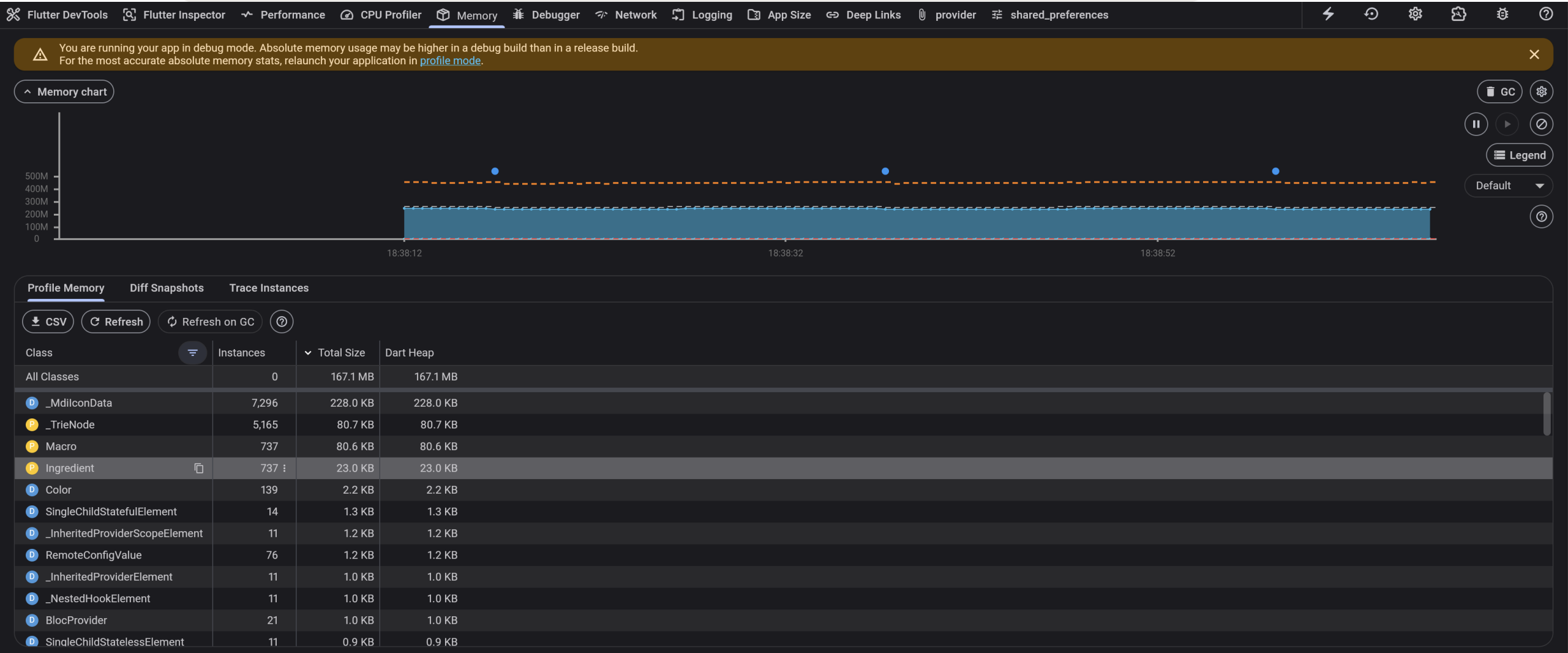
Mobile and Embedded Computing

Lecture 4. Debugging & Tools, State management techniques intro, Equatable

