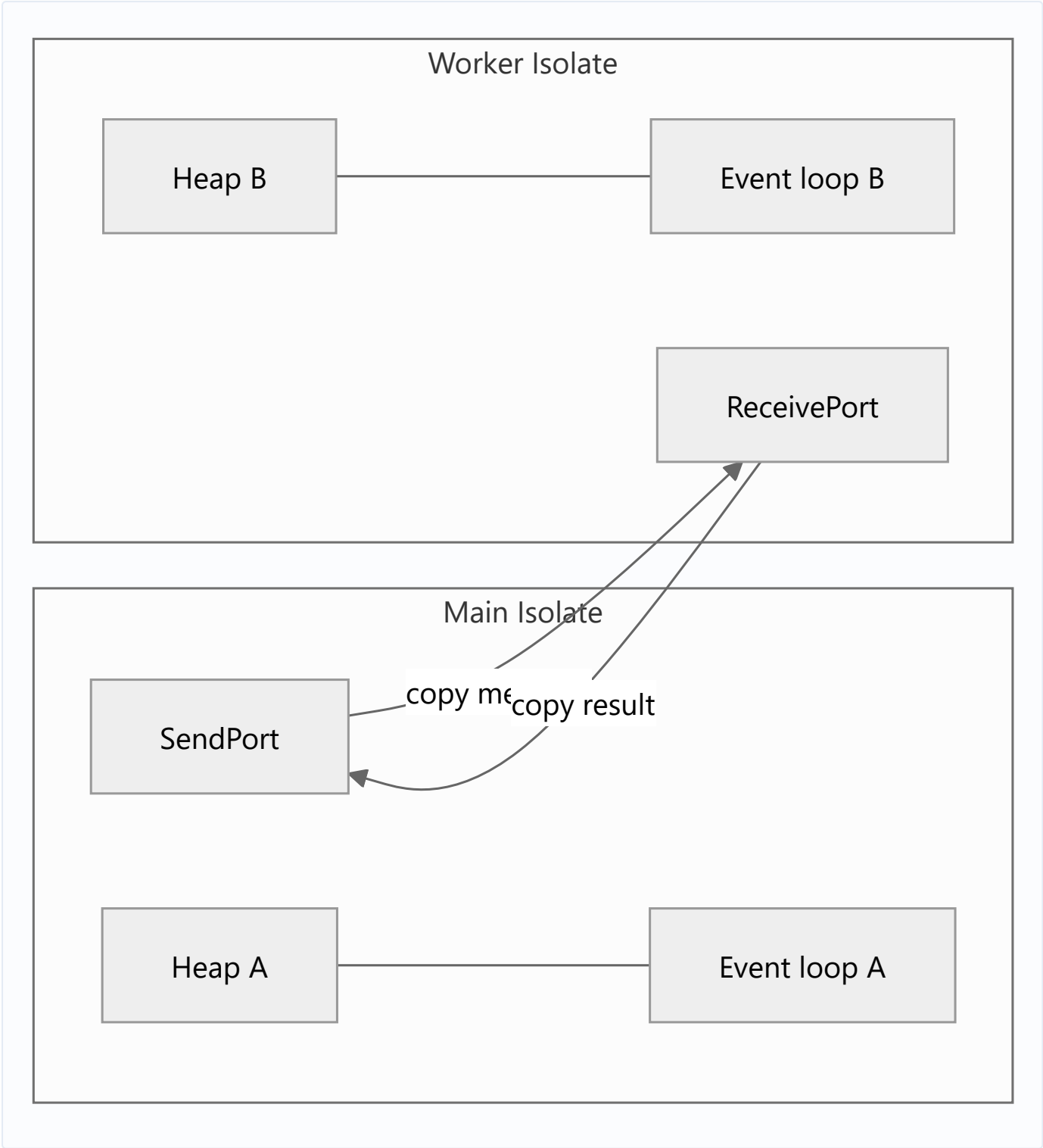
Real-world analogy

Isolates are **separate kitchens**, each with its own ingredients (heap) and cook (loop). They can't reach into each other's fridges (no shared memory). To collaborate, they pass **sealed containers through a hatch** (messages over ports). Two kitchens cook simultaneously (true parallelism) without fighting over the same knife (no locks).

Problem Statement

Your app freezes for 500ms parsing a large JSON on tap. Move the parse off the UI isolate so frames keep rendering, get the result back, and understand what data can cross the boundary. You'll use `compute` / `Isolate.run`.

Internal Working



- Each isolate has a **private heap** and its own event loop. No shared mutable state.
- Communication is via **ports**: `SendPort` / `ReceivePort`. Messages are **copied** (deep) between isolates (except a few transferable types).
- `Isolate.run(fn)` (modern, Dart 2.19+) runs a function on a fresh isolate and returns its result as a `Future` — the easiest path.
- Flutter's `compute(fn, arg)` is a thin wrapper spawning a one-shot isolate.
- For long-lived workers, spawn with `Isolate.spawn` and set up two-way port communication.

Memory Representation

- Objects passed to another isolate are **deep-copied** into that isolate's heap (structured-clone-like). Large messages cost copy time + memory.
- `TransferableTypedData` moves byte buffers with zero-copy transfer of ownership (for big binary payloads).
- Closures passed to `Isolate.run` must not capture non-sendable state; top-level/static functions are safest.

Compiler Behavior

- The entry function for `Isolate.spawn` must be a top-level or static function (it's referenced by a sendable closure); the compiler/runtime enforces sendability of messages.

Runtime Behavior

- Spawning an isolate has real cost (memory + startup ~ms); for tiny tasks the copy+spawn overhead can exceed the savings.
- Sending non-sendable objects (e.g., a `SendPort` is fine, but a socket or a closure over UI state is not) throws at runtime.

Flutter Engine Behavior

Flutter runs the UI on the **main (root) isolate**; the engine's raster/IO threads are separate. Heavy Dart work must leave the UI isolate (via `compute` / `Isolate.run`) to avoid dropped frames. `RootIsolateToken` + `BackgroundIsolateBinaryMessenger` are needed if a background isolate must use platform channels/plugins. See Module 21 Performance and Module 33 Background Services.

Dart VM Behavior

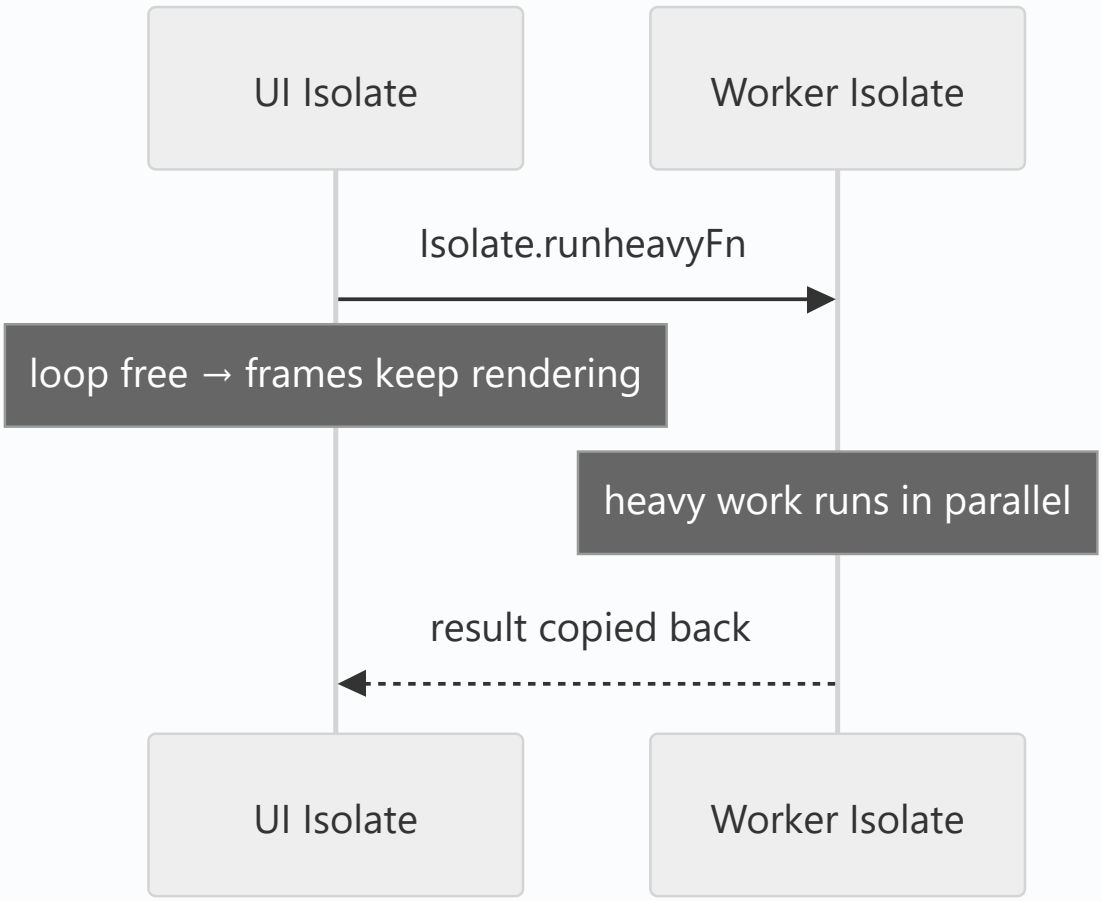
- The VM schedules isolates across OS threads (an isolate group can share the runtime/code, reducing memory in newer VMs). AOT isolates start faster than one might expect but still aren't free.

Examples

```
import 'dart:convert';

import 'dart:isolate';


// Must be a top-level or static function for Isolate.run/spawn.

int _sumOfSquares(int n) {
```

```
  var total = 0;
  for (var i = 1; i <= n; i++) {
    total += i * i;
  }
  return total;
}

Map<String, dynamic> _parse(String raw) =>
    jsonDecode(raw) as Map<String, dynamic>;

Future<void> main() async {
  // Modern one-shot parallel work — does NOT block the caller's loop:
  final result = await Isolate.run(() => _sumOfSquares(100000000));
  print('sum = $result');

  // Offload a heavy parse (in Flutter you'd use compute(_parse, raw)):
  final parsed = await Isolate.run(() => _parse('{"a":1,"b":2}'));
  print(parsed); // {a: 1, b: 2}

  // Long-lived worker with two-way messaging:
  final rp = ReceivePort();
  await Isolate.spawn(_worker, rp.sendPort);
  final workerSendPort = await rp.first as SendPort;

  final answer = ReceivePort();
  workerSendPort.send([21, answer.sendPort]);
  print(await answer.first); // 42
}

void _worker(SendPort toMain) {
  final rp = ReceivePort();
  toMain.send(rp.sendPort);
  rp.listen((msg) {
    final value = msg[0] as int;
    final reply = msg[1] as SendPort;
    reply.send(value * 2); // runs on the worker isolate, in parallel
  });
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using `async` to "parallelize" CPU work	Same thread; still blocks	Use `Isolate.run` / `compute`
Spawning an isolate for tiny work	Spawn+copy cost > savings	Keep small work on main; batch
Passing non-sendable objects	Throws at runtime	Send plain data; rebuild objects in the isolate
Using plugins in a background isolate without setup	Platform channel unavailable	Register `RootIsolateToken` /background messenger
Forgetting to close `ReceivePort`	Leak; isolate stays alive	`close()` ports and `kill()` workers when done

Best Practices

- Reach for `Isolate.run` / `compute` for any main-isolate work exceeding a frame budget (JSON parse, image/crypto/compression).
- Send **immutable plain data**; use `TransferableTypedData` for large byte buffers.

- Pool/reuse long-lived isolates for repeated heavy tasks instead of respawning.
- Measure: only offload when the copy+spawn cost is worth it.

Performance

- True parallelism: N isolates use N cores for CPU-bound work.
- Cost model: `spawn` (~ms + memory) + message copy (∝ payload size). For big results, prefer transferable buffers or streaming partial results back.

Advantages / Disadvantages

- + Real parallelism, no locks/races (share-nothing), keeps UI at 60/120fps.
- − No shared memory (copy overhead), spawn cost, more complex messaging, plugin restrictions in background isolates.

Interview Questions

1. 🟢 **What is an isolate?** — An independent Dart execution unit with its own memory heap and event loop; isolates share no mutable memory and communicate by message passing.
2. 🟢 **Isolate vs Thread?** — Threads share memory (need locks); isolates share nothing and pass copied messages, eliminating data races.
3. 🟡 **Why doesn't `async` give parallelism?** — `async` schedules work on the *same* single-threaded loop; only isolates run on other threads.
4. 🟡 **How do you offload heavy work in Flutter?** — `compute(fn, arg)` or `Isolate.run(() => ...)`; both spawn a worker isolate.
5. 🟡 **How do isolates communicate?** — Via `SendPort` / `ReceivePort`; messages are deep-copied into the receiver's heap.
6. 🔴 **What crosses the isolate boundary cheaply?** — `TransferableTypedData` (zero-copy ownership transfer of byte buffers); most objects are deep-copied.
7. 🔴 **Can a background isolate call plugins?** — Only with `RootIsolateToken` + `BackgroundIsolateBinaryMessenger.ensureInitialized`; otherwise platform channels aren't wired up.

Senior Engineer Tips

- Default to `Isolate.run` for one-shot tasks; only build long-lived worker isolates with ports when you have sustained/streaming workloads.
- Profile before offloading — the copy of a huge object back to main can itself jank; consider returning summarized/transferable results.
- Structure code so the offloaded function is a **pure, top-level function** over plain data — easiest to send and test.

Architect Perspective

An explicit "compute offload" boundary is a scaling decision: define which operations are CPU-bound and route them to isolates behind a clean API, so feature code stays synchronous-looking and the UI isolate stays free. This is essential for media apps, sync engines, and anything crunching data locally (Modules 19, 33).

Summary

- Isolates = share-nothing parallelism with private heap + loop, communicating via copied messages.
- `Isolate.run` / `compute` offload CPU-bound work to keep the UI responsive.
- Mind copy/spawn costs; send plain data; use transferable buffers for big binaries.

Revision Notes

- Isolate = own heap + own loop; no shared memory; message passing (deep copy).
- `Isolate.run` / `compute` for CPU work; `async` ≠ parallel.
- Ports: `SendPort` / `ReceivePort`; big bytes → `TransferableTypedData`.
- Background isolate + plugins → `RootIsolateToken`.

Practice Questions

1. Why does moving a JSON parse to `compute` stop the UI from freezing?
2. When is spawning an isolate *not* worth it?
3. What data can and cannot be sent across isolates?

Coding Questions

1. Use `Isolate.run` to compute the nth prime without blocking a concurrent periodic timer.
2. Build a long-lived worker isolate that squares numbers on request via ports; kill it cleanly.
3. Compare wall-clock time of parsing a large synthetic JSON on the main isolate vs `Isolate.run`.

Mini Project

Parallel batch processor (pure Dart): Given a list of heavy tasks, process them across a small pool of worker isolates with bounded concurrency, stream progress back to main, and aggregate results. Measure speedup vs sequential. Acceptance: workers reused (not respawned per task); ports closed and isolates killed on completion; measured speedup reported.

Generics (Classes, Methods, Bounds, Variance)

> Generics let you write one type-safe implementation that works for many types — reuse without sacrificing the compiler's guarantees.

Introduction

Generics parameterize code by type: `List<T>`, `Map<K,V>`, `Future<T>`. This file covers generic classes and methods, type parameters, **bounds** (`T extends X`), Dart's **covariance** rules, and advanced patterns (generic constructors, `F`-bounded generics). You met a generic `groupBy<T,K>` in Module 01; here we go deep.

Why this concept exists

Without generics you either duplicate code per type (a `IntList`, a `StringList`) or drop to `dynamic` and lose safety. Generics give you **both** reuse and static type-checking — the compiler tracks `T` so a `List<int>` can't accidentally hold a `String`.

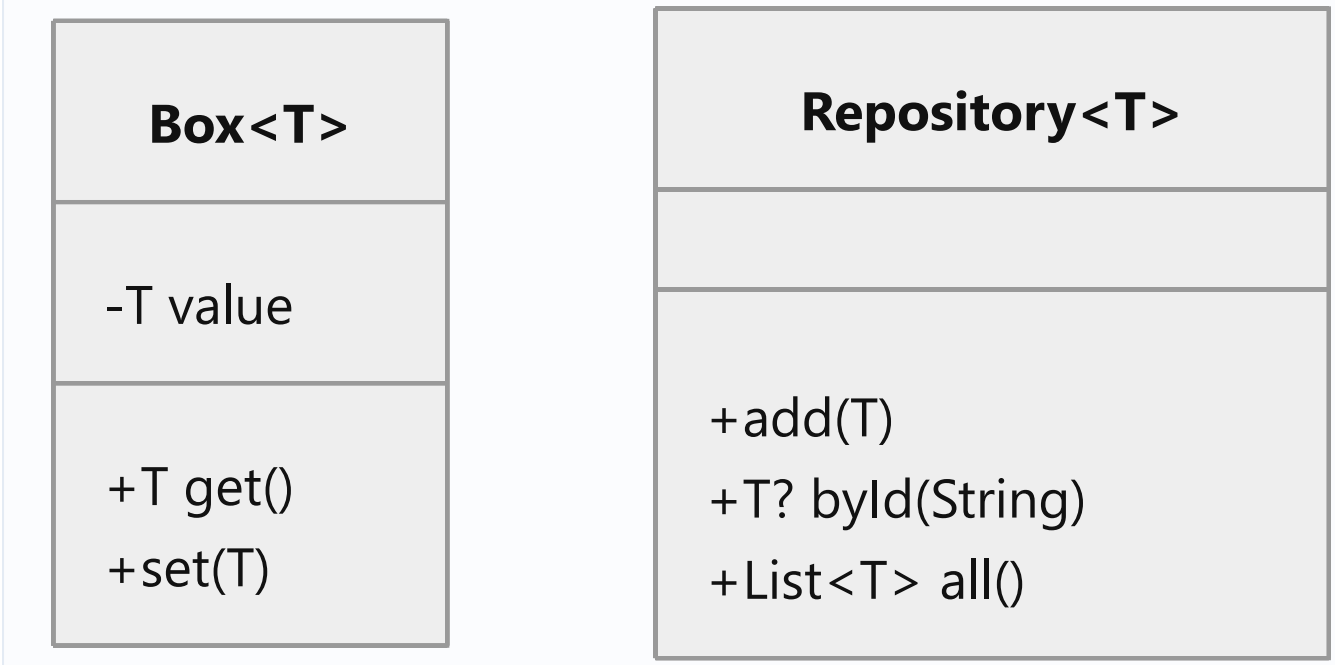
Real-world analogy

A generic class is a **vending machine blueprint** with a slot for "what it dispenses." Build one configured for `Soda`, another for `Snack`. Same machine mechanism (`add`, `dispense`), different, type-checked contents. Bounds (`T extends Drink`) mean "this machine only accepts drinks."

Problem Statement

Build a type-safe `Box<T>`, a `max<T extends Comparable>` function, and a generic `Repository<T>` — and understand why `List<int>` is assignable to `List<num>` but that can bite you. You'll use bounds and reason about variance.

Internal Working



- A **type parameter** (`<T>`) is a placeholder filled per use; it must be *declared* before use (`class Box<T>` , `T fn<T>(...)`).
- **Bounds** constrain it: `T extends Comparable<T>` means only comparable types, and `T` gains `Comparable` 's API.
- **Generic methods** declare their own parameters: `R map<R>(R Function(T) f)` .
- **Variance:** Dart generics are **covariant** — `List<Cat>` is a subtype of `List<Animal>` . Convenient, but *unsound for writes*, so the runtime inserts checks.

Memory Representation

- Generics are **reified** in Dart: type arguments exist at runtime (`list is List<int>` is meaningful), unlike Java's erasure. This costs a little metadata but enables real runtime type checks.

Compiler Behavior

- Type inference fills `T` from arguments/return context (`Box(5)` → `Box<int>`).
- Using `T` outside a declaring scope is a compile error (see Module 01's generics gotcha).
- Covariant assignment `List<Cat> → List<Animal>` compiles; an illegal write is caught at **runtime** with a `TypeError` .

Runtime Behavior

- Reified types let `is` / `as` inspect type arguments: `x is List<String>`.
- Covariance check: `(cats as List<Animal>).add(Dog())` throws at runtime because the backing list is really `List<Cat>`.

Flutter Engine Behavior

Not applicable. (But generics power `State<T>`, `ValueNotifier<T>`, `Future` / `Stream<T>` throughout the framework.)

Dart VM Behavior

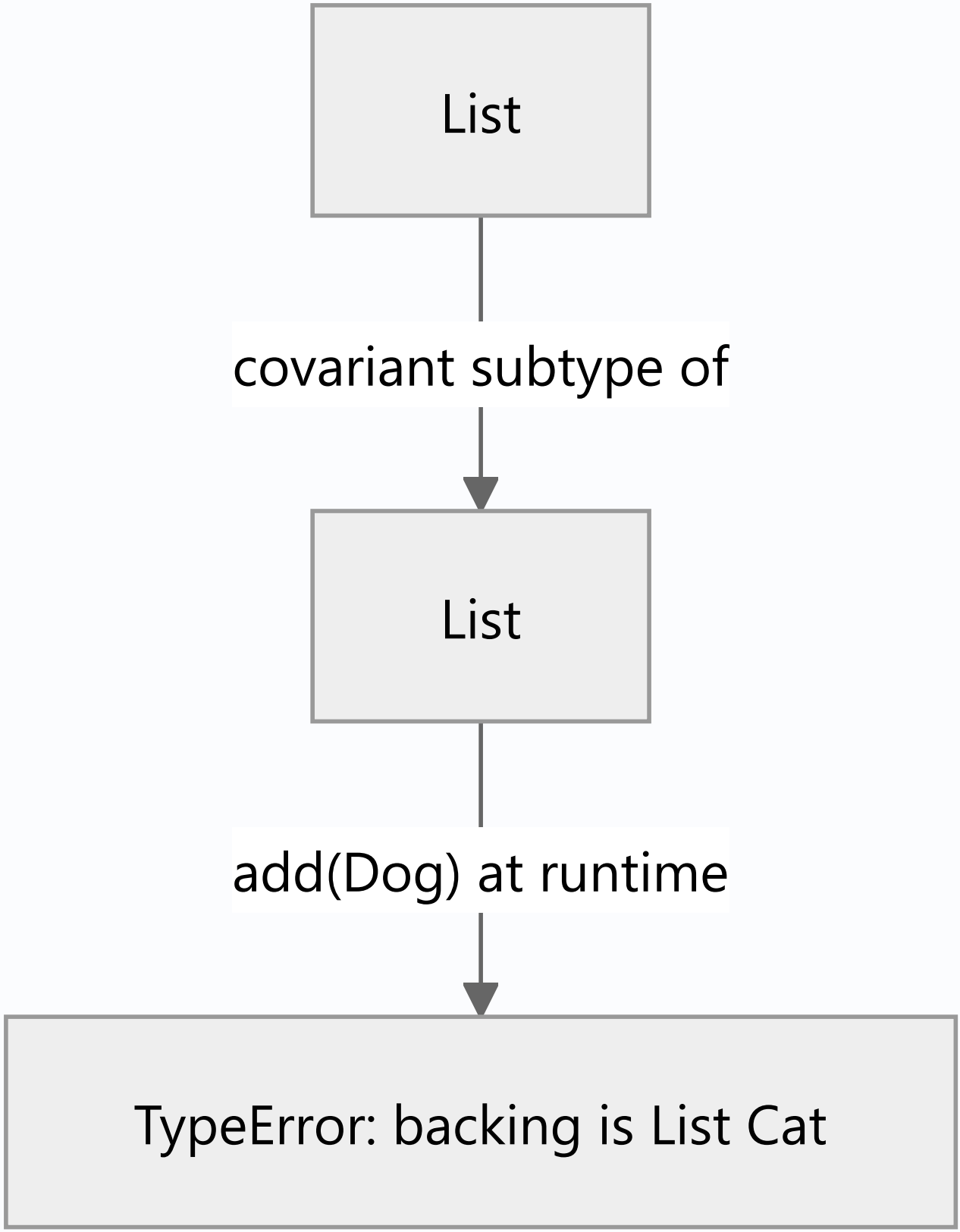
- Reified generics are represented via type argument vectors; AOT specializes/monomorphizes hot generic code paths where possible.

Examples

