FLUTTERFEVER
DEVELOPER MAGAZINE

DEEPER DART. CLEANER FLUTTER.

ADVANCED DART

A practical guide to modern language features, type-safe APIs, concurrency and expressive architecture.

MODERN DART IN PRACTICE

```
switch (result) {
  case Success(value: final data) => render(data),
  case Failure(:final error)      => recover(error),
};

final output = await Isolate.run(() => compute(input));
extension type UserId(String value) {}
```

Patterns · Generics · Sealed APIs · Records · Isolates · Streams

FROM LANGUAGE FEATURES TO PRODUCTION ARCHITECTURE

Designed for Flutter developers who already know the basics.

SCAN TO LEARN MORE

flutterfever.com

Advanced Dart • Flutter architecture • AI development

START HERE

Advanced Dart: the mental models that change how you write code

This guide assumes you can already write classes, functions, collections and async code. The goal now is to make your Dart more precise, more expressive and easier to evolve.

What "advanced" means here

Not obscure syntax. Advanced Dart is about using the type system, language constructs and concurrency model to make invalid states difficult to represent and expensive work safe to execute.

A five-part roadmap

1. Type precision

Promotion, Never, dynamic boundaries, generics and variance.

2. Data modeling

Records, patterns, sealed families, value semantics and constructors.

3. API design

Class modifiers, mixins, extensions and extension types.

4. Concurrency

Futures, streams, event loop behavior and isolates.

5. Production craft

Libraries, error design, performance, testing and architecture.

Read this guide actively

For each concept, ask three questions: what problem does it remove, what contract does it create, and what trade-off does it introduce? That habit matters more than memorizing syntax.

1. Type precision beyond "strong typing"

Dart's type system becomes most valuable when you treat types as executable design constraints. A useful type narrows what callers can do and tells the analyzer enough to eliminate defensive code.

Object?, Object, dynamic and Never are not interchangeable

TYPE BOUNDARIES

```
void inspect(Object? value) {
  if (value case String s when s.isNotEmpty) {
    print(s.length);
  }
}

Never fail(String message) => throw StateError(message);
```

Prefer Object? at unknown boundaries

It accepts any value while preserving static checking. You must narrow before using members.

Use dynamic deliberately

dynamic opts out of static checks for that expression. Keep it at interop or decoding boundaries, then convert quickly.

Never is a signal

A function returning Never does not complete normally. The analyzer can use that fact for reachability and promotion.

Promotion is data-flow analysis

Promotion is not just “null check magic.” Dart tracks control flow and can narrow local variables after tests, pattern matches and early returns. Write guard clauses that help both humans and the analyzer.

```
String label(Object? input) {
  if (input == null) return 'none';
  if (input is! String) return 'unsupported';
  // input is promoted to String here.
  return input.trim();
}
```

2. Generics that express real constraints

Generics are most powerful when they encode relationships between inputs and outputs. A generic API should preserve information, not erase it.

GENERIC CONTRACTS

```
abstract interface class Repository<ID, Entity> {
  Future<Entity?> find(ID id);
  Future<void> save(Entity value);
}

T firstWhereOr<T>(Iterable<T> items, bool Function(T) test, T fallback) {
  for (final item in items) {
    if (test(item)) return item;
  }
  return fallback;
}
```

Bound type parameters when behavior depends on a capability

```
num sum<T extends num>(Iterable<T> values) =>
    values.fold<num>(0, (total, value) => total + value);
```

Good generic API

The caller keeps exact types and the compiler verifies the relationship.