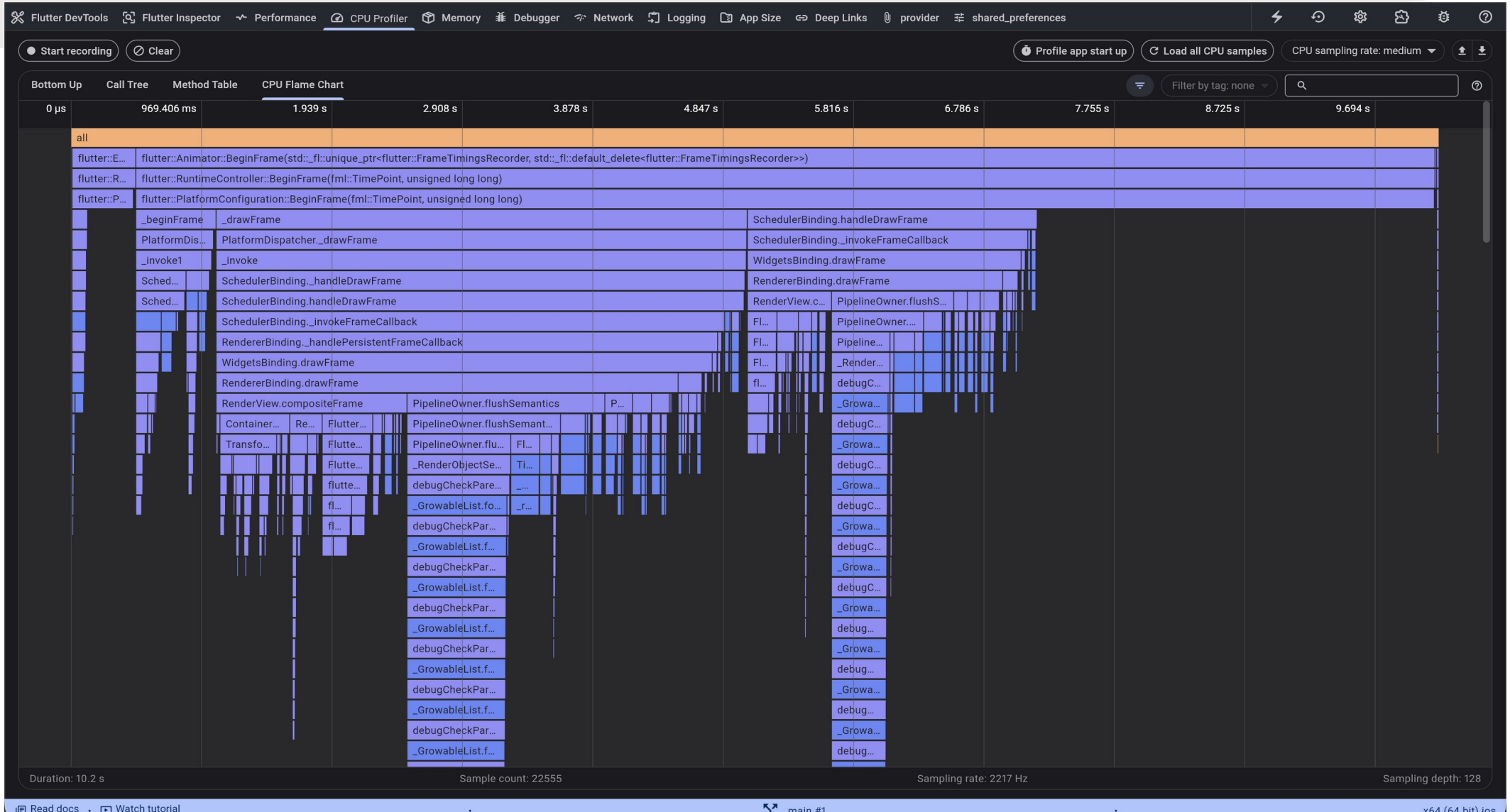
Flutter Dev Tools

- Usually on http://127.0.0.1:9100/home
- You will need this URL that is printed in the console:

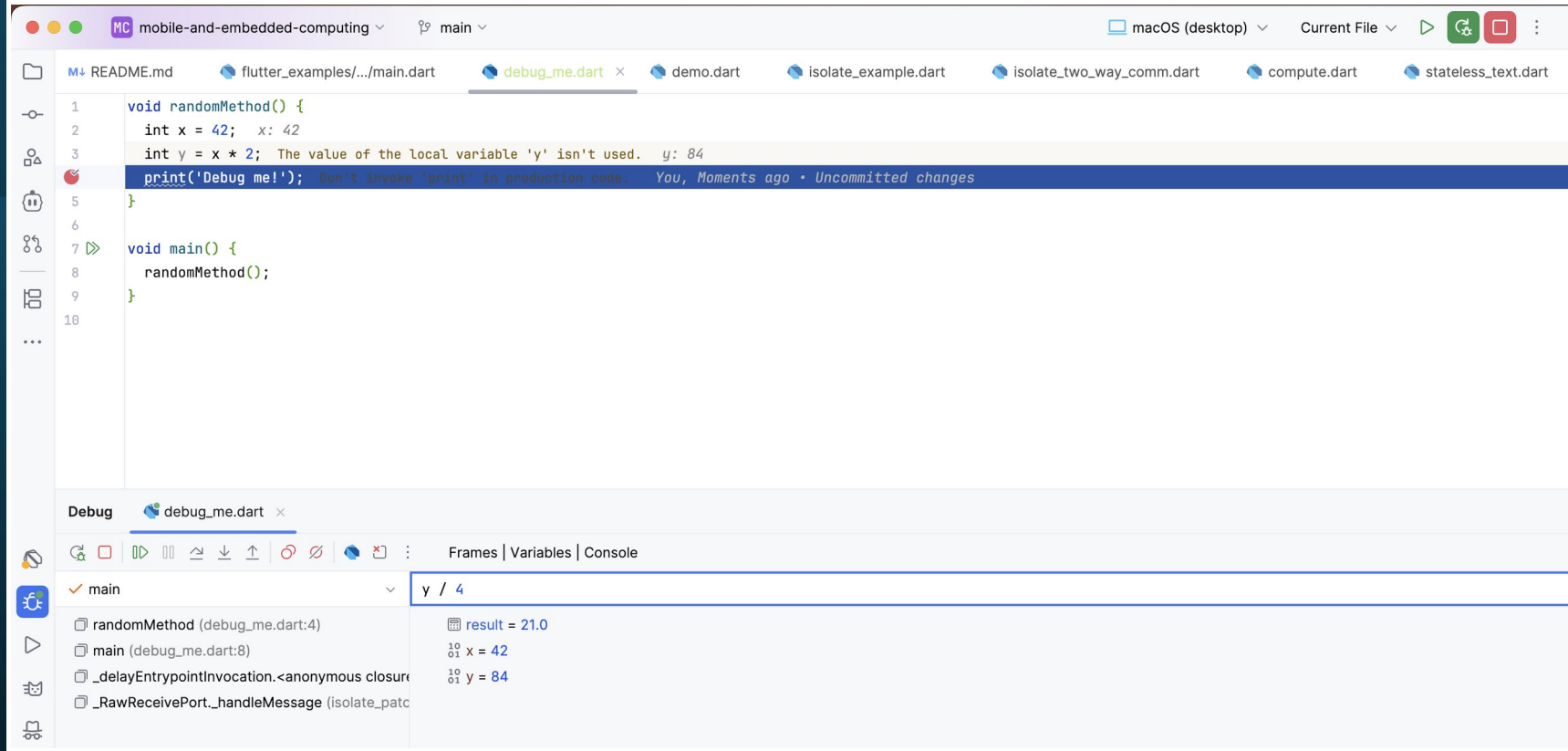
Flutter Dev Tools



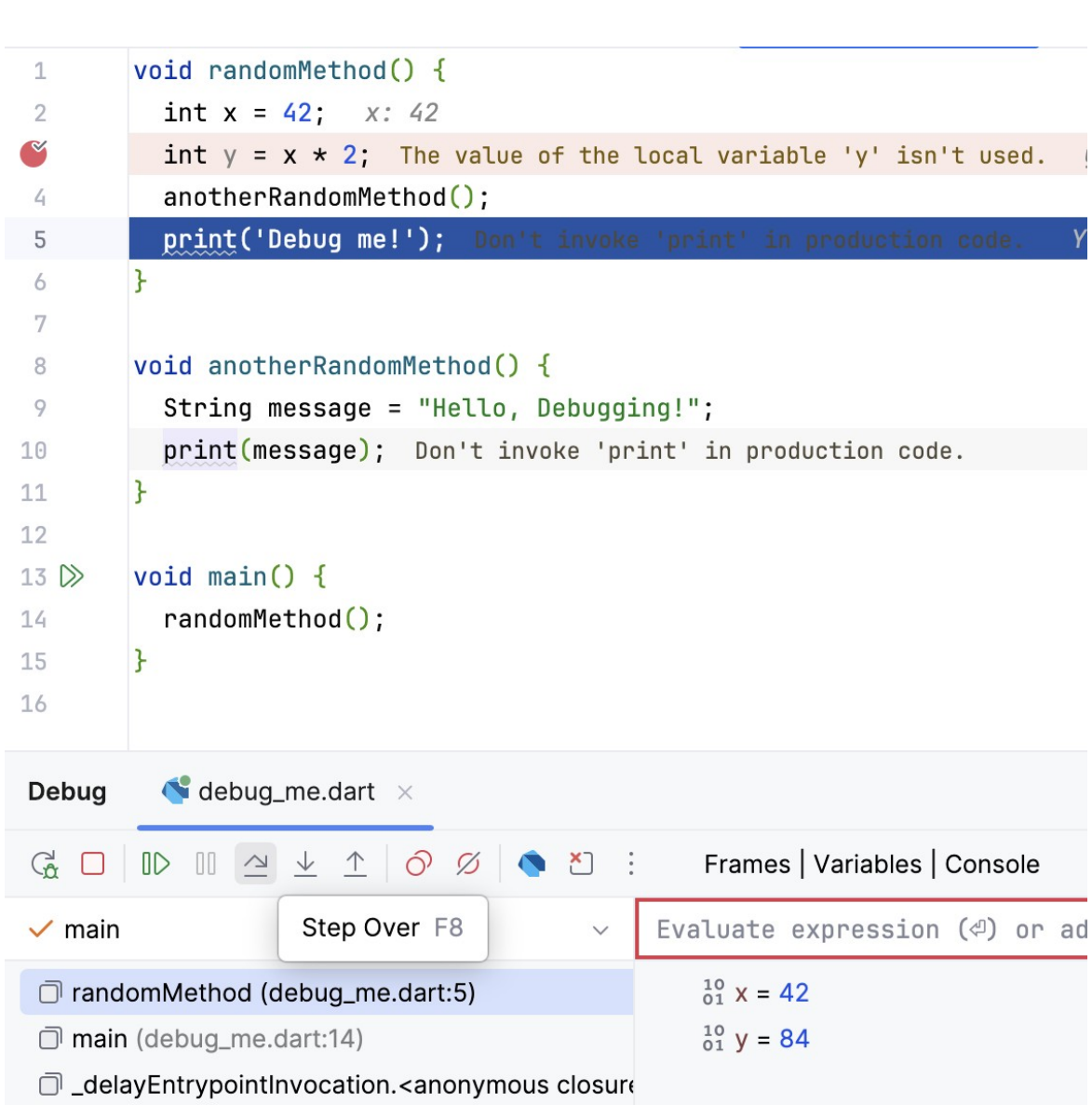