```
class Box<T> {
  T value;
  Box(this.value);
  R mapTo<R>(R Function(T) f) => f(value); // generic method
}

// bound: T must be Comparable to itself
T maxOf<T extends Comparable<T>>(T a, T b) => a.compareTo(b) >= 0 ? a : b;

abstract class Entity {
  String get id;
}

class Repository<T extends Entity> {
  final _store = <String, T>{};
  void add(T item) => _store[item.id] = item;
  T? byId(String id) => _store[id];
  List<T> all() => _store.values.toList(growable: false);
}

class User extends Entity {
  @override
  final String id;
```

```
  final String name;
  User(this.id, this.name);
}

void main() {
  final b = Box<int>(5);
  print(b.mapTo((v) => 'val=$v')); // val=5

  print(maxOf(3, 9));        // 9
  print(maxOf('a', 'z'));    // z
  // maxOf(User('1','A'), ...) // COMPILE ERROR: User isn't Comparable

  final repo = Repository<User>();
  repo.add(User('u1', 'Ada'));
  print(repo.byId('u1')?.name); // Ada

  // reified generics:
  final ints = <int>[1, 2, 3];
  print(ints is List<int>);  // true (type arg known at runtime)

  // covariance footgun:
  List<num> nums = <int>[1, 2]; // OK (covariant)
  // nums.add(1.5); // runtime TypeError: backing list is List<int>
}
```

Diagrams


List
covariant subtype of
List
add(Dog) at runtime
TypeError: backing is List Cat

Common Mistakes

Mistake	Why	Fix
Using `T` without declaring `<T>`	Compile error	Declare on class/method
Relying on covariant writes	Runtime `TypeError`	Don't widen then write; keep precise types
Overusing `dynamic` instead of a bound	Loses safety	Add `T extends X`
Assuming erasure (Java habits)	Dart reifies types	Use `is List<T>` freely, but know it costs metadata
Unbounded `T` then calling members	`T` only has `Object?` API	Add a bound to unlock methods

Best Practices

- Add the **narrowest useful bound** so `T` exposes the API you need and callers get precise errors.
- Let inference work; annotate explicitly only when it improves clarity or inference fails.
- Prefer generic methods over `dynamic` for transform utilities.
- Keep variance in mind when exposing mutable generic collections — expose read-only views.

Performance

- Reified generics add minor metadata cost; negligible for app code. Monomorphic generic call sites are optimized in AOT.
- Excessive runtime `is`-checks on hot paths can add up — cache/refactor if profiling shows it.

Advantages / Disadvantages

- **+** Reuse + type safety; reified types enable real runtime checks; expressive bounds.
- **−** Covariance is unsound for writes (runtime checks); complex bounds can hurt readability.

Interview Questions

1. 🟢 **Why use generics over `dynamic`?** — Generics keep static type safety and inference; `dynamic` discards checking (runtime crashes) and autocomplete.
2. 🟢 **What is a bound?** — A constraint `T extends X` limiting acceptable types and granting `T` access to `X`'s members.
3. 🟡 **Are Dart generics reified or erased?** — Reified: type arguments exist at runtime, so `x is List<int>` is meaningful (unlike Java's erasure).
4. 🟡 **Explain covariance in Dart generics.** — `List<Cat>` is a subtype of `List<Animal>`. It's convenient but unsound for writes, so the runtime inserts type checks that can throw.

5. 🟡 **What's a generic method vs generic class?** — A method declares its own type parameter (`R map<R>(...)`); a class parameterizes the whole type (`Box<T>`).
6. 🔴 **Why can `(cats as List<Animal>).add(Dog())` throw at runtime?** — The actual backing list is `List<Cat>` ; the covariant view allows the call to compile, but the runtime check rejects inserting a non- `Cat` .
7. 🔴 **What is an F-bounded type parameter?** — `T extends Comparable<T>` — the bound references `T` itself, enabling self-typed APIs like comparison.

Senior Engineer Tips

- Design generic APIs to **accept broadly, return precisely**; expose read-only generic views to sidestep covariance pitfalls.
- Use bounds to make illegal states uncompilable (`Repository<T extends Entity>` guarantees an `id`).
- When inference produces `dynamic` unexpectedly, add an explicit type argument to restore safety.

Architect Perspective

Generics are how you build reusable infrastructure (repositories, result types, caches, DI containers) once and apply them across the domain with full type safety. A well-designed generic core (e.g., `Result<T,E>` , `UseCase<In,Out>`) massively reduces boilerplate and error surface across a large codebase (Modules 38, 40).

Summary

- Generics = reuse + type safety via type parameters, with bounds to constrain and enrich `T` .
- Dart reifies generics (runtime type args) and makes them covariant (unsound for writes, runtime-checked).
- Add narrow bounds; expose read-only views; let inference do the rest.

Revision Notes

- Declare `<T>` before use; bound `T extends X` for API + constraint.
- Reified (runtime type args) — `is List<int>` works.
- Covariant: `List<Cat>` <: `List<Animal>` ; illegal write → runtime `TypeError` .
- Generic method: `R map<R>(...)` . F-bound: `T extends Comparable<T>` .

Practice Questions

1. Why does `maxOf` require `T extends Comparable<T>`?
2. Give a concrete covariance example that compiles but throws at runtime.
3. How does reification change what `is` can check compared to Java?

Coding Questions

1. Implement `Result<T, E>` (a sealed generic with `Ok(T)` / `Err(E)`) + `map` / `mapErr`.
2. Write `Iterable<T> distinctBy<T, K>(Iterable<T> xs, K Function(T) key)`.
3. Build a generic `Cache<K, V>` with capacity + LRU eviction.

Mini Project

Generic repository + result core (pure Dart): Build `Entity`, `Repository<T extends Entity>`, and `Result<T, Failure>`; implement CRUD returning `Result`s, with a generic `find<T>` transform. Add tests across two entity types. Acceptance: no `dynamic`; bounds enforce `id`; covariance not abused; `dart analyze` clean.

Mixins (`mixin`, `on`, Linearization)

> A mixin is reusable behavior you "mix into" a class without inheritance — Dart's answer to sharing code across unrelated class hierarchies while avoiding the diamond problem.

Introduction

A mixin packages methods/fields that can be applied to many classes via `with`. Unlike inheritance (one superclass) you can mix in **several**. This file covers declaring mixins, the `on` constraint, how conflicts resolve via **linearization**, and when to choose a mixin over inheritance or composition.

Why this concept exists

Dart has **single inheritance**: a class has one superclass. But behavior like "can be serialized," "is loggable," "is a `ChangeNotifier`" cuts across unrelated types. Copy-pasting is bad; forcing a shared base class is worse. Mixins let you compose orthogonal behaviors cleanly — the core mechanism behind Flutter's `SingleTickerProviderStateMixin`, `WidgetsBindingObserver`, etc.

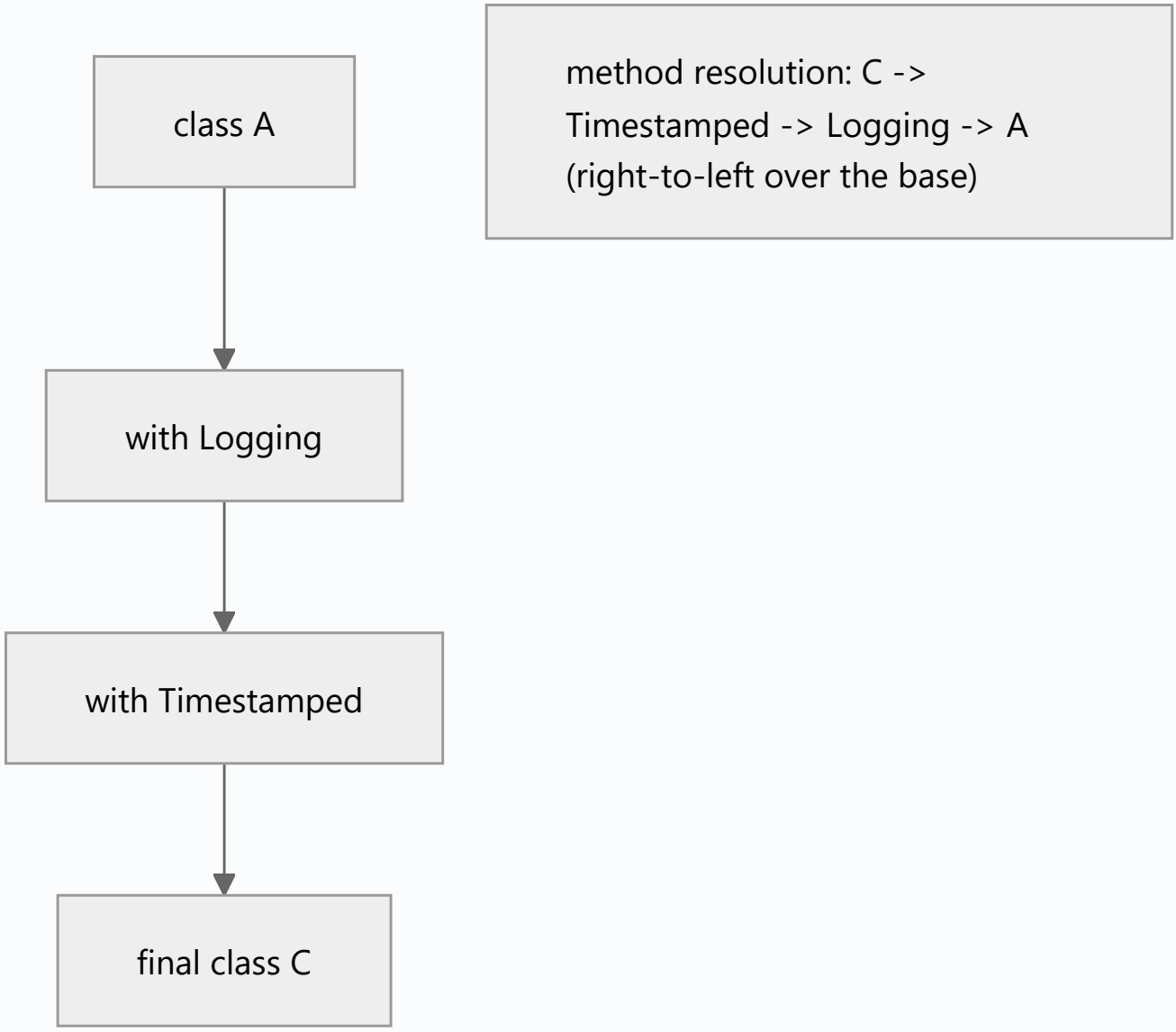
Real-world analogy

Mixins are **superpowers you grant a character**. A base `Hero` can gain `Flying`, `Healing`, and `Invisibility` independently. Each power is defined once and mixed into many characters. `on Hero` means "this power only works on Heroes" (it relies on Hero abilities).

Problem Statement

Add JSON-logging and timestamping behavior to several unrelated classes, and add ticker behavior that requires a `State`. You'll write mixins, use `on` to require a base, and resolve a method-name conflict via ordering.

Internal Working



- Declare with `mixin Logging { ... }`; apply with `class C extends A with Logging, Timestamped`.
- `mixin M on Base` constrains: `M` can only be mixed into classes that are (or extend) `Base`, and may call `Base`'s members and `super`.
- **Linearization:** Dart flattens the mixin application into a linear chain. Later mixins (rightmost in `with`) override earlier ones. `super` inside a mixin refers to the *next* type down the linearized chain — enabling stackable behavior.
- A mixin can't have a (generative) constructor; `mixin class` (Dart 3) can be used both as a mixin and a normal class.

Memory Representation

- No separate runtime object: mixin members become part of the composed class's method table via the synthesized linearized classes. Instances are ordinary objects.

Compiler Behavior

- Applying a mixin whose `on` constraint isn't satisfied is a compile error.
- Name conflicts resolve by linearization order; ambiguous cases you don't override may require an explicit override.
- `mixin` (pure) can't be instantiated or extended; `mixin class` can.

Runtime Behavior

- Method dispatch follows the linearized order; `super.method()` in a mixin walks to the next member in the chain (used for "wrap the parent" patterns).

Flutter Engine Behavior

Not applicable. But Flutter relies heavily on mixins: `SingleTickerProviderStateMixin` (animations), `WidgetsBindingObserver` (lifecycle), `AutomaticKeepAliveClientMixin` (list item retention).

Dart VM Behavior

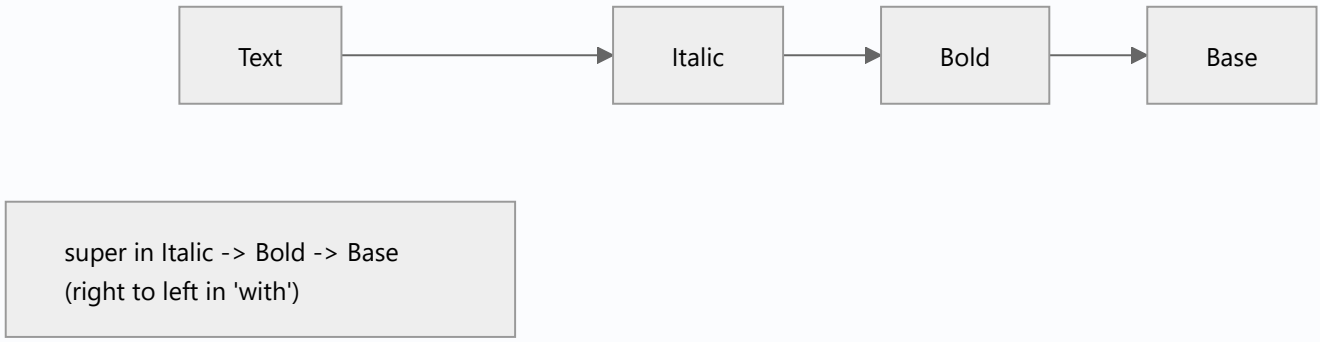
- Linearized mixin application produces synthetic classes the VM treats like normal classes for dispatch; no special runtime cost beyond normal virtual calls.

Examples

```
mixin Logging {
  void log(String msg) => print('[${runtimeType}] $msg');
}

mixin Timestamped {
  DateTime get now => DateTime.now();
  String stamp(String msg) => '$now: $msg';
}

// `on` constraint: requires a base that has `describe`
abstract class Describable {
  String describe();
}

mixin PrettyPrint on Describable {
  void printPretty() => print('>> ${describe()}'); // can call base member
}
```

```
class Service with Logging, Timestamped {
  void run() {
    log(stamp('started')); // uses both mixins
  }
}

class Report extends Describable with PrettyPrint {
  @override
  String describe() => 'Q3 report';
}

// stackable super in mixins (linearization):
class Base {
  String render() => 'base';
}
mixin Bold on Base {
  @override
  String render() => '<b>${super.render()}</b>';
}
mixin Italic on Base {
  @override
  String render() => '<i>${super.render()}</i>';
}

class Text extends Base with Bold, Italic {} // Italic is applied last

void main() {
  Service().run(); // [Service] <timestamp>: started

  Report().printPretty(); // >> Q3 report

  // linearization: Text -> Italic -> Bold -> Base
  print(Text().render()); // <i><b>base</b></i>
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Expecting left-to-right override	Rightmost mixin wins (applied last)	Order `with` deliberately
Adding a constructor to a mixin	Pure mixins can't have generative ctors	Use `mixin class` or move state to the host
Using inheritance for cross-cutting behavior	Forces artificial hierarchies	Use a mixin
Ignoring `on` needs	Mixin can't access required base members	Add `on Base`
Mixin with mutable state shared confusion	State lives per-instance of host	Keep mixins behavior-focused

Best Practices

- Use mixins for **orthogonal, reusable behavior**; keep them small and focused (SRP).
- Prefer `on` to declare dependencies explicitly rather than assuming host members.
- Order `with` clauses intentionally when behaviors stack via `super`.
- Prefer **composition** when the behavior has its own lifecycle/state or you need runtime swapping.