Generic for its own sake

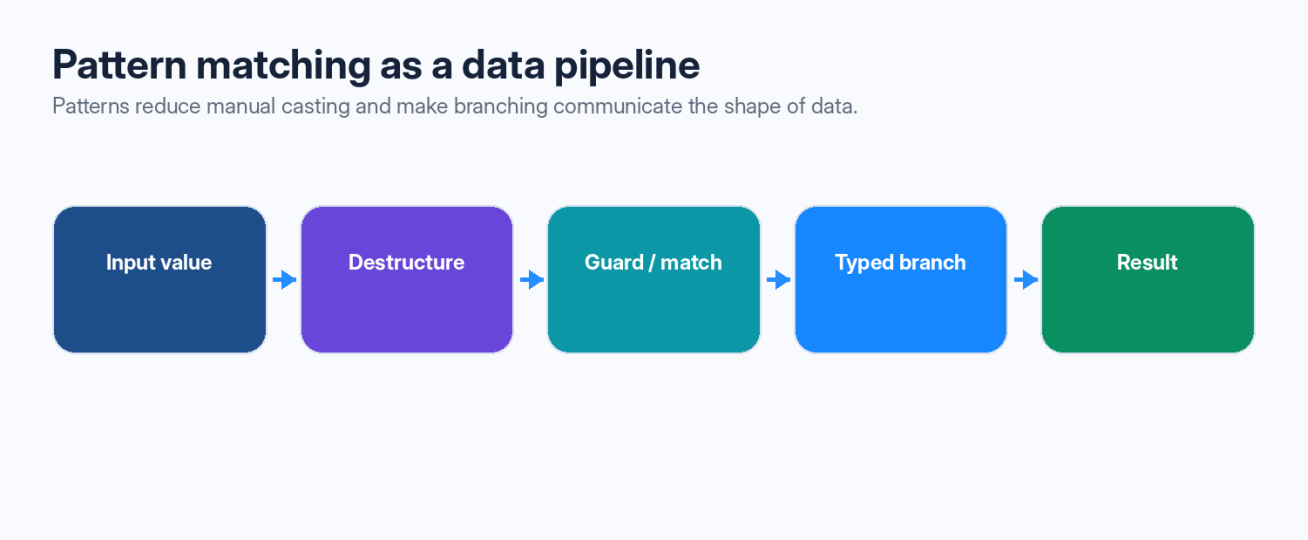
If every path immediately casts to dynamic or Object, the abstraction is probably hiding rather than preserving information.

> **Variance is an API question**
>
> Think about whether a type parameter appears in input positions, output positions, or both. Mutable generic containers are where “looks compatible” assumptions become dangerous.

MODERN DATA MODELING

Records + patterns: concise data without weak structure

Records are anonymous immutable aggregates. Patterns let code verify and destructure the shape of values in declarations, assignments, loops, if-cases and switch constructs.



3. Records when a named class adds no meaning

NAMED FIELDS + DESTRUCTURING

```
({String name, int count}) summarize(List<String> items) {
  return (name: items.first, count: items.length);
}

final (:name, :count) = summarize(['dart', 'flutter']);
```

A record is ideal for local multi-value returns and structural data that does not need identity, behavior or encapsulation. If the concept deserves invariants or methods, prefer a class.

Records have structural types

The shape and field types determine a record's type. Named field names are part of that shape; positional field names in documentation are not.

Use records for	Prefer classes for
Private transformations, parser results, paired metrics, temporary grouped values.	Domain concepts, invariants, long-lived public APIs, behavior and abstraction boundaries.

4. Pattern matching that removes manual plumbing

SWITCH EXPRESSION

```
String describe(Object value) => switch (value) {
  (int x, int y) when x == y => 'diagonal point',
  [String first, ...final rest] => '$first + ${rest.length}',
  {'status': 'ok', 'data': final data} => 'ok: $data',
  _ => 'unknown',
};
```

Pattern vocabulary worth mastering

- Record patterns destructure positional or named record fields.
- List and map patterns match selected collection shapes without manually indexing.
- Object patterns call getters and match against a type.
- Logical patterns combine alternatives or requirements.
- Guards add a boolean condition after structural matching succeeds.

> **Pattern-first thinking**
>
> Instead of asking "which casts and null checks do I need?", ask "what shape of value does this branch accept?" That shift produces shorter and safer branching code.