Flutter Dev Tools



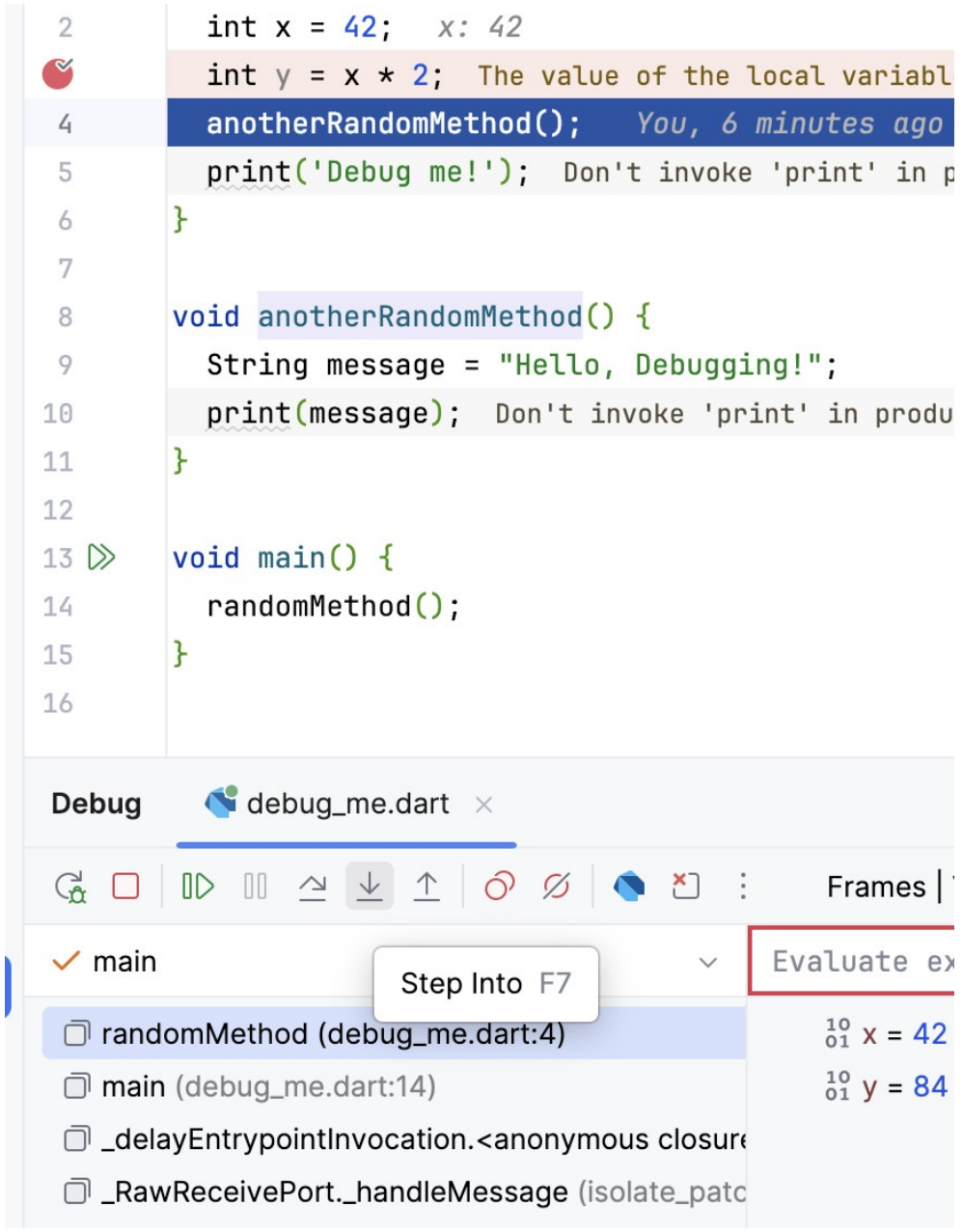
Using a debugger