Performance

- No meaningful runtime overhead beyond normal virtual dispatch.

Advantages / Disadvantages

- + Reuse across unrelated hierarchies; multiple mixins; stackable via `super`; no diamond ambiguity.

- – Order-sensitive; can obscure where a method comes from; not a substitute for composition when state/lifecycle is involved.

Interview Questions

1. 🟢 **What is a mixin and why use it?** — Reusable behavior mixed into classes via `with`, enabling code sharing across unrelated hierarchies without multiple inheritance.
2. 🟢 **`extends` vs `with` vs `implements`?** — `extends`: inherit one superclass's implementation. `with`: mix in reusable implementation(s). `implements`: adopt only a contract (reimplement all).
3. 🟡 **What does `on` do in a mixin?** — Constrains which classes can use the mixin and lets it call the required base's members/`super`.
4. 🟡 **How are method conflicts resolved?** — By linearization: the class flattens into a chain; the rightmost mixin in `with` wins; `super` walks the chain.
5. 🟡 **Can a mixin have a constructor?** — A pure `mixin` can't have a generative constructor; a `mixin class` can act as both a class and a mixin.
6. 🔴 **How does Dart avoid the diamond problem?** — Linearization imposes a single, unambiguous order of superclasses/mixins, so there's always exactly one method to call.
7. 🔴 **When choose composition over a mixin?** — When the behavior has independent state/lifecycle, needs runtime swapping, or you want to avoid tight coupling to the host's type.

Senior Engineer Tips

- Read `with A, B, C` as "apply A, then B, then C over the base" — that's your `super` chain and override order.
- Use `on` to make mixin dependencies explicit and self-documenting; it turns "assumes host has X" into a compile-time guarantee.
- Flutter's `...Mixin` classes are your models: study `SingleTickerProviderStateMixin` to see `on State` + lifecycle hooks in action.

Architect Perspective

Mixins are a composition tool for cross-cutting concerns (logging, caching hooks, lifecycle observation). Used judiciously they reduce duplication; overused they create "where did this method come from?" confusion. Establish conventions: small, single-purpose, `on`-constrained mixins, and prefer explicit composition for stateful collaborators (Module 03 OOP).

Summary

- Mixins add reusable behavior via `with`, across unrelated hierarchies, without multiple inheritance.
- `on` constrains the host; linearization resolves conflicts (rightmost wins; `super` stacks).
- Prefer small focused mixins; use composition for stateful/lifecycle behavior.

Revision Notes

- `with M1, M2` = apply left→right; rightmost overrides; `super` walks the linearized chain.
- `on Base` = required host type; enables `super`/base calls.
- Pure `mixin` has no generative ctor; `mixin class` doubles as a class.
- extends=inherit code, with=mix behavior, implements=contract only.

Practice Questions

1. Predict `Text().render()` and explain via linearization.
2. When is `on` necessary, and what does it buy you?
3. Mixin vs composition — give a case for each.

Coding Questions

1. Write a `Serializable` mixin (`toJson`) and mix it into two unrelated classes.
2. Create stackable `Encrypted` and `Compressed` mixins over a `Pipeline` base using `super`.
3. Implement a `Disposable` mixin tracking and closing resources.

Mini Project

Text decoration engine (pure Dart): Build a `Base` renderer plus stackable mixins (`Bold`, `Italic`, `Underline`) using `super`, and demonstrate how `with` order changes the output nesting. Add an `on`-constrained `Debuggable` mixin. Acceptance: linearization behavior documented with expected outputs; mixins are single-purpose; `dart analyze` clean.

Extension Methods

> Extensions let you add methods, getters, and operators to *existing* types you don't own — without subclassing or modifying the original.

Introduction

An `extension` adds functionality to a type (`String` , `int` , `BuildContext` , a third-party class) as if the members were part of it. This file covers declaring extensions, how they resolve statically, named/unnamed extensions, generic extensions, and their limits.

Why this concept exists

You often want a helper "on" a type — `'  hi '.cleaned` , `context.theme` , `42.seconds` — but can't edit `String` / `Duration` /the SDK. Before extensions you wrote `StringUtils.cleaned(s)` , which reads backwards and doesn't autocomplete on the value. Extensions give fluent, discoverable, type-safe helpers without inheritance.

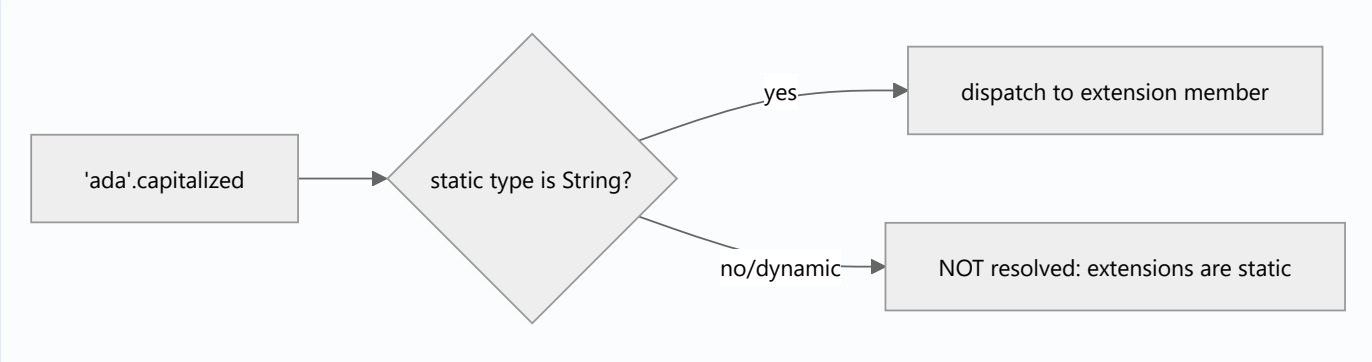
Real-world analogy

An extension is a **clip-on attachment** for a tool you already own — a bottle-opener that snaps onto your keychain. The keychain (the type) is unchanged; you've added a capability you can use as if it were built in.

Problem Statement

Add `String.capitalized` , `int.isPrime` , `List<T>.secondOrNull` , and a Flutter-style `BuildContext.theme` . You'll write extensions, including a generic one, and understand static resolution.

Internal Working



- Syntax: `extension Name on Type { members }`. `Name` can be omitted (but named extensions can be hidden/shown on import to resolve conflicts).
- Members can be methods, getters/setters, operators — but **not instance fields** (no per-object state) and not new constructors.
- **Static resolution:** the extension chosen is based on the **static type** at the call site. Extensions do *not* work through `dynamic`.
- Generic extensions: `extension ListX<T> on List<T> { ... }`.

Memory Representation

- Extensions add **no state** to instances (fields aren't allowed). They compile to static calls passing the receiver — zero per-object memory cost.

Compiler Behavior

- Extension dispatch is resolved at **compile time** by static type; a real instance method with the same name always wins over an extension.
- Import conflicts (two extensions with the same member on the same type) are resolved with `show` / `hide` or explicit application `ExtensionName(value).member`.

Runtime Behavior

- Because resolution is static, calling an extension member on a `dynamic` receiver throws `NoSuchMethodError` at runtime (it wasn't resolved to the extension).

Flutter Engine Behavior

Not applicable. (But `context.theme` / `context.textTheme` style extensions are ubiquitous in Flutter codebases and packages.)

Dart VM Behavior

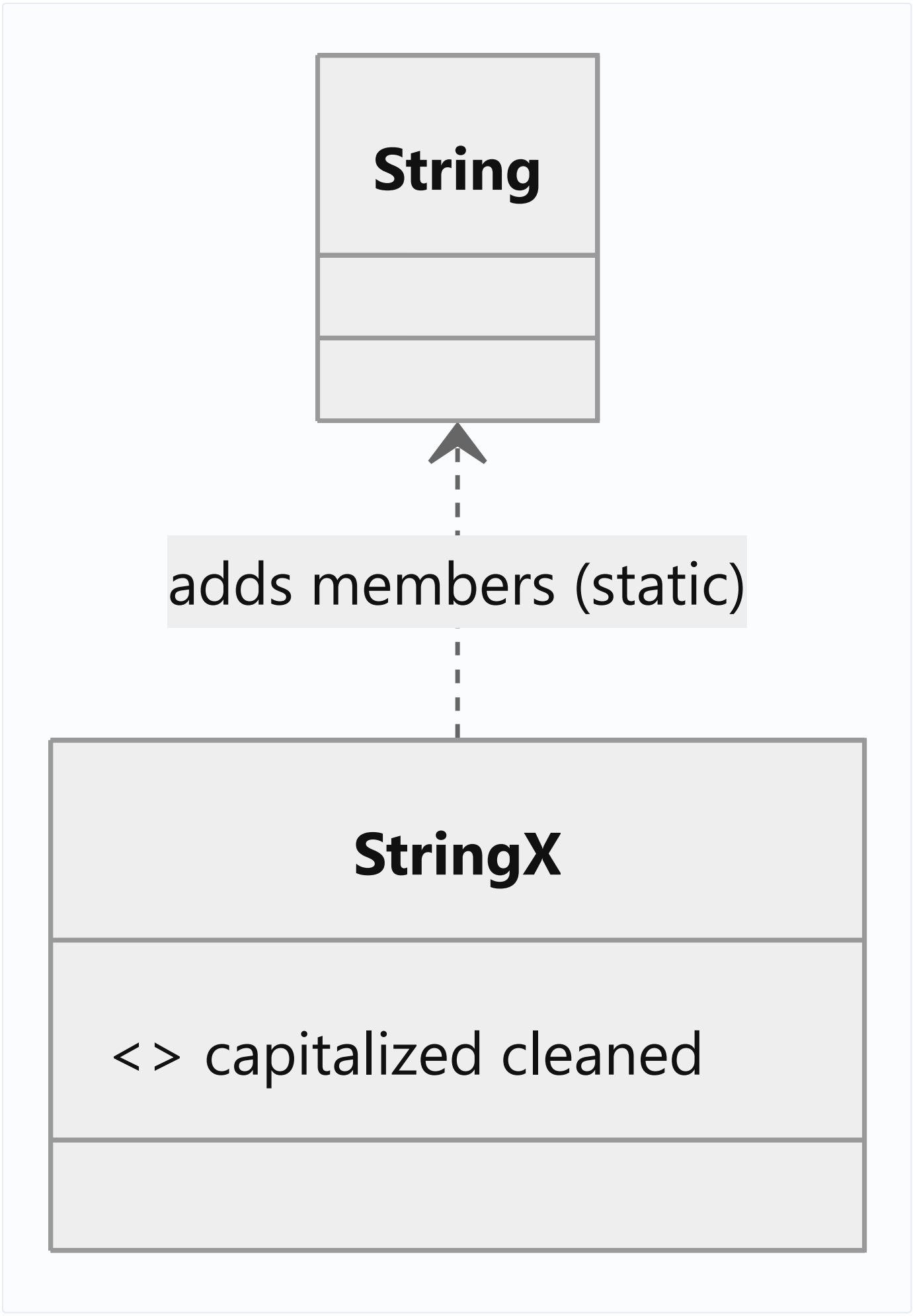
- Compiled to ordinary static method calls; no dynamic lookup or overhead beyond a normal function call.

Examples

```
extension StringX on String {
  String get capitalized =>
      isEmpty ? this : '${this[0].toUpperCase()}${substring(1)}';
```

```
  String get cleaned => trim().replaceAll(RegExp(r'\s+'), ' ');
}

extension IntX on int {
  bool get isPrime {
    if (this < 2) return false;
    for (var i = 2; i * i <= this; i++) {
      if (this % i == 0) return false;
    }
    return true;
  }

  Duration get seconds => Duration(seconds: this);
}

// generic extension
extension ListX<T> on List<T> {
  T? get secondOrNull => length >= 2 ? this[1] : null;
}

void main() {
  print('ada'.capitalized);        // Ada
  print('  a   b '.cleaned);       // a b
  print(7.isPrime);                // true
  print(30.seconds);               // 0:00:30.000000
  print(<int>[1, 2, 3].secondOrNull); // 2

  // static resolution gotcha:
  dynamic d = 'ada';
  // print(d.capitalized); // runtime NoSuchMethodError — dynamic bypasses extensions
}
```

Diagrams


String
adds members (static)
StringX
<> capitalized cleaned

Common Mistakes

Mistake	Why	Fix
Calling an extension on `dynamic`	Not resolved statically	Use a typed variable
Trying to add instance fields	Extensions are stateless	Use a wrapper class / extension type
Name clashes across packages	Ambiguous member	`show` / `hide` on import, or `Ext(x).m()`
Shadowing a real method unintentionally	Instance method wins silently	Name extension members distinctly
Overstuffing "God" extensions	Hard to discover/maintain	Group cohesive helpers per extension

Best Practices

- Name extensions (helps conflict resolution and readability).
- Keep them **cohesive and small**; group by purpose (`StringCaseX` , `ContextThemeX`).
- Use for fluent, discoverable helpers; don't hide heavy logic behind a trivial-looking getter.
- Prefer extensions over `XyzUtils` static-helper classes for value-oriented helpers.

Performance

- Zero runtime overhead beyond a static call; safe in hot paths.

Advantages / Disadvantages

- **+** Fluent, discoverable, type-safe helpers on any type; no subclassing; no runtime cost.
- **−** No state/fields; static-only (no `dynamic`); potential import conflicts; can be overused.

Interview Questions

1. 🟢 **What are extension methods for?** — Adding methods/getters/operators to existing types you don't own, without subclassing or editing them.
2. 🟢 **Can extensions hold state?** — No; they can't declare instance fields (use a wrapper class or extension type).
3. 🟡 **How are extension members resolved?** — Statically, by the receiver's static type at compile time; a real instance method with the same name wins.
4. 🟡 **Why doesn't an extension work on `dynamic` ?** — Resolution is static; a `dynamic` receiver has no static type to dispatch on, so it fails at runtime.
5. 🟡 **How do you resolve two extensions with the same member?** — `show` / `hide` on import, or apply explicitly: `MyExt(value).member` .
6. 🔴 **Extension vs inheritance vs wrapper?** — Extension: stateless helpers on existing types. Inheritance: is-a with shared implementation. Wrapper/extension type: add state or a distinct

type identity.

7. 🔴 **Do extensions have runtime cost?** — No; they compile to static calls.

Senior Engineer Tips

- Extensions shine for domain-specific fluency: `context.theme`, `ref.watch(...)`-style helpers, `DateTime.isSameDay(...)`.
- Keep them in a well-known `extensions/` location so the team discovers and reuses them.
- If you find yourself wanting a field, you actually want an **extension type** or a wrapper class, not an extension.

Architect Perspective

Extensions let you build an ergonomic, consistent "vocabulary" over SDK/third-party types without wrapping everything. Standardize a small set of project extensions (theme access, spacing, formatting) to reduce boilerplate and enforce consistency across teams — a subtle but real maintainability win.

Summary

- `extension X on T` adds stateless methods/getters/operators to `T`.
- Resolved statically by the receiver's static type; instance methods win; no `dynamic`.
- Great for fluent helpers with zero runtime cost; use wrappers/extension types when you need state.

Revision Notes

- `extension Name on Type {}`; methods/getters/operators, **no fields**.
- Static resolution by static type; real method wins; fails on `dynamic`.
- Conflicts: `show` / `hide` or `Ext(x).m()`.
- Zero runtime cost (static calls).

Practice Questions

1. Why does calling an extension getter on a `dynamic` variable throw?
2. When would you choose a wrapper class over an extension?
3. How do you disambiguate two same-named extension members from different packages?

Coding Questions

1. Write `NumX` with `clampMin`, `clampMax`, and `percentOf`.
2. Add `DateTimeX` with `isToday`, `startOfDay`, and `isSameDay(other)`.
3. Create a generic `IterableX<T>` with `firstWhereOrNull(test)` and `partition(test)`.

Mini Project

Fluent formatting toolkit (pure Dart): Build cohesive extensions for `String` (case/trim/mask), `num` (currency/percent), and `DateTime` (relative/format), each named and grouped. Add tests. Acceptance: no state in extensions; named extensions; conflicts avoidable via naming; `dart analyze` clean.

Extension Types (Zero-Cost Wrappers)

> An extension type is a compile-time-only wrapper that gives an existing type a new, distinct static identity and API — with **no runtime allocation or overhead**.

Introduction

Extension types (stable since Dart 3.3) let you wrap an existing representation type (e.g., `int`, `String`) in a new named type that the compiler treats as distinct, while at runtime it *is* the underlying value — no wrapper object is created. This file covers what they are, how they differ from extensions and wrapper classes, `implements` for transparency, and their key limitation (no runtime type identity).

Why this concept exists

You often want type-safe distinctions that shouldn't cost performance: a `UserId` that can't be mixed up with an `OrderId` even though both are `int`; a `Meters` that isn't accidentally added to `Seconds`. A wrapper class gives safety but allocates an object per value. Extension types give the **safety of a distinct type with the performance of the raw value** — zero-cost domain typing.

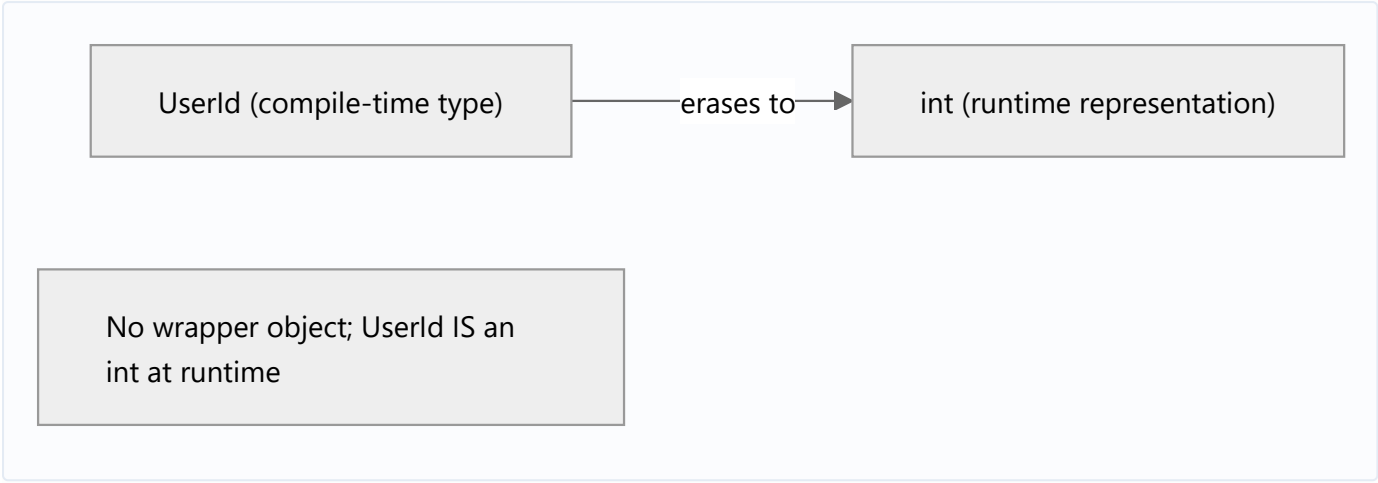
Real-world analogy

An extension type is a **role badge** on a person. The person (the `int`) is unchanged and weighs nothing extra; the badge ("Doctor") changes how the system treats them and what they're allowed to do — but it's purely a label enforced by the rules, not a second physical person following them around.

Problem Statement

You keep passing `int` IDs around and once swapped a `userId` for an `orderId` — a bug the compiler didn't catch. Introduce distinct `UserId` / `OrderId` types with no runtime cost, exposing only safe operations. You'll use extension types.