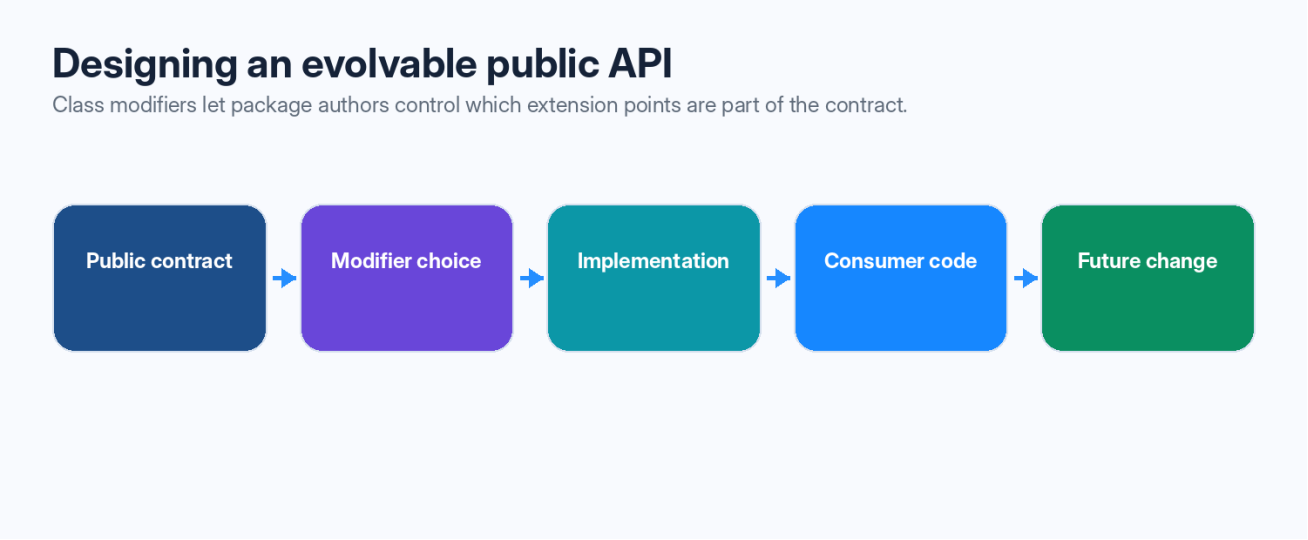
5. Exhaustive modeling with sealed types

CLOSED STATE FAMILY

```
sealed class LoadState<T> {}
final class Idle<T> extends LoadState<T> {}
final class Loading<T> extends LoadState<T> {}
final class Data<T> extends LoadState<T> { Data(this.value); final T value; }
final class Failure<T> extends LoadState<T> { Failure(this.error); final Object error; }

String text(LoadState<String> state) => switch (state) {
  Idle() => 'Ready',
  Loading() => 'Loading…',
  Data(:final value) => value,
  Failure(:final error) => 'Error: $error',
};
```

A sealed hierarchy makes the family of direct subtypes known inside its library. Exhaustive switches then become a maintenance tool: adding a new state can force all important decision points to be revisited.

6. Class modifiers as API architecture



Modifiers are not decorative keywords. They define which forms of reuse your public type allows outside its library.

Modifier	External extend?	External implement?	Intent
interface	No	Yes	Expose a contract, control inherited implementation
base	Yes (with constraints)	No	Preserve implementation inheritance invariants
final	No	No	Close both extension and implementation externally
sealed	No direct external subtypes	No	Define a closed family for exhaustive reasoning

Package author mindset

Choose the least-permissive modifier that still supports intended use. Every extension point you expose becomes part of your compatibility burden.

REUSABLE BEHAVIOR

Mixins, extensions and extension types

Three tools that look similar at first glance solve very different problems: implementation reuse, API convenience and zero-allocation static abstraction.

7. Mixins: reuse behavior across class hierarchies

```
mixin Diagnostic on Object {
  String get debugLabel => runtimeType.toString();
  void log(String message) => print('[$debugLabel] $message');
}

class Cache with Diagnostic {
  void clear() => log('cache cleared');
}
```

A mixin injects member implementations into multiple class hierarchies. Use `on` when the mixin assumes a superclass capability. Prefer composition when the behavior has independent state, lifecycle or replaceable dependencies.

Mixin is a fit	Composition is a fit
Small orthogonal behavior; the host type semantically “has” that capability.	The collaborator has its own identity, configuration, lifecycle or should be replaced in tests.

8. Extension methods: convenience without ownership

```
extension DurationFormatting on Duration {
  String get compact => '${inMinutes}m ${inSeconds % 60}s';
}

print(const Duration(seconds: 95).compact);
```

Extensions add statically resolved members without changing the original type. They are excellent for presentation helpers and domain-specific convenience, but they cannot enforce invariants on the original object.