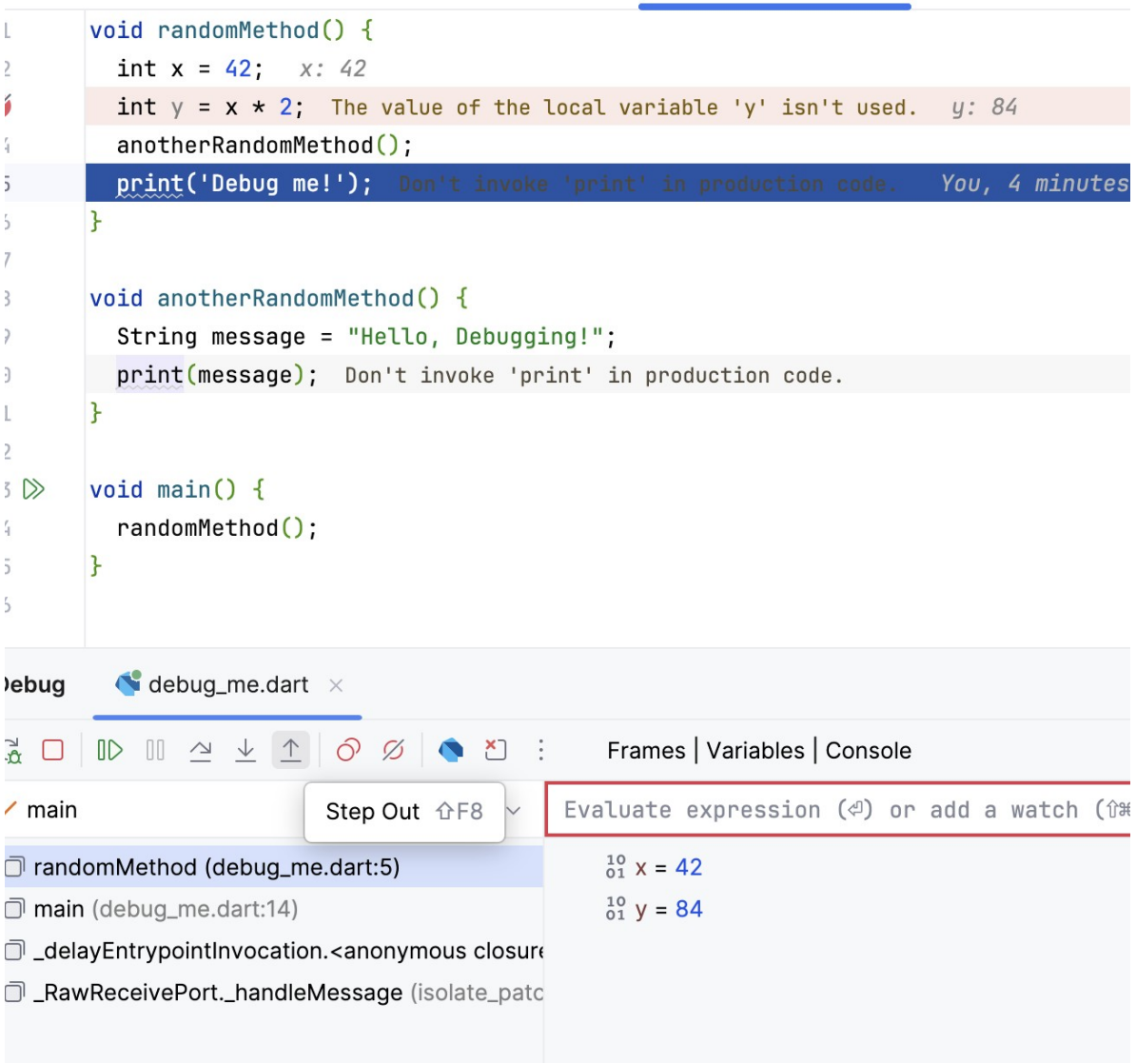
Step Over – Moves line by line without entering in any method