Internal Working



- Declared: `extension type UserId(int value) { ... }` — `int value` is the **representation**.
- The new type is **distinct at compile time**: you can't pass a raw `int` where `UserId` is expected (unless via the constructor), and `UserId` / `OrderId` aren't interchangeable.
- At **runtime** it's erased to the representation type (`int`) — no allocation.
- You choose the exposed API: only members you declare (plus, if you write `implements int`, the underlying `int` members are visible too — a "transparent" extension type).
- Can be generic: `extension type Wrapped<T>(T value)`.

Memory Representation

- **Zero wrapper allocation.** A `List<UserId>` has the same memory layout as a `List<int>`. This is the headline benefit over wrapper classes.

Compiler Behavior

- Enforces the distinct type statically: `takesUserId(orderId)` is a compile error.
- Without `implements`, the underlying type's members are **hidden** (you expose a curated API). With `implements int`, they're available (transparent).
- Conversions to/from the representation are explicit (via the constructor / `.value`).

Runtime Behavior

- Because it erases to the representation, `is` **/runtime type checks see the representation**, not the extension type. `userId is int` is `true`; there's no distinct runtime `UserId` type to test. This is the main caveat.
- No runtime safety: if you obtain the value via `dynamic` /casts, the distinction is gone.

Flutter Engine Behavior

Not applicable. (Useful for typed IDs/units in domain and data layers; JS interop also uses extension types heavily on web.)

Dart VM Behavior

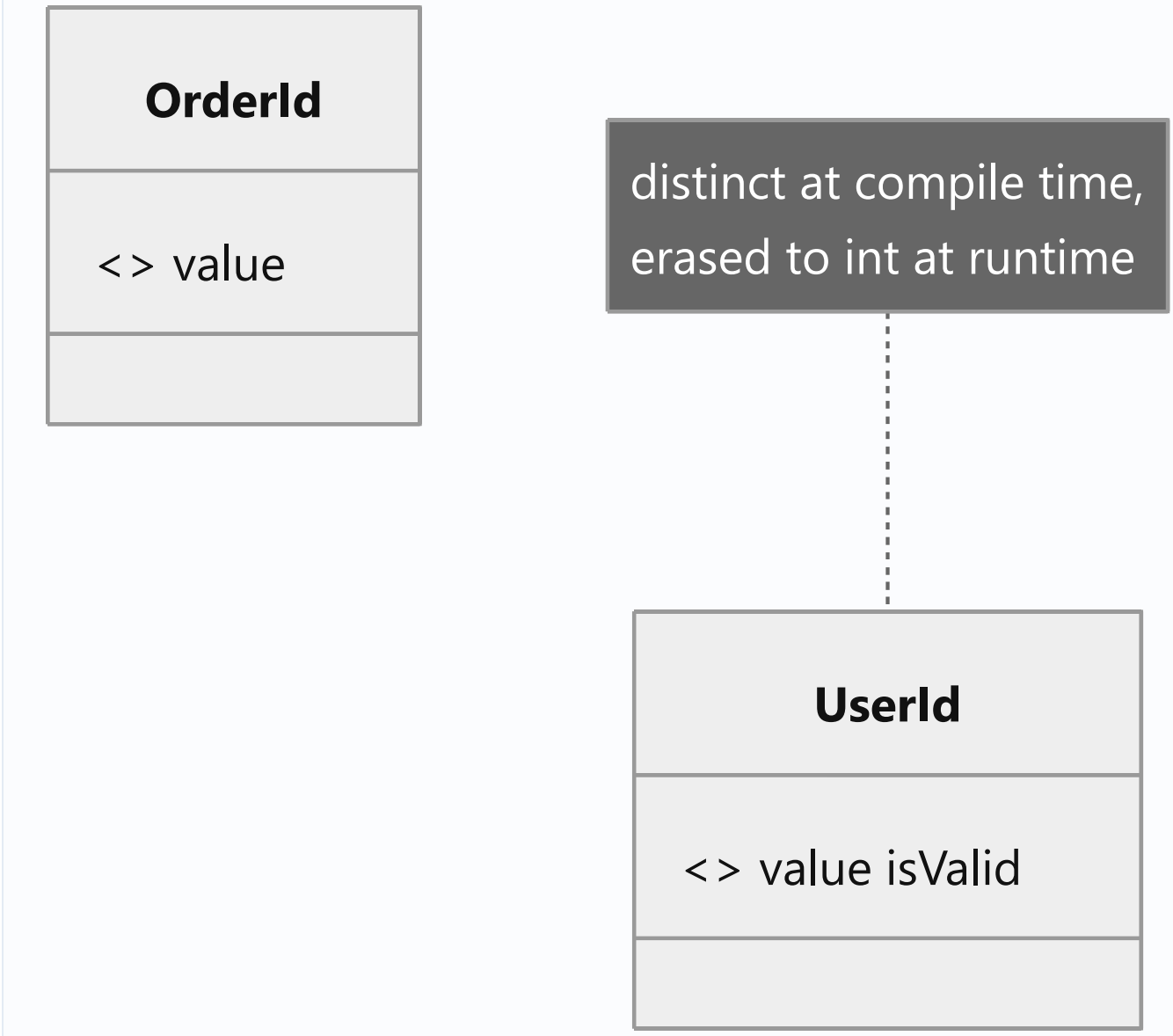
- Fully erased in AOT/JIT; call sites compile to operations on the representation type — identical performance to using the raw value.

Examples

```
extension type UserId(int value) {
  // curated API only:
  bool get isValid => value > 0;
}

extension type OrderId(int value) {}

// transparent: also exposes int's members
extension type Cents(int value) implements int {
  String get formatted => '\$${(value / 100).toStringAsFixed(2)}';
}

void takesUserId(UserId id) => print('user ${id.value}');

void main() {
  final u = UserId(42);
  final o = OrderId(42);

  takesUserId(u);        // ok
  // takesUserId(o);     // COMPILE ERROR: OrderId is not UserId
  // takesUserId(42);    // COMPILE ERROR: int is not UserId

  print(u.isValid);      // true
  print(u.value);        // 42

  // transparent extension type can use int's API directly:
  final price = Cents(1999);
  print(price + 1);      // 2000 (int member available via implements int)
```

```
  print(price.formatted);// $19.99


  // runtime caveat: erases to int
  print(u.value is int); // true
  print(u is int);       // true — no distinct runtime type
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Expecting runtime type safety	Erases to representation	Rely on it only at compile time
Using `is ExtensionType` checks	Not a distinct runtime type	Check the representation, or use a real class
Confusing with extension methods	Extensions add methods to a type; extension types make a *new* type	Pick by whether you need a distinct identity
Overusing `implements`	Leaks the whole underlying API	Only implement when you want transparency
Needing polymorphism/subtyping	Extension types aren't for rich hierarchies	Use classes

Best Practices

- Use for **zero-cost domain typing**: IDs, units, validated primitives, opaque handles.
- Expose a **minimal curated API**; add `implements` only when transparency is genuinely wanted.
- Document that safety is **compile-time only**; don't rely on runtime checks.
- Prefer a real class when you need runtime type identity, inheritance, or added state beyond the representation.

Performance

- Identical to the underlying representation — no allocation, no boxing. Ideal for large collections of typed primitives.

Advantages / Disadvantages

- **+** Distinct type safety at zero runtime cost; curated APIs over primitives; great for interop.
- **−** Compile-time only (no runtime type identity/ `is`); no subtyping/polymorphism; can confuse newcomers.

Interview Questions

1. 🟢 **What is an extension type?** — A compile-time-only wrapper giving an existing representation type a new distinct static type and curated API, erased to the representation at runtime (zero cost).
2. 🟢 **Extension type vs wrapper class?** — Wrapper allocates an object per value; extension type has zero runtime allocation but only compile-time distinctness.

3. 🟡 **Extension type vs extension method?** — Extension methods add members to an existing type (same identity); extension types create a new distinct type over a representation.
4. 🟡 **What does `implements int` do on an extension type?** — Makes it "transparent" — the underlying `int`'s members are accessible through it.
5. 🟡 **What's the main caveat?** — No runtime type identity: `userId is int` is true and there's no distinct `UserId` at runtime; safety is compile-time only.
6. 🔴 **When choose an extension type over a class?** — When you want type-safe primitives/IDs/units in hot or large-collection code paths without allocation cost, and don't need runtime polymorphism.
7. 🔴 **Why are extension types big for web interop?** — They provide typed, zero-cost views over JS values without wrapper allocations.

Senior Engineer Tips

- Reach for extension types to kill "stringly/inty-typed" bugs (mixing `userId` / `orderId`, `email` / `name`) without perf cost.
- Keep the API curated: the whole point is to *prevent* invalid operations, so don't blanket-`implements` the representation.
- Remember tests/JSON/`is`-based dispatch see the representation — design serialization and equality accordingly.

Architect Perspective

Extension types enable **making illegal states unrepresentable** cheaply — a DDD value-object flavor at primitive cost. In large domains, typed IDs and units eliminate a whole class of mix-up bugs at compile time while keeping data structures lean, which matters in data-heavy and performance-sensitive apps (Module 46 DDD).

Summary

- Extension types: distinct compile-time type over a representation, erased to it at runtime — zero cost.
- Curate the API; add `implements` for transparency; safety is compile-time only.
- Prefer classes when you need runtime type identity, state, or polymorphism.

Revision Notes

- `extension type UserId(int value) {}` — distinct at compile time, IS `int` at runtime.
- Zero allocation; `List<UserId>` == `List<int>` in memory.
- `implements int` = transparent (exposes int API).

- Caveat: no runtime type (`is int` true); compile-time safety only.

Practice Questions

1. Why can't the compiler let you pass an `OrderId` to a `UserId` parameter?
2. What does `userId is int` return and why does that matter?
3. When is a wrapper class the better choice?

Coding Questions

1. Define `Email(String value)` extension type with a `isValid` getter and a validating factory-like constructor.
2. Model `Meters` and `Seconds` extension types; make `Meters + Meters` compile but `Meters + Seconds` fail.
3. Create a transparent `Percentage(double value) implements double` with a `clamped` getter.

Mini Project

Typed-primitives domain kit (pure Dart): Introduce `UserId`, `OrderId`, `Email`, and `Money` (Cents) extension types; refactor a small service so IDs can't be swapped and money math is safe — all with zero wrapper allocation. Add tests proving type mix-ups fail to compile (documented) and behavior is correct. Acceptance: minimal curated APIs; documented runtime caveat; `dart analyze` clean.

Constructors, Factories, Singletons & Callable Classes

> Dart's constructor toolkit — generative, named, `const`, factory, private — controls *how* and *whether* an object is created, enabling patterns from immutability to singletons to caching.

Introduction

This file covers Dart's full constructor system: generative (default/named), initializer lists, `const` constructors, **factory** constructors (which may return existing/cached/subtype instances), private constructors (`_`), the **singleton** pattern, and **callable classes** (objects invoked like functions via `call`).

Why this concept exists

Object creation isn't one-size-fits-all. Sometimes you must set `final` fields before the body (initializer lists), enable compile-time constants (`const`), return a cached instance instead of a new one (factory + singleton), forbid external construction (private constructor), or make an object behave like a function (`call`). Dart bakes these into the language rather than requiring boilerplate.

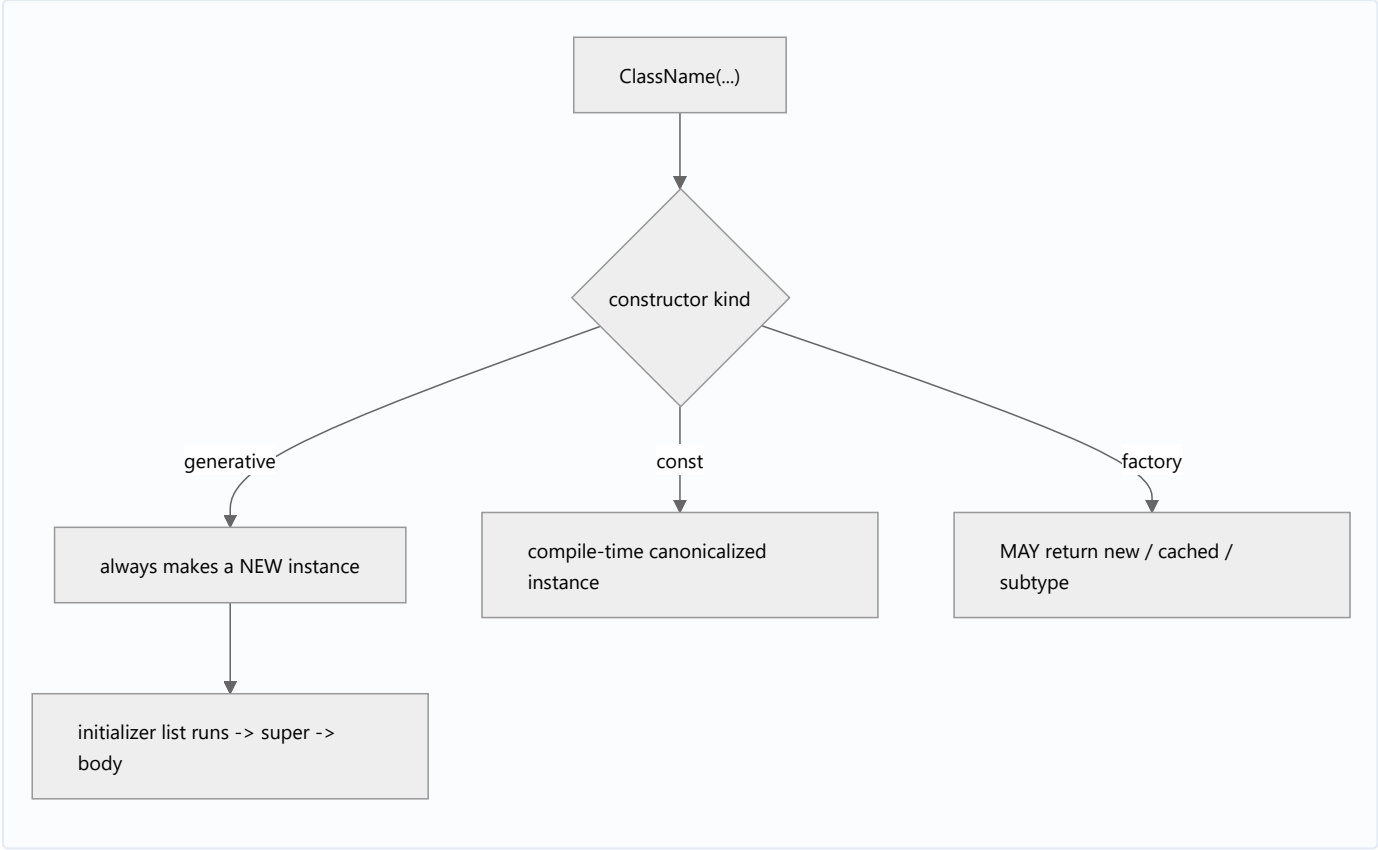
Real-world analogy

- Generative constructor = **building a house from scratch** to spec.
- Factory constructor = **a dealership**: you ask for a car; it might hand you one off the lot (cached), a specific model (subtype), or refuse — you don't control *whether* a new one is built.
- Singleton = **the one and only town hall** — everyone gets the same instance.
- Callable class = **a stamp** — it's an object, but you "use" it by pressing it (`stamp(doc)`).

Problem Statement

Create an immutable `Point` with a `const` constructor, a `Color.fromHex` factory that caches, a `Logger` singleton, and a `Validator` object you can call like a function. You'll use each constructor kind.

Internal Working



- **Generative:** `Point(this.x, this.y)` — always creates a new instance. Shorthand `this.field` assigns fields.
- **Initializer list:** `Point(int x, int y) : _x = x, assert(...) { }` — runs before the body; required for `final` fields, asserts, and `super(...)`. `this` isn't available yet.
- **Named:** `Point.origin() : this(0, 0);` — alternative constructors; can redirect.
- `const` **constructor:** all fields `final`; enables compile-time constant, canonicalized instances.
- **Factory:** `factory Color.fromHex(...)` — has **no initializer list**, must `return` an instance; can return cached objects or subtypes.
- **Private constructor:** `ClassName._()` — blocks external instantiation (basis of singletons/enums-like patterns).
- **Callable class:** define `ReturnType call(args)` → instances are invocable as `obj(args)`.

Memory Representation

- Generative constructors allocate a new heap object each call. Factories may return an existing object (no allocation). `const` instances are shared canonical objects.

Compiler Behavior

- `const` constructor requires all fields `final` and all initializers const-compatible.

- Factory constructors can't access `this` (nothing constructed yet) and can't have initializer lists.
- Private constructor + no public constructor → the compiler prevents external `new`.

Runtime Behavior

- Factory logic runs at call time (e.g., look up a cache map, decide subtype).
- Singleton: the single instance is created lazily (often `static final _instance = Foo._();` or `late final`).
- `obj(args)` dispatches to `call`.

Flutter Engine Behavior

Not applicable. (But `const` widget constructors are central to rebuild performance, and factory constructors appear across the SDK, e.g., `List.from`, `Map.of`.)

Dart VM Behavior

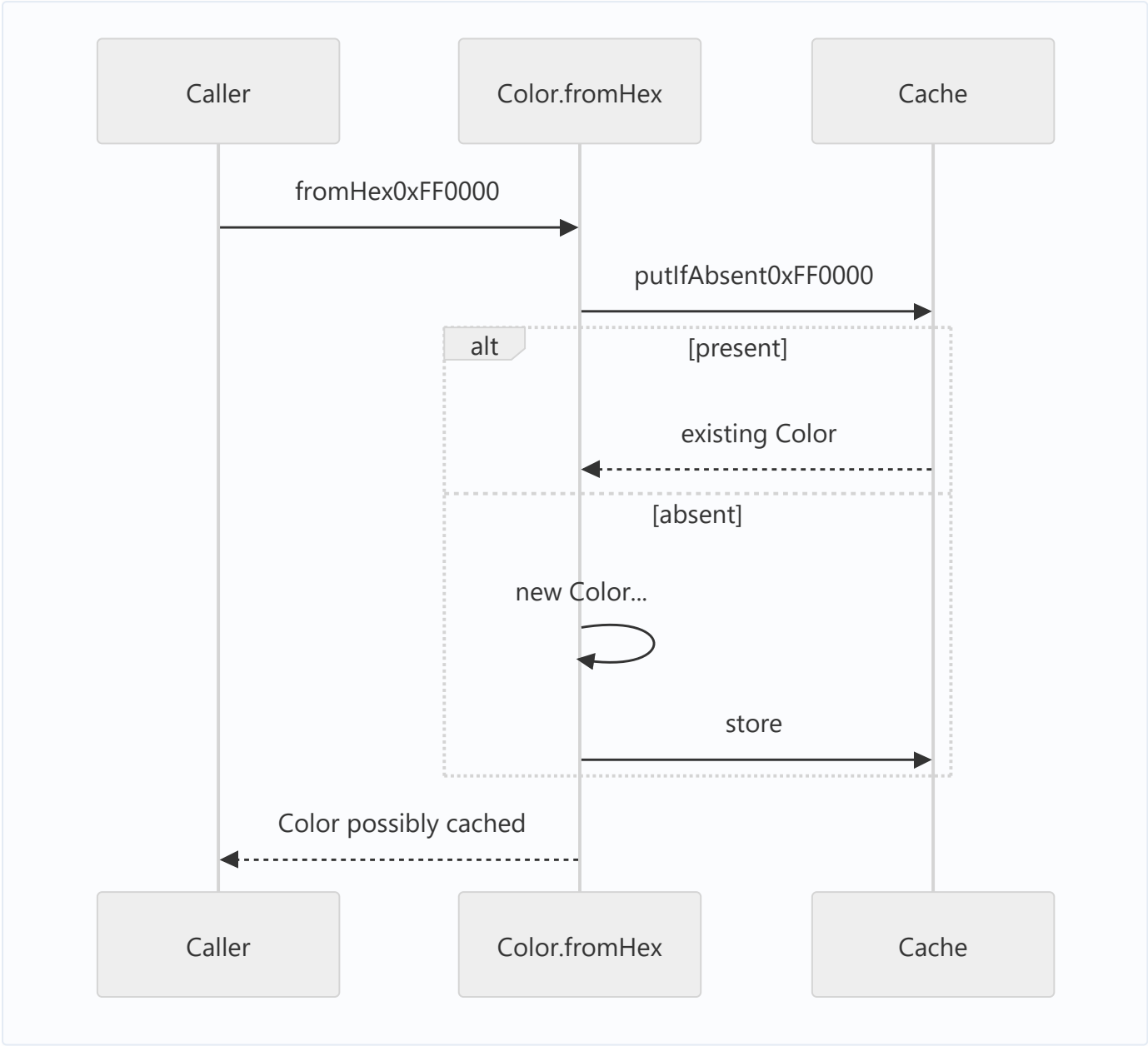
- `const` instances live in the constant pool (canonicalized). Factory returns are ordinary heap references or cached ones.

Examples