> **Static dispatch matters**
>
> An extension member is selected from the static type of the receiver. If the variable is dynamic, extension methods are not the fallback mechanism you might expect.

9. Extension types: static wrappers with a different interface

INTENTIONAL TYPE SURFACE

```
extension type UserId(String value) {
  bool get isValid => value.isNotEmpty;
}

UserId parseUserId(String raw) => UserId(raw.trim());
```

Extension types create a compile-time abstraction over a representation type. They are especially useful for interop and for giving a primitive representation a narrower static interface without ordinary wrapper allocation.

Useful when	Not full encapsulation
Representation cost matters and a static-only API boundary is enough.	If you need robust runtime abstraction, hidden mutable representation or behavioral identity, use a class.

10. Constructors as invariant gates

Advanced constructor design is less about knowing every syntax form and more about controlling which object states can exist.

```
final class Percentage {
  const Percentage._(this.value);
  final double value;

  factory Percentage.from(double raw) {
    if (raw < 0 || raw > 100) {
      throw RangeError.range(raw, 0, 100, 'raw');
    }
    return Percentage._(raw);
  }
}
```

Use the right constructor tool

- Generative constructors create and initialize a new instance.
- const constructors enable canonical compile-time constants when the object is immutable.
- factory constructors can validate, cache, redirect or return a subtype.
- Initializer lists compute fields and validate before the constructor body.
- Redirecting and super parameters can reduce boilerplate in hierarchies.

11. Const, identity and value semantics

Const is both a language feature and a design hint: the object is deeply configured at construction time. Canonical constants can also reduce duplicate allocations for identical compile-time values.

```
final class Point {
  const Point(this.x, this.y);
  final int x, y;

  @override
  bool operator ==(Object other) =>
      other is Point && other.x == x && other.y == y;

  @override
  int get hashCode => Object.hash(x, y);
}
```

> **Equality contract**
>
> If you override `==`, also provide a compatible `hashCode`. Mutable fields used by hash-based collections are a common source of subtle bugs.

Advanced null safety: late, required and promotion boundaries

Null safety is most effective when object initialization rules are explicit. `required` pushes missing information to the call site, while `late` postpones initialization checks until runtime. Use `late` when lifecycle guarantees exist but the compiler cannot see them - not as a shortcut around good initialization design.

INITIALIZATION CONTRACT

```
final class Session {
  Session({required this.token});
  final String token;

  late final DateTime connectedAt;
  void connect() { connectedAt = DateTime.now(); }
}
```

Use late final when	Avoid late when

The value is assigned once after construction and the lifecycle guarantees access happens later.	The value might genuinely be absent. Model optionality with `T?` instead of turning absence into a runtime initialization error.

Promotion has boundaries

The analyzer is conservative around mutable fields, getters and code that could change values between checks. Promote locals or use patterns when you need a stable narrowed value.

Operators and callable objects: powerful, but semantic

DOMAIN-FRIENDLY SYNTAX

```
final class Vector {
  const Vector(this.x, this.y);
  final double x, y;

  Vector operator +(Vector other) =>
      Vector(x + other.x, y + other.y);
}

final class Validator {
  bool call(String value) => value.trim().isNotEmpty;
}

final isValid = Validator();
print(isValid('dart'));
```

Operator overloads and `call()` can make APIs read naturally when the domain already has that meaning. Avoid surprising behavior: `+` should feel additive, comparison should obey ordering expectations, and a callable object should act like a clear function-shaped collaborator.

Primary constructors and concise model declarations

Modern Dart also supports primary-constructor syntax for class-like declarations. It can reduce boilerplate for simple data-oriented types, but the same design rule applies: concise syntax should not hide invariants or lifecycle logic.

CONCISE CONSTRUCTION

```
class Point(var int x, var int y);

final class ApiError(final int code, final String message) {
  String get label => '$code: $message';
}
```

Use with restraint

Primary constructors are most compelling for small types whose fields and construction contract are obvious. For complex initialization, named generative or factory constructors are often clearer.

ASYNCHRONY

Async Dart without UI jank or race conditions

The hard part is rarely `await`. The hard part is ownership, ordering, cancellation strategy, error propagation and deciding when work needs a separate isolate.