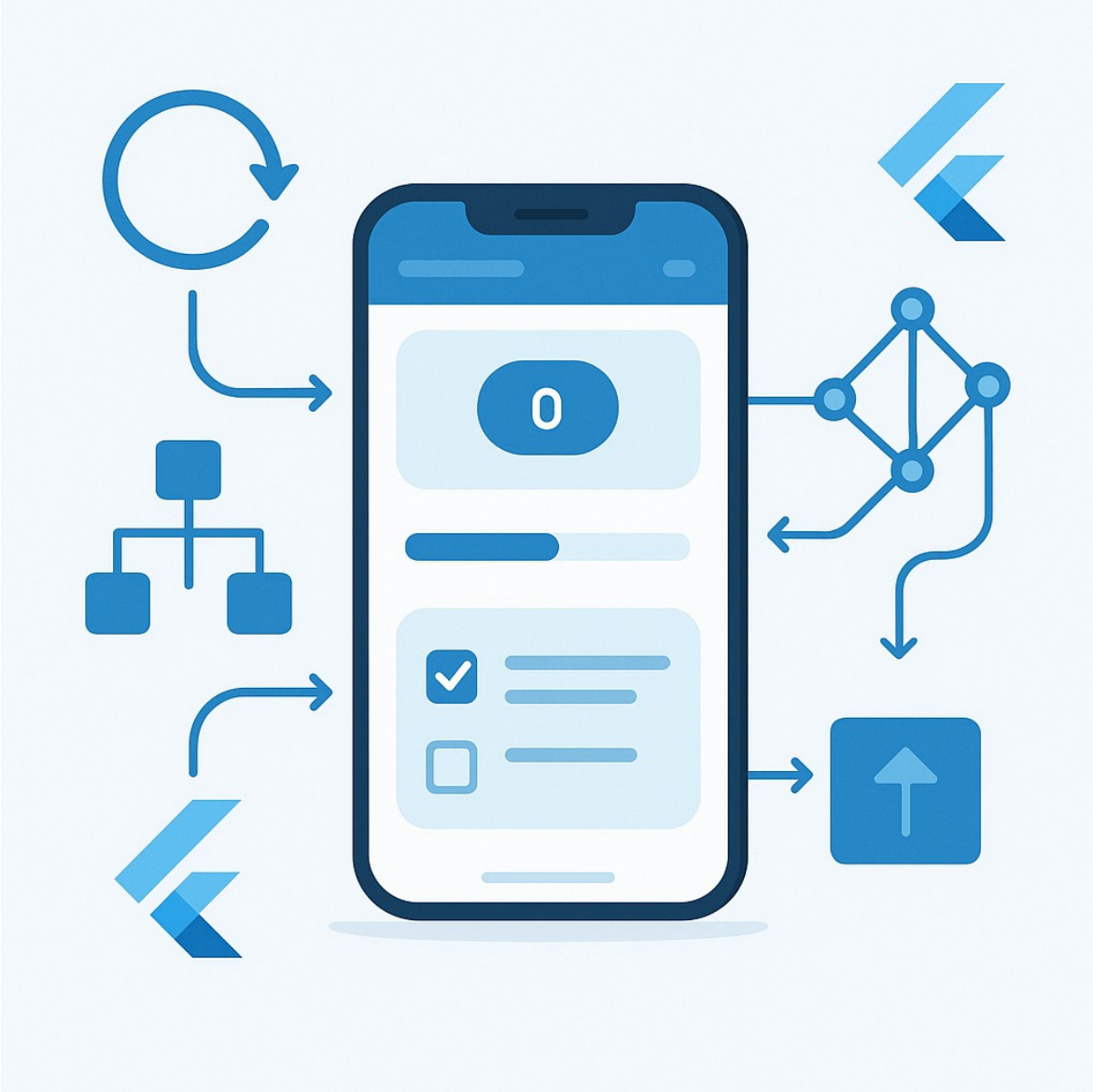
Step Into – If applied on a method, the debugger will move into that method



Step Out – Goes out of a method back to the previous one



State management techniques



Introduction to State

State is the data that changes over time in your app.