```
class Point {
  final int x, y;
  const Point(this.x, this.y);         // const generative
  const Point.origin() : this(0, 0);   // named, redirecting

  @override
  bool operator ==(Object o) => o is Point && o.x == x && o.y == y;
  @override
  int get hashCode => Object.hash(x, y);
}

class Color {
  final int argb;
  const Color(this.argb);

  static final _cache = <int, Color>{};
  factory Color.fromHex(int hex) =>
      _cache.putIfAbsent(hex, () => Color(hex)); // cached factory
```

```
}

class Logger {
  Logger._(); // private constructor
  static final Logger instance = Logger._(); // eager singleton
  void log(String m) => print('[LOG] $m');
}

// callable class
class Validator {
  final int min;
  const Validator(this.min);
  bool call(String s) => s.length >= min; // invoked as validator(x)
}

void main() {
  const a = Point(1, 2);
  const b = Point(1, 2);
  print(identical(a, b)); // true — const canonicalized

  final c1 = Color.fromHex(0xFF0000);
  final c2 = Color.fromHex(0xFF0000);
  print(identical(c1, c2)); // true — same cached instance

  Logger.instance.log('hi'); // singleton
  print(identical(Logger.instance, Logger.instance)); // true

  const isLongEnough = Validator(3);
  print(isLongEnough('ab'));  // false
  print(isLongEnough('abc')); // true — object called like a function
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`this` in an initializer list	Object not built yet	Use fields/params, not `this`
Expecting a factory to always create new	It may cache/return subtype	Read the contract; don't assume identity
Mutable global instead of a proper singleton	Uncontrolled state	Private ctor + single static instance (or DI)
Overusing singletons	Hidden global state, hard to test	Prefer dependency injection (Module 14)
`const` ctor with non-final field	Won't compile	Make all fields `final`

Best Practices

- Use initializer lists for `final` fields, asserts, and `super(...)`.
- Add `const` constructors to immutable value types (unlocks `const` usage/perf).
- Use factories for caching, subtype selection, and validation-at-construction.
- Prefer **DI over singletons**; when you must, use a private-constructor singleton and keep it stateless/small.
- Reserve callable classes for genuinely function-like objects (validators, formatters, strategies).

Performance

- `const`/cached factories avoid allocations. Singletons avoid repeated construction but can hide lifecycle/testability costs.

Advantages / Disadvantages

- + Precise control over creation; enables immutability, caching, singletons, function-like objects.
- − Factory/singleton indirection can obscure identity and complicate testing if overused.

Interview Questions

1. 🟢 **Named vs factory constructor?** — Named: an additional constructor that always creates a new instance normally. Factory: may return an existing/cached instance or a subtype and has no initializer list.
2. 🟢 **What is an initializer list and when is it required?** — The `: field = v, assert(...)` before the body; required to set `final` fields, run asserts, and call `super(...)` — all before `this` exists.
3. 🟡 **How do you implement a singleton in Dart?** — Private constructor `Foo._()` plus a single `static final Foo instance = Foo._();` (or a lazy `late final`).
4. 🟡 **Why can't a factory use `this`?** — No instance exists yet; the factory decides *what* to return, so there's nothing to reference.
5. 🟡 **What is a callable class?** — A class defining `call(...)`; its instances can be invoked like functions (`obj(args)`).
6. 🔴 **Why prefer DI over singletons?** — Singletons are global mutable state: they couple code, hide dependencies, and make testing/mocking hard. DI injects the instance, keeping code testable and swappable.
7. 🔴 **How does a factory enable the flyweight/cache pattern?** — By returning a shared cached instance for equal inputs instead of allocating a new object each time.

Senior Engineer Tips

- A "singleton" registered in a DI container gives you one instance *and* testability — best of both.
- Use factory constructors to validate and normalize inputs so invalid objects can never be constructed.
- Callable classes + `const` make excellent injectable strategies (a `const Validator(3)`).

Architect Perspective

Construction policy is an architectural lever: immutable value objects (`const` ctors), controlled creation (factories/validation), and single instances via DI shape testability and coupling across the system. Favor explicit DI-managed lifetimes over ad-hoc singletons so the dependency graph stays visible and mockable (Modules 14, 40).

Summary

- Generative (new), named (alt/redirect), `const` (canonical immutable), factory (may cache/subtype), private (`_`, blocks external new).
- Singletons = private ctor + single static instance; callable classes define `call`.
- Prefer DI over singletons; use factories for caching/validation; `const` for value types.

Revision Notes

- Initializer list: `final` fields/asserts/`super`, runs before body, no `this`.
- Factory: may return cached/subtype, no initializer list, no `this`.
- Singleton: `Foo._()` + `static final instance`.
- Callable: define `call(...)` → `obj(args)`.
- Prefer DI > singleton.

Practice Questions

1. Why does a `const` constructor require all fields to be `final`?
2. Give a case where a factory constructor returns a subtype.
3. Why are singletons considered a testability risk?

Coding Questions

1. Implement `Temperature` with `const` ctor, `Temperature.celsius`/`.fahrenheit` named ctors, and value equality.

2. Build a `ShapeFactory` factory constructor returning `Circle` / `Square` based on input.
3. Create a callable `RateLimiter` object that returns whether a call is allowed now.

Mini Project

Object-creation patterns kit (pure Dart): Implement an immutable `Money` (`const` + named ctors + equality), a caching `Icon.named(...)` factory, a DI-friendly `Clock` (interface + real/fake), and a callable `PasswordPolicy` . Wire a tiny service using them. Acceptance: no ad-hoc mutable globals; factory caching proven by `identical` ; `Clock` swappable in tests; `dart analyze` clean.

Immutability (Immutable Objects, `copyWith`, Value Equality)

> An immutable object never changes after construction; you "change" it by producing a new copy — the foundation of predictable state, cheap change detection, and safe concurrency.

Introduction

Immutability means an object's fields are `final` and never mutate. To model "edits," you create a modified **copy** (`copyWith`). This file covers building immutable classes, `const` immutability, value equality (`==` / `hashCode`), `copyWith`, and why immutability is central to Flutter's rebuild model and modern state management.

Why this concept exists

Mutable shared state is the root of most concurrency bugs, hard-to-trace UI glitches, and "who changed this?" debugging. Flutter's reactive model compares old vs new state to decide what to rebuild; that comparison is only reliable and cheap when state is immutable with value equality. Immutability makes state **predictable, diffable, and shareable** (even across isolates).

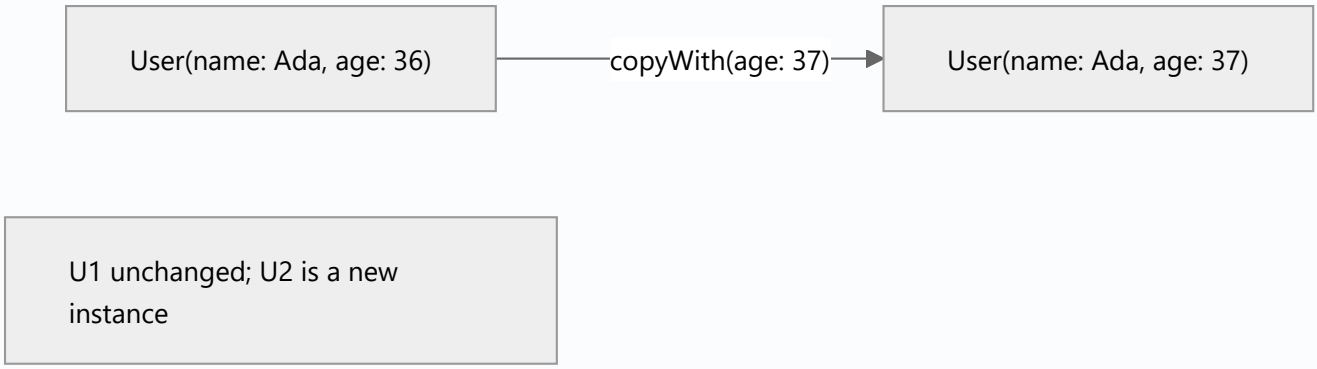
Real-world analogy

A mutable object is a **whiteboard** anyone can edit — you never know its history or who's editing now. An immutable object is a **printed document**: to make changes you print a new version (`copyWith`), leaving the original intact. You can hand copies to many people safely; nobody can alter the original under you.

Problem Statement

Model an immutable `User` with value equality and a `copyWith`, so state updates create new instances and equal users compare equal (enabling rebuild-skipping). You'll implement it by hand, then see how `freezed` / `equatable` automate it.

Internal Working



- All fields `final`; a `const` constructor when possible.
- **Value equality:** override `==` and `hashCode` together so equal-by-content objects are equal (needed for diffing, Set/Map keys, `Bloc`/`Riverpod` selectors).
- `copyWith`**:** returns a new instance with some fields replaced; unspecified fields keep old values.
- Immutable collections: expose `List.unmodifiable`/`const []` to prevent internal mutation.
- Tools: `package:equatable` (value equality), `package:freezed` (immutable data classes + `copyWith` + unions), Dart 3 `records` (built-in value equality for tuples).

Memory Representation

- Each edit allocates a new object; unchanged nested immutable objects can be **shared** (structural sharing) rather than deep-copied — so immutability isn't as expensive as it sounds.

Compiler Behavior

- `const` constructors require all-`final` fields and enable canonicalization.
- `@immutable` (from `meta`) makes the analyzer warn if a subtype adds mutable state.

Runtime Behavior

- Value equality drives framework decisions: `if (oldState == newState)` skip rebuild. Wrong/absent equality → missed or excessive rebuilds.
- Copying is O(fields) but shallow (shares nested immutables).

Flutter Engine Behavior

Not applicable at engine level, but immutability + `const` widgets let the framework skip subtree rebuilds; value equality powers `BuildWhen`/selectors to avoid unnecessary paints (Modules 11, 21).

Dart VM Behavior

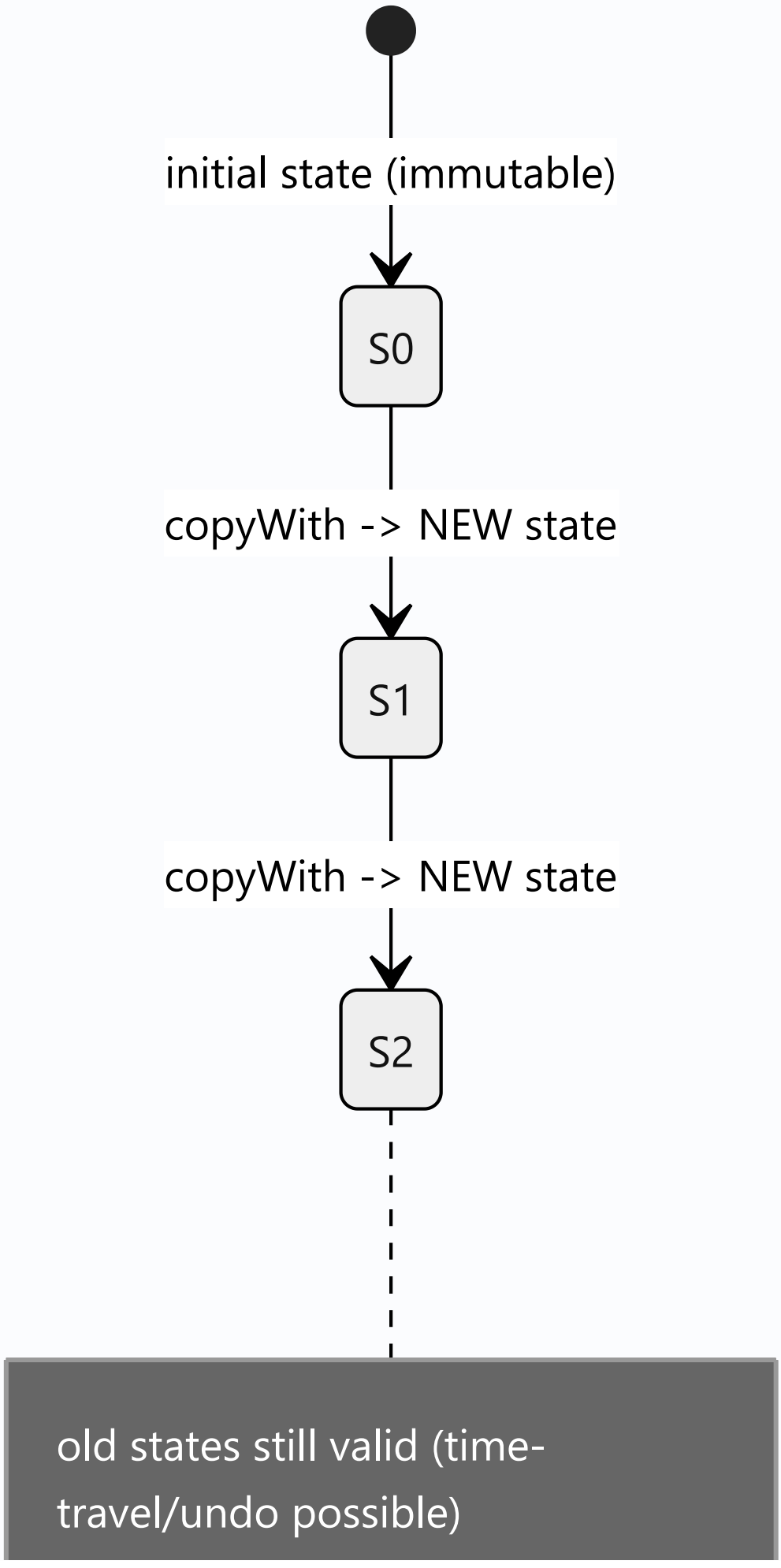
- Immutable objects are GC-friendly (no long-lived mutable references creating surprising retention); `const` instances are shared canonical objects.

Examples

```
import 'package:meta/meta.dart';

@immutable
class User {
  final String name;
  final int age;
  final List<String> roles;

  User({required this.name, required this.age, List<String>? roles})
      : roles = List.unmodifiable(roles ?? const []);

  User copyWith({String? name, int? age, List<String>? roles}) => User(
        name: name ?? this.name,
        age: age ?? this.age,
        roles: roles ?? this.roles,
      );

  @override
  bool operator ==(Object o) =>
      o is User &&
      o.name == name &&
      o.age == age &&
      _listEq(o.roles, roles);

  @override
  int get hashCode => Object.hash(name, age, Object.hashAll(roles));

  static bool _listEq(List a, List b) {
    if (a.length != b.length) return false;
    for (var i = 0; i < a.length; i++) {
      if (a[i] != b[i]) return false;
    }
    return true;
```

```
  }
}

void main() {
  final u1 = User(name: 'Ada', age: 36, roles: ['admin']);
  final u2 = u1.copyWith(age: 37);

  print(u1.age); // 36 — original unchanged
  print(u2.age); // 37 — new copy

  final u3 = User(name: 'Ada', age: 37, roles: ['admin']);
  print(u2 == u3); // true — value equality
  print({u2, u3}.length); // 1 — equal objects dedupe in a Set

  // immutability of collection:
  // u1.roles.add('x'); // throws: Unsupported (unmodifiable list)
}
```

Diagrams


initial state (immutable)
S0
copyWith -> NEW state
S1
copyWith -> NEW state
S2
old states still valid (time-travel/undo possible)

Common Mistakes

Mistake	Why	Fix
Overriding `==` without `hashCode`	Breaks Set/Map, diffing	Override both (or use equatable/freezed)
Exposing a mutable `List` field	Callers mutate internal state	`List.unmodifiable` / `const`
Deep-mutating a nested object	Immutability broken	Copy nested immutables too
Hand-writing equality for many fields	Error-prone	Use `equatable` / `freezed` /records
Assuming copies are expensive	They share nested immutables	Measure; structural sharing is cheap

Best Practices

- Make domain/state models immutable: all `final`, `const` where possible, value equality, `copyWith`.
- Prefer `freezed` (data classes + unions + `copyWith`) or `equatable` for boilerplate-free equality.
- Expose collections as unmodifiable views.
- Keep nested models immutable too (immutability is only as deep as its weakest field).

Performance

- Change detection becomes O(1)-ish reference/ `==` checks; rebuild-skipping saves paint work.
- Copies allocate but share nested immutables; churn only matters in very hot update loops (then consider targeted mutation behind an immutable facade).

Advantages / Disadvantages

- **+** Predictable state, reliable diffing/rebuild-skipping, safe sharing/concurrency, easy undo/time-travel.
- **−** More allocation; boilerplate without codegen; must be disciplined about nested mutability.

Interview Questions

1. 🟢 **How do you make an immutable class in Dart?** — All fields `final`, a `const` constructor when possible, value equality (`==` + `hashCode`), and a `copyWith` for edits.

2. 🟢 **Why override `==` and `hashCode` together?** — Hash collections and framework diffing use both; overriding one alone breaks Set/Map semantics and change detection.
3. 🟡 **Why is immutability important in Flutter?** — Reactive rebuilds rely on comparing old/new state; immutable value-equal state makes those comparisons correct and cheap, enabling rebuild-skipping.
4. 🟡 **What is `copyWith` and why not mutate in place?** — It returns a modified copy, preserving the original; mutation breaks predictability, diffing, and shareability.
5. 🟡 **Is copying immutable state expensive?** — Usually not: unchanged nested immutables are shared (structural sharing); only changed fields differ.
6. 🔴 **How do `freezed` / `equatable` help?** — They generate `==` / `hashCode` / `copyWith` (and unions for freezed), removing error-prone boilerplate.
7. 🔴 **What's the risk of a mutable field inside an "immutable" class?** — It leaks mutability; callers can change internal state, breaking equality/diffing. Use unmodifiable/immutable nested types.

Senior Engineer Tips

- Adopt `freezed` for state classes early; hand-written equality across a big model set is a bug farm.
- Treat "expose unmodifiable views" as a hard rule for public APIs returning collections.
- For hot, high-frequency updates (e.g., animation state), an immutable snapshot per frame is fine; measure before optimizing to mutation.

Architect Perspective

Immutability is a load-bearing architectural decision: it makes state management (BLoC/Riverpod), undo/redo, caching, and cross-isolate messaging correct and simple. Standardize immutable models + value equality across the codebase; it pays off in fewer heisenbugs and reliable rebuild performance at scale (Modules 11, 40, 46).

Summary

- Immutable = all `final`, edits via `copyWith`, value equality via `==` / `hashCode`.
- Enables predictable, diffable state and rebuild-skipping; expose unmodifiable collections.
- Use `freezed` / `equatable` /records to avoid boilerplate; keep nesting immutable.

Revision Notes

- Immutable: `final` fields + `const` ctor + `==` / `hashCode` + `copyWith`.
- Value equality powers diffing / Set-Map keys / rebuild-skip.

- Expose `List.unmodifiable`; keep nested models immutable.
- `freezed` / `equatable` /records automate equality + copyWith.