12. Futures and the event loop

An async function runs synchronously until its first suspension point. `await` pauses that async function until the awaited future completes; it does not freeze the entire Dart isolate. Meanwhile, the isolate's event loop can process other queued work.

```
Future<User> loadUser(String id) async {
  final response = await client.get('/users/$id');
  return User.fromJson(response);
}
```

Sequential await

Use it when step B depends on step A or when order is semantically required.

Concurrent futures

Start independent operations first, then await them together to reduce total latency.

INDEPENDENT WORK

```
final profileFuture = loadProfile();
final prefsFuture = loadPreferences();
final (profile, prefs) = (await profileFuture, await prefsFuture);
```

> **Race-condition mindset**
>
> Async bugs often come from stale responses overwriting newer state. Track request identity, invalidate old work, or make state transitions explicit.

Event queue vs microtask queue

An isolate processes asynchronous work through its event loop. Microtasks run before the next event, so repeatedly scheduling microtasks can delay timers, I/O callbacks and UI events. Most application code should prefer normal Future/async APIs and use explicit microtasks only when ordering truly requires it.

ORDERING MENTAL MODEL

```
scheduleMicrotask(() => print('microtask'));
Future<void>(() => print('event'));
print('sync');
```

Microtask

Runs after the current synchronous stack but before the next event. Useful for narrow ordering needs.

Event

Timers, I/O and many Future completions enter normal event processing. This keeps the isolate responsive.

> **Do not "optimize" with microtasks**
>
> A long chain of microtasks can starve ordinary events. If you need sustained work, restructure the algorithm or use an isolate.

Timeouts and ownership

A timeout does not necessarily cancel the underlying operation; it only stops waiting after the deadline. Design APIs so callers know whether work can continue in the background and whether late results must be ignored.

DEADLINE

```
final response = await request()
    .timeout(const Duration(seconds: 8));
```

13. Streams as sequences over time

A Future answers once. A Stream represents zero or more asynchronous events. That makes streams a natural fit for sockets, sensors, database change feeds and progressive pipelines.

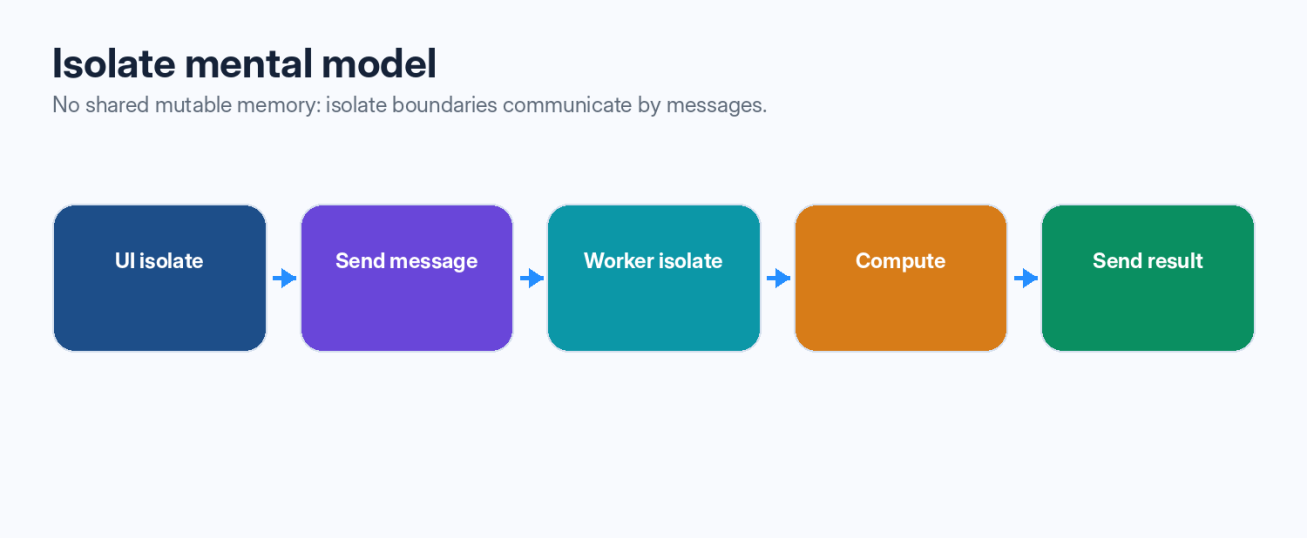
```dart
Stream<int> ticks(int count) async* {
  for (var i = 0; i < count; i++) {
    await Future<void>.delayed(const Duration(milliseconds: 300));
    yield i;
  }
}

await for (final value in ticks(3)) {
  print(value);
}
```

Single-subscription vs broadcast

Single-subscription	Broadcast
Represents one logical sequence. A listener consumes that stream's lifecycle.	Allows multiple listeners and is appropriate for event-like sources. Understand what happens when there are no listeners.