Whenever that data changes, your UI must update to reflect it.

Example:

The number on a counter app.

The text in a form field.

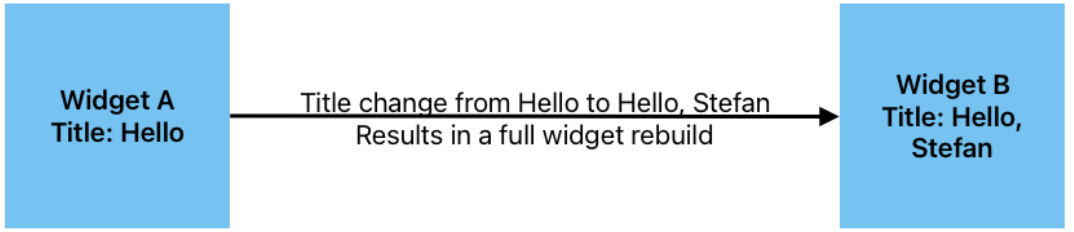
Whether a user is logged in or not.

In Flutter, widgets themselves are immutable once created, they can’t change.

So when the state changes, Flutter rebuilds widgets with the new data.

What is Immutability?

- “Immutable” means unchangeable, once an object is created, it cannot be modified.
- Instead of changing the object, Flutter creates a new instance when something changes.



Why are widgets immutable ?

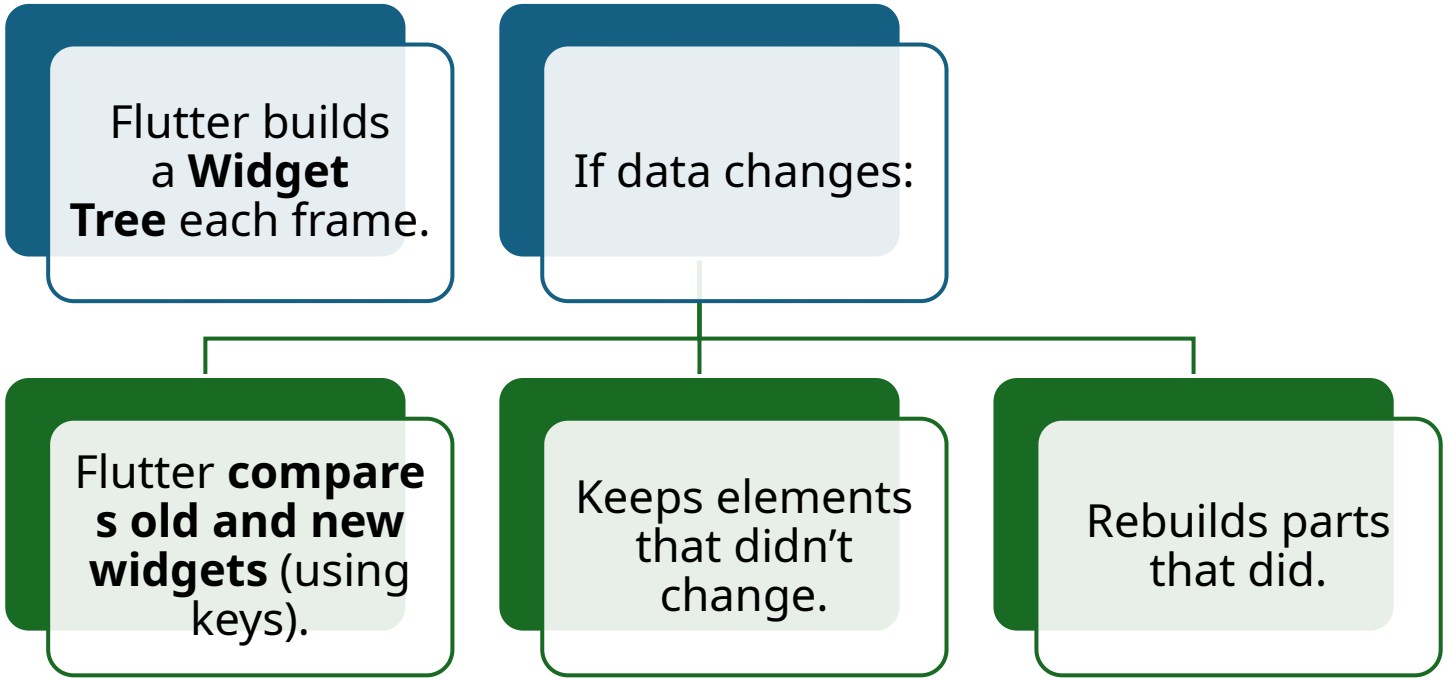
Predictability: UI doesn’t mutate unexpectedly .