Practice Questions

1. Why does missing `hashCode` break a `Set<User>`?
2. How does immutability make undo/redo trivial?
3. When might immutable copying actually hurt performance, and what would you do?

Coding Questions

1. Implement an immutable `Address` with `copyWith` and value equality (no packages).
2. Convert a mutable `CartItem` to immutable + `copyWith`; update quantity via copy.
3. Rewrite the hand-written `User` using `equatable` (or `freezed`) and compare boilerplate.

Mini Project

Immutable app-state core (pure Dart): Model an immutable `AppState` (user + settings + cart) with `copyWith` and value equality, and a pure reducer `AppState next(AppState, Action)` producing new states. Keep a history list to demonstrate undo. Acceptance: no in-place mutation; equal states compare equal; undo restores prior state; `dart analyze` clean.

Libraries & Packages (`library`, `part`, `import`/`export`, pub)

> A Dart *library* is the unit of privacy and reuse (usually one file); a *package* bundles libraries with a `pubspec.yaml` so others can depend on them.

Introduction

Dart's code organization has two levels: **libraries** (privacy + import boundary) and **packages** (distribution unit on pub.dev). This file covers `import` / `export` / `show` / `hide` / `as`, `part` / `part of`, library-level privacy (`_`), `pubspec.yaml`, dependency types, and versioning.

Why this concept exists

Big codebases need boundaries: what's public vs private, what a module exposes, and how to depend on external code reproducibly. Libraries give encapsulation (the `_` privacy you've used is *library*-scoped). Packages + pub give semantic-versioned, reproducible dependency management — the reason `flutter pub get` yields consistent builds.

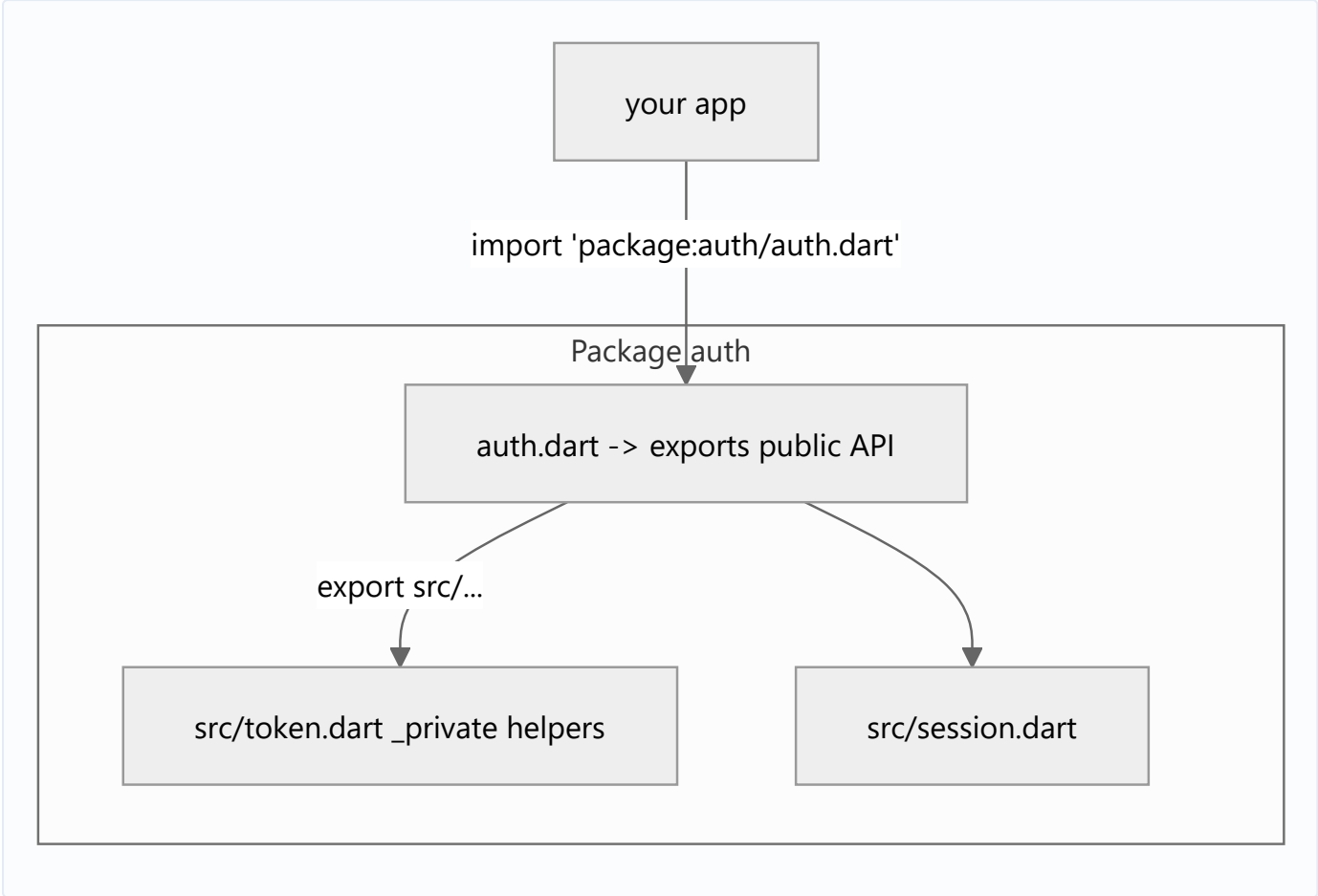
Real-world analogy

A library is a **room** in a house: things marked private (`_`) stay inside the room; a door (`export`) decides what the rest of the house sees. A package is a **prefab module** shipped with a spec sheet (`pubspec.yaml`) you bolt onto your house; the version number is the model number so you can reorder the exact same part.

Problem Statement

You're building a reusable `auth` module: hide internal helpers, expose a clean public API from one entry file, split a big library across files, and depend on `http` with a sane version constraint. You'll use `_` privacy, `export`, `part`, and `pubspec.yaml`.

Internal Working



- **Privacy** is per-library: `_name` is visible only within the same library (file, or files joined by `part`).
- `import` brings a library in; modifiers: `show` / `hide` (select names), `as` (prefix), `deferred as` (lazy load, web).
- `export` re-exposes another library's API from your entry file (the "barrel" pattern) — consumers import one file.
- `part` / `part of` splits one *library* across multiple files (they share privacy). Modern style prefers many small libraries + `export` over `part`, except for codegen (`*.g.dart`).
- **Package layout:** public API in `lib/`, private implementation in `lib/src/` (convention: don't import another package's `src/`). `pubspec.yaml` declares name, SDK/env constraints, and dependencies.

Memory Representation

Not applicable — libraries/packages are a source-organization and linking concern, not runtime data.

Compiler Behavior

- The analyzer enforces library privacy and unresolved imports.

- Tree shaking (see 14_dart_compilation.md) drops unused top-level symbols across libraries in AOT.
- `deferred as` splits code for lazy loading (primarily Flutter web).

Runtime Behavior

- Normal imports are resolved at compile/link time; `deferred` libraries load on `await lib.loadLibrary()`.

Flutter Engine Behavior

Not applicable. (Package structure affects app size via tree shaking and deferred loading on web.)

Dart VM Behavior

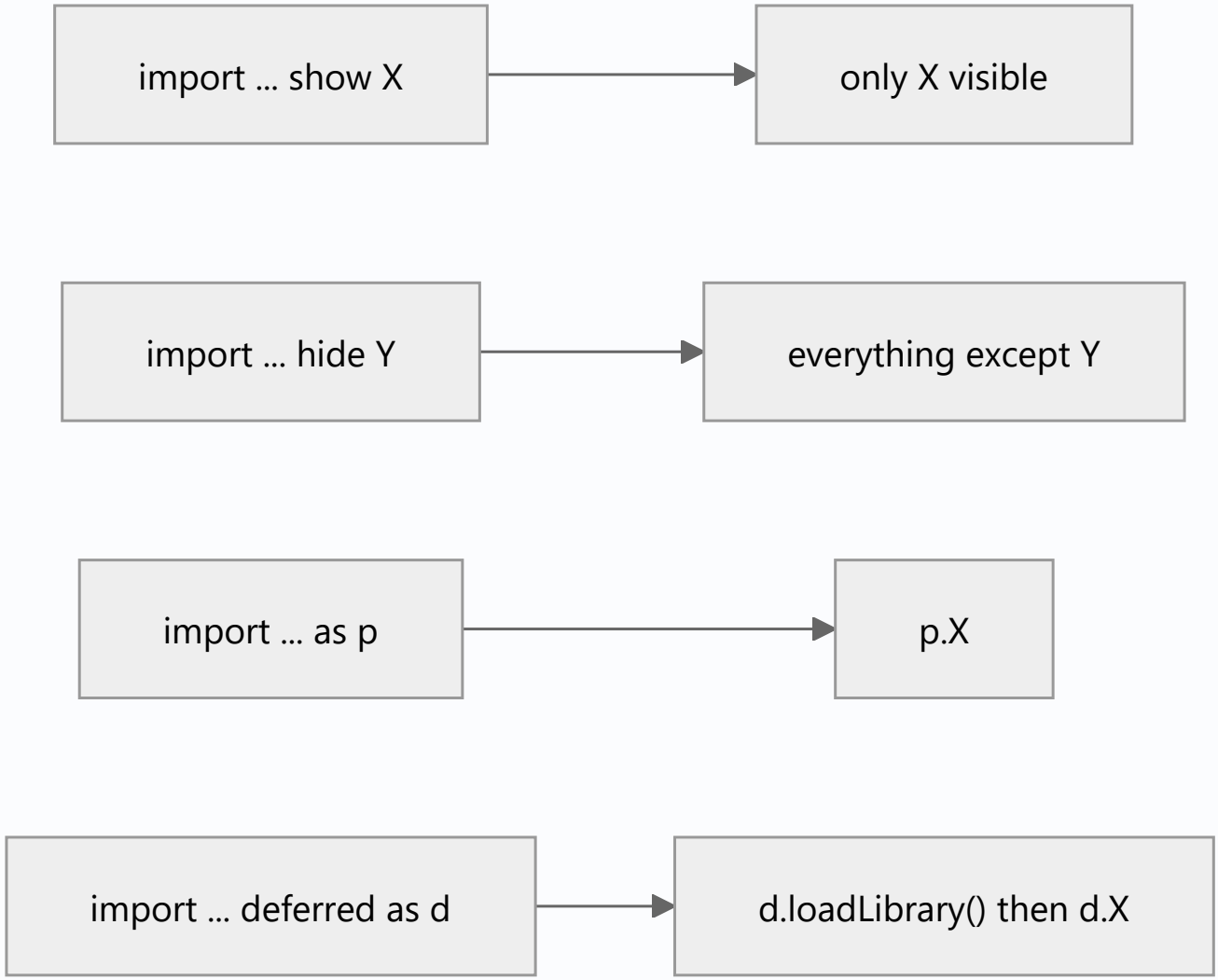
- Libraries compile into the kernel/snapshot; unused code is shaken out in AOT builds.

Examples

```
// lib/src/token.dart
String _sign(String data) => 'signed:$data'; // library-private helper
class TokenService {
  String issue(String user) => _sign(user);  // can use private in same library
}

// lib/src/session.dart
class Session {
  final String user;
  Session(this.user);
}

// lib/auth.dart  (public entry / barrel)
export 'src/token.dart' show TokenService; // expose only TokenService
export 'src/session.dart';

// ---- consumer ----
// import 'package:auth/auth.dart';
// import 'package:http/http.dart' as http;      // prefix
// import 'dart:math' show max, min;             // selective

void main() {
  // print(max(2, 9)); // from dart:math via show
}
```

```
# pubspec.yaml
name: auth
description: Reusable auth module
environment:
  sdk: ^3.4.0
dependencies:
  http: ^1.2.0          # ^ = >=1.2.0 <2.0.0 (caret / semver)
dev_dependencies:
  test: ^1.25.0
  lints: ^4.0.0
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Importing another package's `lib/src/`	Not part of its public API; breaks on updates	Import its public entry file
Overusing `part` / `part of`	Fragile, shares privacy widely	Prefer small libraries + `export` (barrel)
Loose version constraints (`any`)	Non-reproducible/breaking builds	Use caret `^` constraints; commit lockfile
Name conflicts on import	Ambiguous symbols	`show` / `hide` / `as` prefix
Putting public API in `lib/src/`	Consumers can't (shouldn't) reach it	Public API in `lib/` , internals in `lib/src/`

Best Practices

- Expose a **single barrel entry** (`lib/package_name.dart`) via `export` ; keep internals in `lib/src/` .
- Use `show` / `hide` / `as` to keep imports clean and unambiguous.
- Pin dependencies with caret constraints; commit `pubspec.lock` for apps (not for published packages).
- Reserve `part` for generated files; otherwise prefer many focused libraries.
- Follow semantic versioning when publishing.

Performance

- Good modularization + tree shaking reduces binary size; `deferred` imports cut initial web load.

Advantages / Disadvantages

- + Encapsulation, reusable distribution, reproducible builds, clean public APIs.
- − `part` fragility if overused; dependency management/version conflicts require care.

Interview Questions

1. 🟢 **How is privacy scoped in Dart?** — Per library (file, or files joined by `part`): a `_name` is visible only within the same library, not per class.
2. 🟢 `import` **vs** `export` **?** — `import` brings names into your library; `export` re-exposes another library's names from yours (barrel pattern).
3. 🟡 `show` **/** `hide` **/** `as` **?** — Selectively import specific names, exclude names, or prefix an import to avoid conflicts.
4. 🟡 `part` **/** `part of` **vs multiple libraries?** — `part` splits one library across files (shared privacy); modern style favors separate libraries + `export` , reserving `part` for codegen.
5. 🟡 **What does** `^1.2.0` **mean?** — Caret/semver: `>=1.2.0 <2.0.0` — compatible updates within the same major version.
6. 🔴 **Why not import a package's** `lib/src/` **?** — It's private-by-convention implementation; it can change/break across versions. Only the public entry is stable API.
7. 🔴 **What is a deferred import for?** — Lazy-loading a library on demand (`loadLibrary()`), primarily to reduce initial load size on Flutter web.

Senior Engineer Tips

- Design packages with a **narrow public surface**: one barrel, `src/` internals, and `show` on exports to hide accidentals.

- Use a monorepo with path/workspace dependencies for modular apps (Module 45 Modular Architecture).
- Keep `pubspec.lock` committed for apps to guarantee reproducible CI builds.

Architect Perspective

Library/package boundaries *are* your architecture's enforcement mechanism: separate feature packages with explicit public APIs prevent cross-layer leakage far better than folder conventions alone. This underpins modular and Clean Architecture at scale (Modules 40, 44, 45).

Summary

- Library = privacy + import unit (per file/ `part`); package = distribution unit with `pubspec.yaml`.
- `import` / `export` / `show` / `hide` / `as` / `deferred` control visibility; barrel + `src/` gives clean public APIs.
- Use semver caret constraints; commit lockfiles for apps; reserve `part` for codegen.

Revision Notes

- `_privacy` = library-scoped (file/ `part`).
- `import` in, `export` out (barrel); `show` / `hide` / `as` / `deferred`.
- Public API in `lib/`, internals in `lib/src/`; never import others' `src/`.
- `^1.2.0` = `>=1.2.0 <2.0.0`; commit `pubspec.lock` for apps.

Practice Questions

1. Why is `_helper` in one file invisible to a class in another file (without `part`)?
2. When is `export` (barrel) better than making consumers import many files?
3. What breaks if you depend on `any` versions?

Coding Questions

1. Structure a mini package with `lib/calc.dart` (barrel), `lib/src/ops.dart`, and a private helper; expose only the public API.
2. Write imports using `show`, `hide`, and `as` to resolve a name clash between two libraries.
3. Draft a `pubspec.yaml` with runtime + dev deps and correct caret constraints.

Mini Project

Reusable utility package (pure Dart): Create a package with a barrel entry exporting a curated API, internals under `src/` with library-private helpers, a `pubspec.yaml` with pinned deps, and unit tests. Document the public API. Acceptance: only intended symbols are importable; internals hidden; `dart pub get` + `dart test` pass; `dart analyze` clean.

JSON & Serialization

> Serialization is converting objects to/from a transferable format (JSON); doing it at a strict boundary — `dynamic` in, typed models out — is what keeps the rest of your app safe.

Introduction

Almost every app talks JSON to an API. Dart's `dart:convert` gives `jsonEncode` / `jsonDecode`, but decoded JSON is `dynamic`. This file covers manual `fromJson` / `toJson`, code generation (`json_serializable` / `freezed`), handling nulls/nesting/lists/enums, and why serialization is a **type firewall**.

Why this concept exists

Wire formats are untyped text; your app wants typed, validated objects. Serialization bridges them. Concentrating parsing at the boundary prevents `dynamic` from leaking everywhere (the failure mode from 02_data_types.md) and turns malformed data into a clear, single point of failure instead of scattered runtime crashes.

Real-world analogy

Serialization is **customs at a border**. Goods (JSON) arrive loosely packed and unverified. Customs (`fromJson`) inspects, validates, and repackages them into your country's standard containers (typed models). Nothing enters the country unchecked; on the way out, `toJson` re-packs into the international format.

Problem Statement

Parse a nested API response (`user` with a list of `orders`, an optional `nickname`, an enum `status`) into typed models, validate it, and serialize back — first by hand, then with codegen. You'll build a robust `fromJson` / `toJson`.

Internal Working

- `jsonDecode(str)` → a tree of `Map<String,dynamic>`, `List<dynamic>`, `num`, `String`, `bool`, `null`.
- `jsonEncode(obj)` → string; it calls `toJson()` on objects that define it (or you pass a `toEncodable`).

- **Manual:** write `factory Model.fromJson(Map<String,dynamic>)` casting each field, and `Map<String,dynamic> toJson()`.
- **Codegen:** annotate with `@JsonSerializable()` (json_serializable) or use `freezed` + `fromJson`; run `dart run build_runner build` to generate `*.g.dart`.
- Nested/lists: map recursively (`(json['orders'] as List).map((e)=>Order.fromJson(e as Map<String,dynamic>)).toList()`).

Memory Representation

- Decoded JSON is a heap tree of maps/lists. Large payloads are large trees — parse them off the UI isolate (`compute`) to avoid jank (04_isolates.md).

Compiler Behavior

- `jsonDecode` returns `dynamic`; the compiler can't check field access — you must cast, which is exactly why you confine it to `fromJson`.
- Codegen produces typed, analyzable `*.g.dart` with explicit casts.

Runtime Behavior

- Bad casts/missing fields throw at parse time (`TypeError` / `FormatException`) — surface them as clear errors.
- `jsonEncode` throws if it hits a non-encodable object without `toJson` / `toEncodable`.

Flutter Engine Behavior

Not applicable. (Parsing big responses on the main isolate causes dropped frames — offload.)

Dart VM Behavior

- Parsing is CPU work on the calling isolate; `compute(jsonDecodeAndMap, raw)` moves it to a worker.

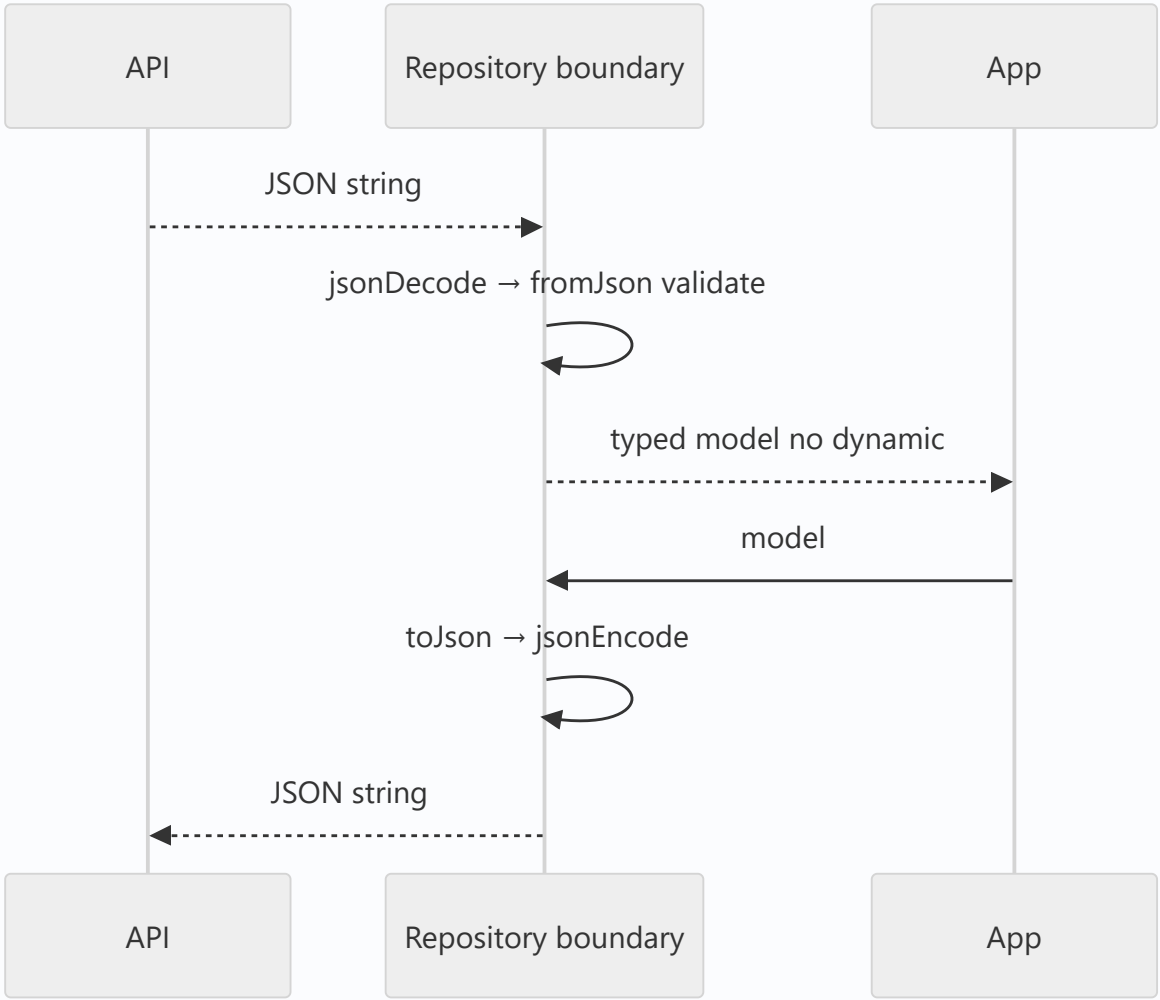
Examples