Transform streams close to the data boundary. Parsing, filtering and normalization belong in a pipeline; UI code should ideally receive typed domain events rather than raw bytes or unvalidated maps.

14. Isolates: concurrency without shared mutable memory



All Dart code executes inside isolates. Each isolate has its own memory and event loop. Multiple isolates communicate with messages rather than sharing arbitrary mutable objects.

ONE-SHOT CPU WORK

```dart
final digest = await Isolate.run(() {
  return expensiveDigest(largePayload);
});
```

When an isolate is justified

- CPU-heavy parsing or transformation that causes frame drops.
- Image, compression, cryptographic or data-processing work with measurable latency.
- Independent workers that can communicate using sendable messages.

> **Do not move I/O to an isolate by reflex**
>
> Network and file APIs are already asynchronous. Use another isolate when CPU work blocks the current isolate long enough to hurt responsiveness.

15. Error handling as a typed design problem

Exceptions are useful for exceptional control flow, but production APIs often benefit from distinguishing recoverable domain outcomes from programming errors.

```
sealed class Result<T> {}
final class Ok<T> extends Result<T> { Ok(this.value); final T value; }
final class Err<T> extends Result<T> { Err(this.error); final Object error; }

String message(Result<User> result) => switch (result) {
  Ok(:final value) => 'Hello ${value.name}',
  Err(:final error) => 'Could not load: $error',
};
```

Throw for

Broken preconditions, impossible states, infrastructure failures that the current layer cannot meaningfully resolve.

Return domain outcomes for

Expected alternatives the caller is required to handle, such as validation failure or “not found.”

ARCHITECTURE AND APIS

Advanced Dart in package and application design

Language features pay off when they clarify ownership. This section connects syntax to module boundaries, public contracts and production maintenance.

16. Library privacy and deliberate exports

Dart privacy is library-scoped: identifiers beginning with `_` are private to the library. Use small public entry points and export only the types consumers actually need.

```
// lib/my_package.dart
export 'src/client.dart' show ApiClient, ApiResponse;
export 'src/models.dart' show User;

// Internal helpers remain under lib/src and are not exported.
```

> **A public API is a promise**
>
> Every public class, constructor, field and subtype relationship can become something consumers depend on. Smaller surfaces are easier to evolve.

17. Function types, typedefs and higher-order APIs

```
typedef Decoder<T> = T Function(Map<String, Object?> json);

Future<T> fetch<T>(String path, Decoder<T> decode) async {
  final json = await getJson(path);
  return decode(json);
}
```

Function types are first-class contracts. Use typedefs when the same callback shape represents a meaningful role; do not create aliases that merely rename trivial syntax.

Closures capture variables

A closure can retain variables from its lexical scope. This is powerful for factories and callbacks, but be aware of retaining large object graphs or mutable state longer than intended.

18. Collection power tools

```
final visible = [
  for (final item in items)
    if (item.enabled) ...item.children,
];

Iterable<int> countdown(int from) sync* {
  for (var i = from; i >= 0; i--) yield i;
}
```

Collection-if, collection-for and spread operators let you build immutable-looking values declaratively. Generator functions (`sync*` and `async*`) make lazy sequences readable without allocating the entire result up front.

Prefer Iterable pipelines when	Materialize with toList/toSet when
You can process lazily and avoid intermediate collections.	You need stable repeated iteration, random access, mutation or a snapshot at a specific time.

19. Performance: measure allocations and work, not syntax

- Use `const` where it represents truly immutable configuration.
- Avoid repeated parsing, sorting or mapping inside hot rebuild paths.
- Prefer StringBuffer for large incremental string construction.
- Keep lazy iterables lazy until a concrete collection is required.
- Move demonstrably CPU-heavy work off the UI isolate.
- Benchmark realistic workloads; micro-optimizations can obscure code without moving user-perceived latency.

> **Performance hierarchy**
>
> Fix algorithmic complexity and unnecessary work before chasing tiny allocation differences. Readability that prevents duplicated work is often a performance feature.

20. Static analysis and lint-driven design

The analyzer is part of the language experience, not a separate cleanup tool. Strong annotations, exhaustive switches and package lints can convert runtime uncertainty into editor feedback.

EXAMPLE DIRECTION

```
analysis_options.yaml

linter:
  rules:
    - avoid_dynamic_calls
    - prefer_final_locals
    - unnecessary_lambdas
    - use_super_parameters
```

Choose rules that support your team’s architecture. A lint that creates noise will be disabled; a small focused ruleset that catches real mistakes is more valuable than maximal strictness.

Metadata, annotations and generated code

Annotations attach compile-time metadata to declarations. Frameworks and code generators can use that metadata to generate serializers, routing tables, dependency wiring or immutable models without relying on runtime reflection in Flutter.

CUSTOM METADATA

```
class RouteName {
  const RouteName(this.path);
  final String path;
}
```

```
@RouteName('/profile')
final class ProfilePageModel {}
```

Good generator boundary	Risky generator boundary
Generated files are deterministic implementation detail; handwritten code owns the domain contract.	Business logic exists only in generated output or developers must manually edit generated files.

> **Reflection is not the default Flutter tool**
>
> Dart has platform-specific reflection capabilities, but Flutter app architectures usually prefer static code generation and explicit APIs for predictable tree shaking and deployment.

Advanced testing: prove contracts, not syntax

- Test sealed-state transitions as behavior: input event → expected state family member.
- Test generic APIs with more than one concrete type so accidental assumptions are exposed.
- For stream code, verify event order, completion and error behavior.
- For async race fixes, simulate slow-old and fast-new requests and prove stale results are ignored.
- For isolate boundaries, test the pure computation independently before testing message plumbing.

CONCURRENCY REGRESSION TEST

```
test('newer request wins', () async {
  // Arrange two controllable futures.
  // Complete the newer request first, then the older one.
  // Assert state still represents the newer result.
});
```

CASE STUDY

Putting advanced Dart together: a typed data pipeline

This miniature architecture combines generic decoding, sealed outcomes, pattern matching and isolate-based CPU work. The value is not any single feature; it is how the features reinforce one another.

CORE PIPELINE

```
typedef JsonMap = Map<String, Object?>;
typedef Decoder<T> = T Function(JsonMap json);

sealed class FetchResult<T> {}
final class Success<T> extends FetchResult<T> {
  Success(this.value);
  final T value;
}
final class Failure<T> extends FetchResult<T> {
  Failure(this.error);
  final Object error;
}

Future<FetchResult<T>> fetchAndDecode<T>(
  Uri uri,
  Decoder<T> decoder,
) async {
  try {
    final raw = await fetchJson(uri);
    final value = await Isolate.run(() => decoder(raw));
    return Success(value);
  } catch (error) {
    return Failure(error);
  }
}
```

Consume the result exhaustively

```
final result = await fetchAndDecode<User>(uri, User.fromJson);

final label = switch (result) {
  Success(:final value) => value.name,
  Failure(:final error) => 'Failed: $error',
};
```

Why this architecture scales

- The decoder keeps parsing generic but type-safe.
- The sealed result forces callers to acknowledge success and failure.
- The switch is exhaustive and destructures the state directly.
- CPU-heavy decoding can move to an isolate without changing the public result type.
- No dynamic map escapes into the presentation layer.

21. Advanced Dart code-review checklist

Review area	Question
Types	Are unknown inputs narrowed quickly? Is dynamic contained? Do generic parameters preserve information?
States	Can impossible combinations be represented? Would a sealed family or enum make transitions clearer?
APIs	Is the public surface intentionally extensible? Are class modifiers and exports appropriate?
Async	Could stale work overwrite current state? Are independent operations unnecessarily sequential?
Streams	Is the stream lifecycle clear? Is the correct single-subscription/broadcast model used?
CPU work	Is frame-blocking computation measured? Would Isolate.run simplify it?
Errors	Are expected domain outcomes distinguishable from exceptional failures?
Performance	Is expensive work repeated? Are lazy operations materialized too early?