```
import 'dart:convert';


enum OrderStatus { pending, paid, cancelled }


class Order {
  final String id;
```

```
  final int amount;
  final OrderStatus status;
  Order({required this.id, required this.amount, required this.status});

  factory Order.fromJson(Map<String, dynamic> j) => Order(
        id: j['id'] as String,
        amount: (j['amount'] as num).toInt(),
        status: OrderStatus.values.byName(j['status'] as String),
      );

  Map<String, dynamic> toJson() =>
      {'id': id, 'amount': amount, 'status': status.name};
}

class User {
  final String name;
  final String? nickname; // optional
  final List<Order> orders;
  User({required this.name, this.nickname, required this.orders});

  factory User.fromJson(Map<String, dynamic> j) => User(
        name: j['name'] as String,
        nickname: j['nickname'] as String?, // null-safe optional
        orders: (j['orders'] as List<dynamic>)
            .map((e) => Order.fromJson(e as Map<String, dynamic>))
            .toList(),
      );

  Map<String, dynamic> toJson() => {
        'name': name,
        if (nickname != null) 'nickname': nickname, // omit null
        'orders': orders.map((o) => o.toJson()).toList(),
      };
}

void main() {
  const raw = '''
  {"name":"Ada","orders":[{"id":"o1","amount":1200,"status":"paid"}]}
  ''';
```

```
  final user = User.fromJson(jsonDecode(raw) as Map<String, dynamic>);
  print('${user.name}: ${user.orders.first.status}'); // Ada: OrderStatus.paid

  final back = jsonEncode(user.toJson());
  print(back); // {"name":"Ada","orders":[{"id":"o1","amount":1200,"status":"paid"}]}
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Leaking `Map<String,dynamic>` into the app	Loses type safety everywhere	Parse to models at the boundary
Unchecked `as int` on a `num`	JSON numbers may be double	`(j['x'] as num).toInt()`
Ignoring null/missing keys	Runtime crash	Use `as T?` + defaults; validate
Parsing huge JSON on UI isolate	Jank	`compute` / `Isolate.run`
Hand-writing many models	Error-prone/tedious	Use `json_serializable` / `freezed`
Persisting enum `index`	Reorder corrupts data	Serialize enum `name`

Best Practices

- Treat serialization as a **firewall**: `dynamic` only inside `fromJson`; typed models everywhere else.
- Validate required fields and throw a clear `FormatException` on bad data.
- Use codegen (`json_serializable` / `freezed`) for anything beyond a couple of models.
- Offload large parses to an isolate.
- Serialize enums by `name`; omit nulls when the API expects absence.

Performance

- `jsonDecode` is O(size); big payloads belong on a worker isolate.
- Codegen parsers are as fast as hand-written and far less error-prone.

Advantages / Disadvantages

- **+** Type-safe boundary, single failure point, codegen removes boilerplate.
- **–** Manual parsing is tedious/fragile; codegen adds a build step; large parses need isolates.

Interview Questions

1. 🟢 **What does `jsonDecode` return?** — A `dynamic` tree of `Map<String,dynamic>` / `List` / `num` / `String` / `bool` / `null`.
2. 🟢 **Why keep parsing at the boundary?** — To prevent `dynamic` from leaking through the app; malformed data fails in one clear place.
3. 🟡 **How do you parse a nested list of objects?** — Cast the list, `map` each element through its `fromJson`, `.toList()`.
4. 🟡 **Manual vs codegen serialization?** — Manual is fine for a few models; `json_serializable` / `freezed` generate `fromJson` / `toJson` (and equality/copyWith for freezed),

reducing bugs at scale.

5. 🟡 **Why serialize enums by `name` not `index`?** — `index` shifts if you reorder values, corrupting stored/transmitted data; `name` is stable.
6. 🔴 **Why parse large JSON off the main isolate?** — Decoding is CPU-bound and blocks the event loop → dropped frames; use `compute` / `Isolate.run`.
7. 🔴 **How do you handle API nulls/optional fields safely?** — Type fields as nullable (`String?`), cast with `as T?`, and apply defaults/validation in `fromJson`.

Senior Engineer Tips

- Separate **DTOs** (mirror the wire) from **domain entities** (your model); map between them so API changes don't ripple into the domain (Module 40).
- Centralize a `safeCast<T>`/validation helper to standardize error messages.
- For polymorphic JSON, use a discriminator field + a `switch`/pattern to pick the subtype.

Architect Perspective

Serialization strategy is a boundary-layer decision: DTOs + mappers isolate the domain from API drift, codegen enforces consistency, and isolate offloading protects UX. Combined with typed failures (Module 38), this yields a resilient data layer that fails loudly at exactly one place.

Summary

- `dart:convert` gives `jsonEncode` / `jsonDecode`; decoded JSON is `dynamic`.
- Parse to typed models at the boundary (manual or codegen); validate, handle nulls, serialize enums by `name`.
- Offload big parses to isolates; separate DTOs from domain entities.

Revision Notes

- `jsonDecode` →dynamic tree; confine to `fromJson`.
- Nested lists: `(j['x'] as List).map(fromJson).toList()`.
- Enums by `name`; omit/allow nulls with `as T?` + defaults.
- Big parse → `compute`; use `json_serializable` / `freezed`; DTO ≠ entity.

Practice Questions

1. Why is `(j['amount'] as num).toInt()` safer than `j['amount'] as int`?
2. How would you parse a response whose `data` is either an object or a list?
3. What breaks if you stored an enum by index and later reordered it?

Coding Questions

1. Write `fromJson` / `toJson` for a `Product` with nested `Category` and a `List<String> tags`.
2. Add codegen: annotate a model with `@JsonSerializable()` and generate the parser.
3. Implement `Future<List<User>> parseUsers(String raw)` that decodes+maps inside `Isolate.run`.

Mini Project

Typed API client (pure Dart): Build DTOs + domain entities + mappers for a small API (users/orders), a `parse` layer that validates and throws clear `FormatException`s, and an isolate-backed parser for large payloads. Add tests for valid/invalid/edge JSON. Acceptance: no `dynamic` beyond `fromJson`; enums by name; big parse offloaded; `dart analyze` clean.

Memory Management & Garbage Collection

> Dart manages memory automatically with a generational garbage collector; your job is to avoid *retaining* objects longer than needed — leaks in Dart are almost always unintended references, not missing frees.

Introduction

Dart has **automatic memory management**: you never `free` / `delete`. A **generational garbage collector (GC)** reclaims unreachable objects. This file covers the heap model, the young/old generation GC, what "reachable" means, common Flutter memory leaks (uncancelled subscriptions, retained `context`, controllers), and how to find them with DevTools.

Why this concept exists

Manual memory management (C/C++) is error-prone (leaks, use-after-free). GC trades a little runtime cost for safety and productivity. But GC only reclaims what's **unreachable** — so understanding *reachability* is the key skill: a leak in Dart means something still points to an object you thought was gone.

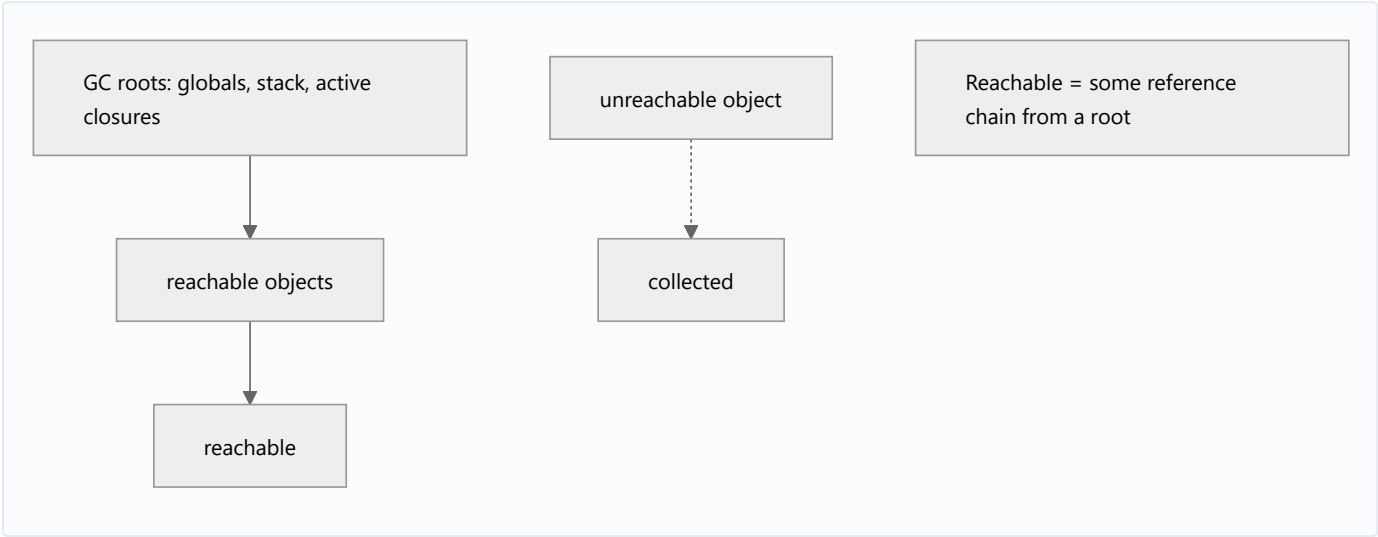
Real-world analogy

The GC is a **janitor** who throws away anything no one is "holding onto." An object is kept as long as *any* chain of references reaches it from a root (globals, the stack, active closures/subscriptions). A leak is like leaving a **sticky note** (a subscription/timer) pinned to a whiteboard you meant to erase — the janitor sees it's still referenced and won't clear it.

Problem Statement

Your Flutter screen's memory keeps climbing each time it's opened and closed. You suspect a leak. You'll learn the reachability model, the usual culprits (subscriptions/controllers/timers not disposed), and how to confirm with DevTools' memory tools.

Internal Working



Dart's GC is **generational**, exploiting "most objects die young":

- **Young/new generation** — a small space collected frequently with a fast **scavenger** (copying) collector. Cheap because most young objects are already dead.
- **Old generation** — objects that survive several young collections are **promoted**; collected less often via a **mark-sweep(-compact)** collector.
- GC runs concurrently/incrementally where possible to minimize pauses (important for frame timing).

Reachability, not scope, determines lifetime: an object is alive while any root can reach it (a static list holding it, an active `StreamSubscription`'s closure capturing it, a running `Timer`).

Memory Representation

- Objects live on the isolate's heap; each isolate has its own heap and GC (no cross-isolate references — see 04_isolates.md).
- The stack holds references (locals); the heap holds the objects. Closures move captured variables to the heap.

Compiler Behavior

- Not a compile-time concern directly, but `const` canonicalization reduces allocations, and tree shaking removes dead code (less to load, not less to GC).

Runtime Behavior

- Allocation is fast (bump-pointer in the young space). Collection pauses are usually sub-millisecond for young GC; old-gen GC is rarer and larger.

- Finalizers (`Finalizer`) can run cleanup when an object is collected, but timing is **not guaranteed** — don't rely on them for critical resource release.

Flutter Engine Behavior

GC pauses that coincide with a frame can cause jank. Excessive per-frame allocation (rebuilding heavy objects, closures, or lists every frame) increases young-GC frequency. The framework's `dispose()` lifecycle exists precisely so you release retained references (controllers, subscriptions) and let the GC reclaim them. See Module 08 Widget Lifecycle and Module 21 Performance.

Dart VM Behavior

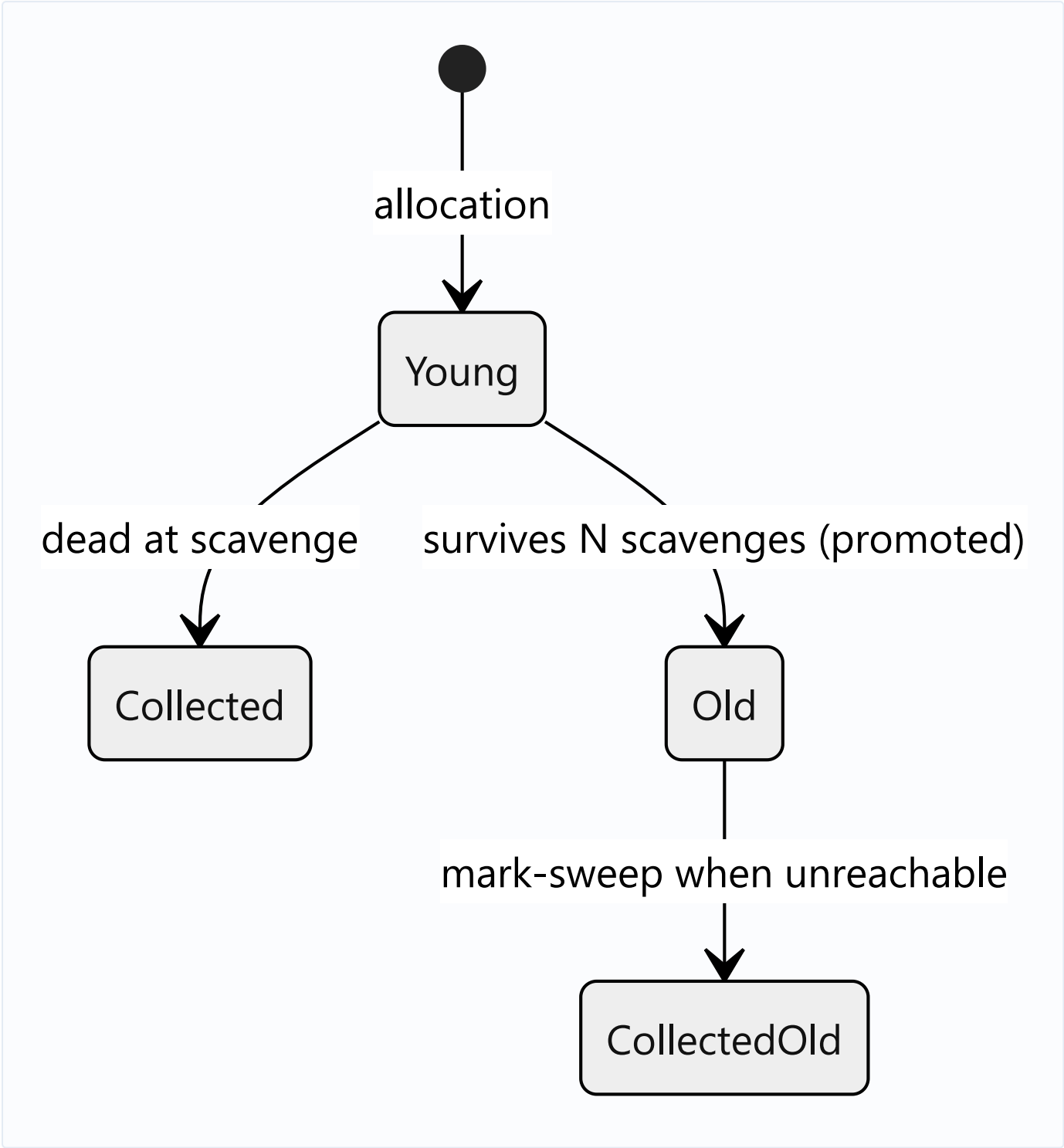
- The VM uses a generational GC (scavenger for young, mark-sweep-compact for old), tuned to keep pauses short. AOT and JIT share the model; heap is per-isolate (isolate groups may share runtime metadata).

Examples

```
import 'dart:async';


// LEAK PATTERN (Flutter-style, illustrated in pure Dart):
class Screen {
  StreamSubscription<int>? _sub;
  final _timer = <Timer>[];


  void open(Stream<int> ticks) {
    // ❌ if never cancelled, this subscription (and its captured `this`)
    //    stays reachable from the stream -> Screen can't be GC'd.
    _sub = ticks.listen((v) => _handle(v));
    _timer.add(Timer.periodic(const Duration(seconds: 1), (_) => _handle(0)));
  }


  void _handle(int v) {/* ... */}


  // ✅ release references so the GC can reclaim this object graph
  void dispose() {
    _sub?.cancel();
    for (final t in _timer) {
      t.cancel();
    }
```

```
    _timer.clear();
  }
}

void main() {
  // demonstrate reachability keeping objects alive
  final held = <Object>[];
  var o = Object();
  held.add(o); // still reachable via `held`
  o = Object(); // the FIRST object is still alive (held references it)
  print(held.length); // 1 — not collected because reachable
}
```

Diagrams


allocation
Young
dead at scavenge
survives N scavenges (promoted)
Collected
Old
mark-sweep when unreachable
CollectedOld

Common Mistakes (Flutter leaks)

Leak source	Why it retains	Fix
Uncancelled `StreamSubscription`	Stream holds the callback (+ captured `this`)	`cancel()` in `dispose()`
Undisposed controllers (`AnimationController`, `TextEditingController`, `ScrollController`)	Framework/ticker keeps references	`dispose()` them
`Timer` / `Timer.periodic` not cancelled	Scheduler holds the callback	`cancel()`
Static/global caches without eviction	Roots keep everything reachable	Bound size / evict / use weak refs
Capturing `BuildContext` / `State` in long-lived closures	Keeps whole subtree alive	Avoid capturing; cancel owners
Global `ValueNotifier` with many listeners never removed	Listeners retained	`removeListener` / `dispose`

Best Practices

- Every subscription/controller/timer has an **owner** that disposes it (in Flutter, `State.dispose`).
- Bound caches (size/TTL); consider `WeakReference` / `Expando` for associative caches that shouldn't retain.
- Minimize per-frame allocations: `const` widgets, hoist closures, reuse buffers.
- Use `Finalizer` only for best-effort native-resource cleanup, never for correctness.
- Profile with **DevTools Memory** (heap snapshot, allocation tracing, "retaining path").

Performance

- Allocation is cheap; *churn* is the cost (more young GCs). Reduce allocations in hot paths.
- Large retained graphs push objects into old gen, making eventual collection more expensive.

Advantages / Disadvantages

- + No manual free/use-after-free; fast young-gen allocation/collection; incremental pauses.
- − Leaks via unintended references still happen; GC pauses can (rarely) affect frames; finalizer timing unguaranteed.

Interview Questions

1. 🟢 **Does Dart require manual memory management?** — No; it's garbage collected. You avoid leaks by not retaining references, not by freeing.
2. 🟢 **What makes an object eligible for collection?** — Being **unreachable** — no reference chain from any GC root reaches it.

3. 🟡 **Describe Dart's GC.** — Generational: a frequent, cheap copying scavenger for the young space; a less-frequent mark-sweep(-compact) for the old space; survivors are promoted young→old.
4. 🟡 **Name three common Flutter leaks.** — Uncancelled stream subscriptions, undisposed controllers (animation/text/scroll), and uncancelled timers — all retain via held references (often capturing `this` / `context`).
5. 🟡 **Why does an uncancelled subscription leak the whole screen?** — The stream holds the `onData` closure, which captures `this` (the `State`), keeping the entire widget/state graph reachable.
6. 🔴 **Can you rely on finalizers for resource cleanup?** — No; finalizer execution timing isn't guaranteed. Use explicit `dispose` / `close`; finalizers are best-effort fallbacks.
7. 🔴 **How do you find a leak?** — DevTools Memory: take heap snapshots across open/close cycles, look for growing instance counts, and inspect the **retaining path** to see what still references the object.

Senior Engineer Tips

- Adopt a rule: "if you `listen` / `create a controller` / `start a timer`, you own its disposal." Code review for it.
- Watch static singletons and app-wide notifiers — they're roots; anything they reference lives forever.
- Reduce per-frame garbage first (it's the common, cheap win) before chasing rare old-gen pauses.

Architect Perspective

Memory discipline is a cross-cutting reliability concern. Establish lifecycle ownership conventions (dispose patterns, bounded caches, weak references for observers) and enforce them via lints/reviews and leak-detection in tests (`flutter test` + leak_tracker). At scale, unmanaged retention is a top cause of gradual slowdowns and OOM crashes on low-end devices (Module 21).

Summary

- Dart is GC-managed; objects die when unreachable. GC is generational (fast young scavenger + old mark-sweep).
- Leaks = unintended retained references — cancel subscriptions/timers, dispose controllers, bound caches.
- Reduce per-frame allocations; profile with DevTools; don't rely on finalizers.

Revision Notes

- Automatic GC; collect when **unreachable** (no root chain).
- Generational: young scavenge (frequent/cheap) → promote → old mark-sweep.
- Leaks: uncancelled subs/timers, undisposed controllers, unbounded static caches, captured `context`.
- Fix: dispose owners; bound caches; weak refs; cut per-frame allocation.
- Finalizers = best-effort only. Debug via DevTools retaining path.

Practice Questions

1. Why can an object referenced only by a still-active `Timer` never be collected?
2. Explain how capturing `this` in a stream callback leaks a whole screen.
3. When would you use `WeakReference` / `Expando`?

Coding Questions

1. Write a `Disposable` base that tracks subscriptions/timers and cancels all in `dispose()`.
2. Implement a size-bounded LRU cache that evicts (so it doesn't retain forever).
3. Use `WeakReference` to build an observer registry that doesn't keep observers alive.

Mini Project

Leak-safe resource manager (pure Dart): Build a `ResourceScope` that registers streams/timers/controllers and disposes them all at once, plus a bounded cache. Simulate open/close cycles and assert (via counters/finalizers) that resources are released. Acceptance: no leaked timers/subscriptions after `dispose`; cache stays bounded; documented reachability reasoning; `dart analyze` clean.

Dart Compilation (VM, JIT, AOT, Snapshots, Tree Shaking)

> Dart compiles two ways: **JIT** during development (enabling stateful hot reload) and **AOT** for release (native machine code, fast startup, small size) — one language, two pipelines tuned for two goals.

Introduction

Understanding Dart's compilation model explains Flutter's superpowers: sub-second **hot reload** in debug and **native, fast-starting** release builds. This file covers the Dart VM, **JIT** vs **AOT**, kernel/snapshots, **tree shaking**, `dart2js`/wasm for web, and how each affects the developer experience and app size/performance.

Why this concept exists

Development and production have opposite priorities. Development wants **fast iteration** (change code, see it live) — served by JIT + hot reload. Production wants **fast startup, predictable performance, small size, no JIT** (app-store rules on iOS forbid runtime codegen) — served by AOT. Dart supports both from one codebase, which is a core reason Flutter feels productive *and* ships fast apps.

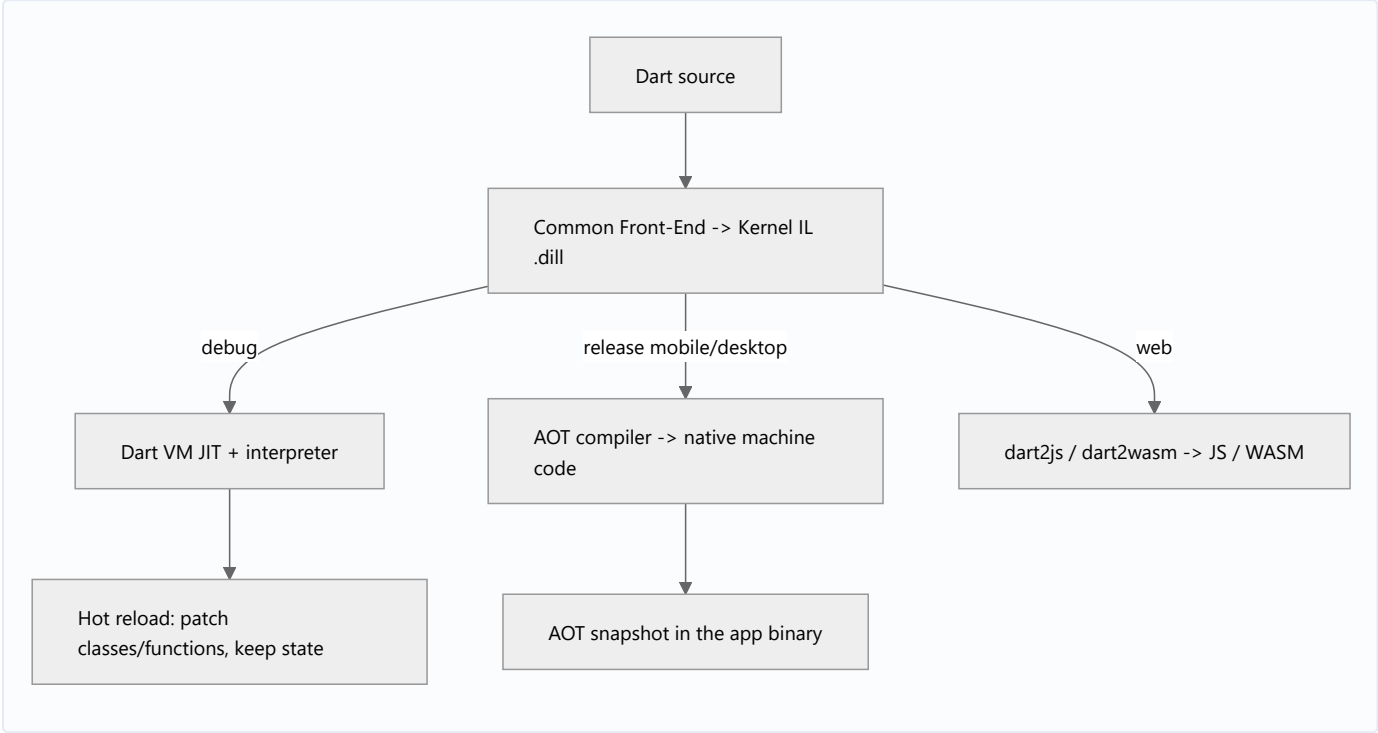
Real-world analogy

JIT is a **live theater rehearsal**: actors can change lines mid-scene and you see it instantly (hot reload), at the cost of some overhead. AOT is the **filmed, edited movie**: locked, optimized, plays back fast anywhere — but you can't change a line without re-shooting (rebuild).

Problem Statement

Explain why hot reload works in debug but not release, why release builds start faster and are smaller, and why unused code doesn't bloat your app. You'll connect JIT→hot reload and AOT→tree-shaken native binary.

Internal Working



- **Front end (CFE):** parses/type-checks Dart into **Kernel** (a.k.a. `.dill`), a language-level IL shared by all backends.
- **JIT (debug):** the VM runs Kernel via an interpreter + just-in-time compiler that optimizes hot code at runtime. Supports **hot reload** (inject updated code, preserve app state) and **hot restart** (reset state).
- **AOT (release, mobile/desktop):** compiles Kernel to **native machine code** ahead of time, bundled as an AOT snapshot. No JIT at runtime → fast startup, consistent perf, satisfies iOS's no-runtime-codegen rule.
- **Web:** `dart2js` compiles to JavaScript; `dart2wasm` compiles to WebAssembly.
- **Tree shaking:** the AOT/web compilers perform whole-program analysis and **drop unreferenced code** (functions, classes, even unused icon glyphs via `--tree-shake-icons`), shrinking the binary.

Memory Representation

- **Snapshots** serialize compiled code/data for fast loading: JIT can use app/kernel snapshots to speed startup; AOT snapshots contain native code + heap image loaded directly at launch.

Compiler Behavior

- Type checking and const evaluation happen in the front end (shared).
- Tree shaking requires a *closed world* (whole-program) view — which is why it's an AOT/web feature; reflection (`dart:mirrors`) breaks it and is therefore **unavailable in Flutter/AOT**.

Runtime Behavior

- JIT: first runs are interpreted, hot paths get optimized (and can deoptimize) → variable early performance but great flexibility.
- AOT: uniform, predictable performance from the first frame; no warmup, no JIT pauses.

Flutter Engine Behavior

- **Debug builds** ship the Dart VM with JIT → hot reload/restart via DevTools/IDE.
- **Release builds** are AOT-compiled → smaller, faster-starting, no hot reload. This is why `flutter run` (debug) supports hot reload but `flutter build apk --release` does not. See Module 10 Flutter Architecture and Module 51 Deployment.

Dart VM Behavior

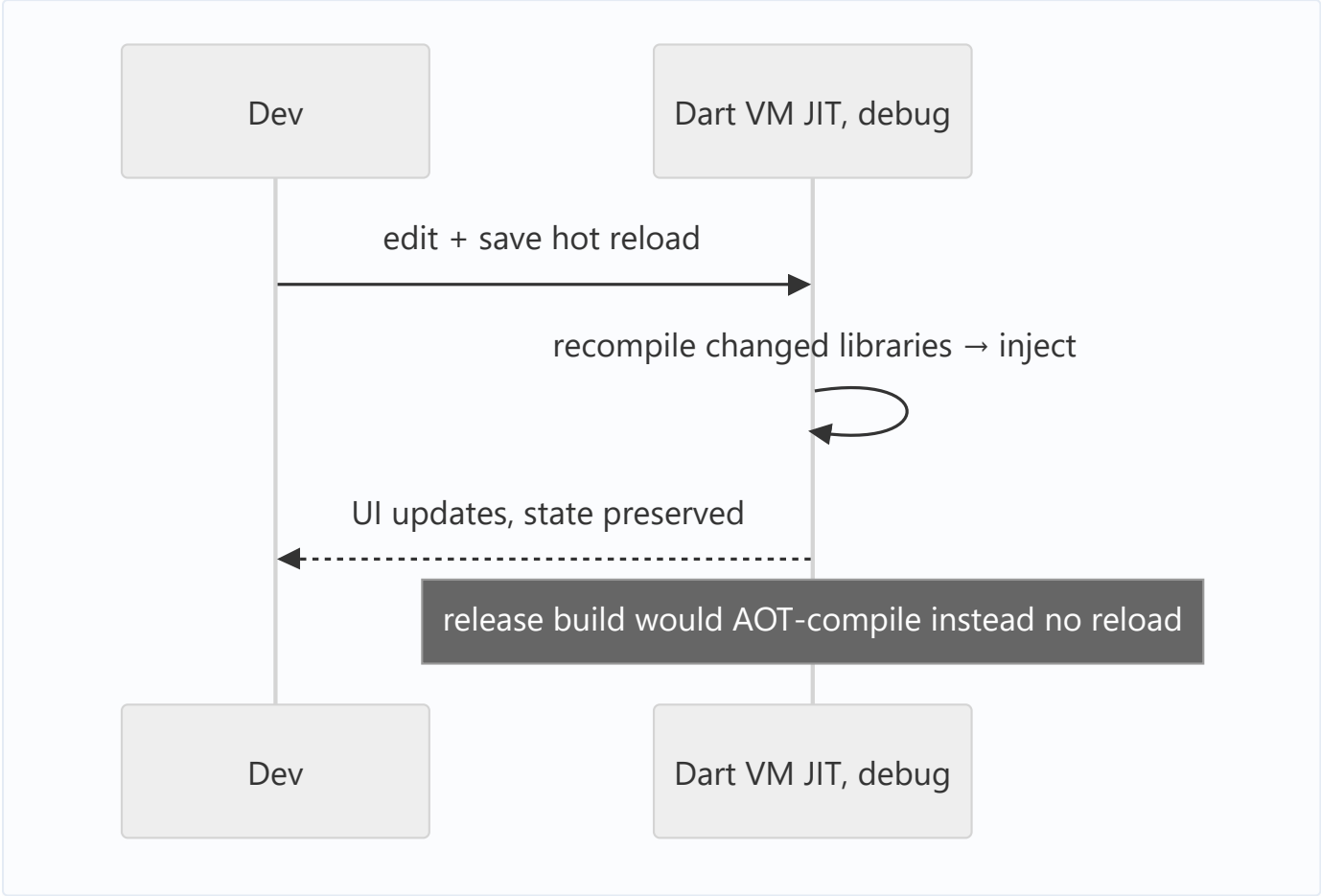
- The VM hosts isolates, GC (13_memory_and_gc.md), and (in debug) JIT. In release the VM runs precompiled AOT code (no JIT compiler shipped).

Examples

```
# Debug (JIT) — hot reload available:
flutter run                       # press 'r' = hot reload, 'R' = hot restart

# Release (AOT) — optimized, no hot reload:
flutter build apk --release
flutter build ios --release
flutter build web --release       # dart2js (or --wasm for dart2wasm)

# Pure Dart:
dart run bin/main.dart            # JIT
dart compile exe bin/main.dart    # AOT native executable
dart compile js bin/main.dart     # to JavaScript

# Icon tree shaking (Flutter release) drops unused MaterialIcons glyphs:
#   "Font asset ... tree-shaken, reducing it from XKB to YKB"
```

```
// Reflection is NOT available under AOT/Flutter (breaks tree shaking):
// import 'dart:mirrors'; // ❌ not supported in Flutter
// Use code generation (build_runner) instead of runtime reflection.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Expecting hot reload in release	Release is AOT (no JIT)	Use debug for iteration
Relying on `dart:mirrors` /reflection	Unavailable under AOT; breaks tree shaking	Use codegen (`build_runner`)
Benchmarking in debug	JIT + asserts skew results	Profile/benchmark in `--profile` / `--release`
Assuming unused imports bloat release	Tree shaking removes dead code	Still avoid unused deps that *are* referenced
Confusing hot reload vs hot restart	Reload keeps state; restart resets	Reload for UI tweaks; restart for state/main changes

Best Practices

- Iterate in **debug** (hot reload); always **profile/measure in profile/release** builds.
- Avoid reflection; adopt code generation for serialization/DI where needed.
- Keep dependencies lean; tree shaking helps but referenced-yet-unused-heavy libs still cost.

- Use `--split-debug-info` /obfuscation for release size/security (Modules 37, 51).

Performance

- AOT: fast cold start, no JIT warmup/pauses, predictable — ideal for mobile.
- JIT: flexible, good for dev; not representative of release performance.
- Tree shaking + icon shaking materially reduce app size.

Advantages / Disadvantages

- + Best-of-both: fast dev iteration (JIT/hot reload) and fast, small, secure release (AOT/tree shaking).
- − Release loses hot reload; no runtime reflection; two behavior profiles to keep in mind (debug vs release).

Interview Questions

1. 🟢 **Why does Flutter have hot reload in debug but not release?** — Debug uses the JIT-capable Dart VM (can inject updated code, keep state); release is AOT-compiled native code with no JIT.
2. 🟢 **JIT vs AOT?** — JIT compiles at runtime (flexible, warms up, enables hot reload); AOT compiles ahead of time to native code (fast startup, predictable, no runtime codegen).
3. 🟡 **What is Kernel/ `.dill` ?** — The common front-end's intermediate representation of Dart, consumed by both JIT and AOT/web backends.
4. 🟡 **What is tree shaking?** — Whole-program elimination of unreferenced code (and unused icon glyphs) during AOT/web compilation, shrinking output.
5. 🟡 **Why is `dart:mirrors` unavailable in Flutter?** — Runtime reflection defeats whole-program tree shaking (can't prove what's unused) and conflicts with AOT; codegen is used instead.
6. 🔴 **Hot reload vs hot restart?** — Reload recompiles/injects changed libraries and preserves app state; restart rebuilds and resets state (re-runs `main`).
7. 🔴 **How does Dart target the web?** — `dart2js` (to JavaScript) and `dart2wasm` (to WebAssembly), separate from the VM/AOT pipelines.

Senior Engineer Tips

- Treat debug numbers as meaningless for performance; always confirm in profile/release.
- If a package needs reflection, it won't work in Flutter release — prefer codegen-based equivalents.
- Understand snapshots when optimizing startup: AOT snapshot size and deferred loading (web) affect launch time.

Architect Perspective

The dual pipeline shapes tooling and delivery decisions: fast local iteration for teams, AOT for store-compliant, performant releases, and tree shaking + deferred loading for size budgets. Choosing codegen over reflection is a foundational constraint that influences serialization, DI, and routing architecture across the whole app (Modules 12–14, 40).

Summary

- One front end (Kernel) feeds two backends: JIT (debug, hot reload) and AOT (release, native).
- Web targets JS/WASM; tree shaking removes dead code; reflection is unavailable under AOT.
- Iterate in debug, measure in release; prefer codegen over reflection.

Revision Notes

- CFE → Kernel `.dill` → JIT (debug/hot reload) or AOT (release/native) or dart2js/wasm (web).
- Hot reload = inject + keep state (JIT only); hot restart = reset state.
- AOT: fast startup, predictable, no runtime codegen (iOS-compliant).
- Tree shaking drops dead code/icons; `dart:mirrors` unsupported → use codegen.
- Benchmark in profile/release, never debug.

Practice Questions

1. Explain to a teammate why their release build ignores hot reload.
2. Why does adopting reflection-based serialization break a Flutter release?
3. What does tree shaking do to unused code and icon fonts?

Coding Questions

1. Compare `dart run` vs `dart compile exe` startup time for a small program; explain the difference.
2. Trigger and read the "tree-shaken icons" message in a Flutter release build.
3. Convert a hypothetical reflection-based mapper to a `build_runner` codegen approach (design/pseudocode).

Mini Project

Compilation report (docs + measurements): For a small Flutter app, build debug, profile, and release variants; record startup time and APK/web size, capture the icon-tree-shaking log, and document why each differs. Add a note on why a reflection-dependent package fails in release.

Acceptance: measured numbers for all three modes; a clear JIT-vs-AOT explanation tied to observed behavior.