22. Practice challenges

Challenge A — state model

Replace three booleans (`loading`, `hasData`, `hasError`) with a sealed state hierarchy and an exhaustive switch.

Challenge B — generic cache

Create `Cache<K, V>` with a bounded capacity and a typed loader callback.

Challenge C — pattern parser

Use map/list/object patterns to parse three known response shapes without manual casts.

Challenge D — isolate threshold

Time a CPU-heavy parser on the main isolate, then move it to `Isolate.run()` and compare responsiveness.

Challenge E — API modifiers

Design a small package with one interface intended for

Challenge F — stream pipeline

Create an async generator, transform the events into

consumers, one base implementation family and one final concrete type.	typed domain objects and handle errors in one place.

23. Advanced Dart quick reference

Need	Reach for
Multi-value local result	Record
Shape-based branching	Patterns + switch expression
Closed state family	sealed + final subtypes
Contract without external extension	interface class
Implementation inheritance contract	base class
Closed public concrete type	final class
Reusable member implementation	mixin
Convenience on foreign type	extension
Static wrapper over representation	extension type
One async result	Future
Many async events	Stream
CPU concurrency	Isolate.run / isolate APIs
Typed reusable algorithm	Generics
Lazy sequence	sync* / async*

Common advanced-Dart anti-patterns

dynamic everywhere

Short-term convenience becomes runtime uncertainty. Narrow unknown values at boundaries.

Inheritance by default

A subclass relationship is a permanent API promise. Prefer composition unless substitutability is real.

Sealed type with dozens of states

A closed family helps only when states represent a coherent domain. Split unrelated state machines.

Isolates for every async task

Isolates solve CPU contention, not ordinary network latency. Measure first.

Patterns as code golf

A giant nested pattern can be harder to read than two named steps. Extract helpers when the shape obscures intent.

Generic abstraction too early

Wait until two or more use cases reveal the actual shared relationship.

> **Senior-level rule**
>
> Advanced Dart is not the maximum number of language features per file. It is the minimum set of features that makes the contract obvious and the failure modes controlled.

Official references

This guide is aligned with the current Dart language documentation. For exact language rules and evolving features, use the official Dart documentation as the source of truth.

Language overview: https://dart.dev/language
Patterns: https://dart.dev/language/patterns
Records: https://dart.dev/language/records
Class modifiers: https://dart.dev/language/class-modifiers
Generics: https://dart.dev/language/generics
Mixins: https://dart.dev/language/mixins
Extension types: https://dart.dev/language/extension-types
Asynchronous programming: https://dart.dev/language/async
Concurrency and isolates: https://dart.dev/language/concurrency
Isolate examples: https://dart.dev/language/isolates

> **One final rule**
>
> Advanced language features should make the code easier to reason about. If a clever construct hides intent from the next developer, simplify it.

Advanced Dart glossary

Term	Meaning
Promotion	Static narrowing of a value after control-flow analysis proves a more specific type.
Exhaustiveness	A switch covers every possible case of a closed type or pattern space.
Variance	How subtype relationships behave when types are wrapped by a generic type.
Canonical const	Equivalent compile-time constants may share the same canonical instance.
Tear-off	A function or method referenced as a first-class function value without invoking it.
Broadcast stream	A stream designed for multiple simultaneous listeners.
Isolate	An independent Dart execution context with its own memory and event loop.
Static extension	Members selected using the receiver's compile-time type rather than runtime dispatch.
Guard	A boolean condition attached to a pattern case after structural matching succeeds.
Public surface	The declarations and extension points consumers can depend on across a library boundary.

FLUTTERFEVER

BUILD WITH DART

LIKE AN ENGINEER.

Not just syntax. Learn the language decisions that make Flutter code safer, clearer, faster and easier to evolve.

01 **EXPRESSIVE TYPES**

Generics, records, patterns, sealed hierarchies and extension types.

02 **CONCURRENCY**

Futures, Streams, event loop thinking and Isolates for real workloads.

03 **API DESIGN**

Small, predictable, testable Dart APIs that scale with your Flutter app.

KEEP LEARNING

More practical Flutter & Dart guides at

flutterfever.com

Scan the QR to visit FlutterFever

Practical guides for modern Flutter developers.