Performance: Flutter can efficiently rebuild only what changes.

Simplicity: No complex change tracking; just rebuild.

Debugging ease: Each UI frame is a pure function of its state.

Widget Lifecycle and Rebuilds



Type	Description	Example Use
StatelessWidget	Immutable, builds once.	Labels, Icons
StatefulWidget	Has internal state; triggers rebuilds.	Forms, Buttons

- Widgets are blueprints for UI elements, not the actual UI on screen.
- When you “change” something, Flutter just uses a new blueprint to draw the updated UI.

```dart
// Stateless: fixed data
class TitleText extends StatelessWidget {
  final String title;
  const TitleText({super.key, required this.title});

  @override
  Widget build(BuildContext context) => Text(title);
}

// Stateful: can change data
class CounterWidget extends StatefulWidget {
  const CounterWidget({super.key});

  @override
  CounterWidgetState createState() => CounterWidgetState();
}
```

Types of state

Type	Description	Example
Ephemeral (Local) State	State that lives in a single widget.	Toggle switch, counter, checkbox.
App (Global) State	Shared across multiple widgets or screens.	Logged-in user, theme mode, shopping cart.

setState() – The Basics

- The setState() method is the most basic way to manage state in Flutter.
- It’s used inside a StatefulWidget to update local state.
- Logic and UI get mixed together.
- Hard to reuse state in other widgets.
- Doesn’t scale for complex apps.

```dart
int _count = 0;

void _increment() {
  setState(() {
    _count++;
  });
}
```

Lifting State Up

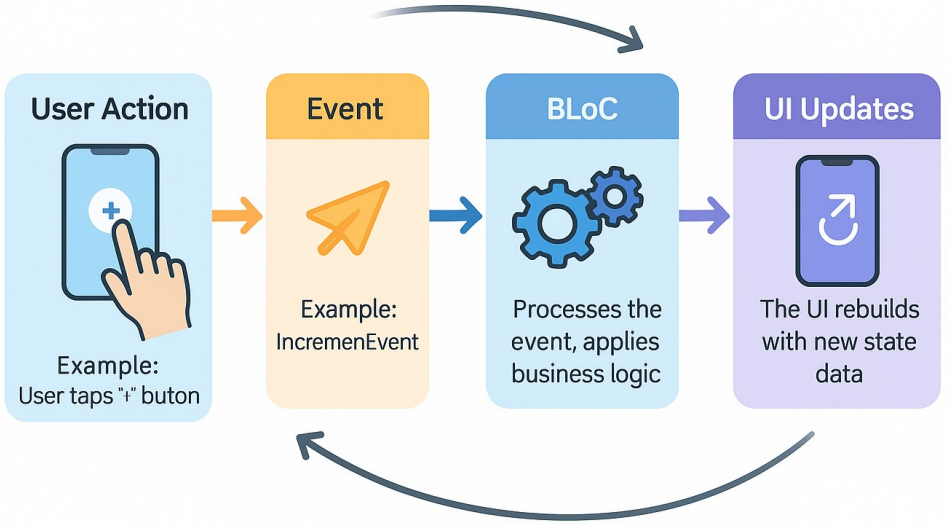
- When two widgets need to share the same data, you can lift state up to their common parent.
- Keeps widgets reusable and independent.
- Promotes clear data flow: Parent → Child (data) and Child → Parent (events).
- If you lift state up too far, it becomes hard to manage. Passing data through many layers (“prop drilling”).

```dart
class ParentWidget extends StatefulWidget {
  const ParentWidget({super.key});
   You, Moments ago • Uncommitted changes
  @override
  ParentWidgetState createState() => ParentWidgetState(
}

class ParentWidgetState extends State<ParentWidget> {
  bool _active = false;

  void _handleTapboxChanged(bool newValue) {  Typo: In w
    setState(() {
      _active = newValue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Container(  Unnecessary instance of 'Container'
      child: ChildWidget(
        active: _active,
        onChanged: _handleTapboxChanged,
      ),  ChildWidget
    );  Container
  }
}
```

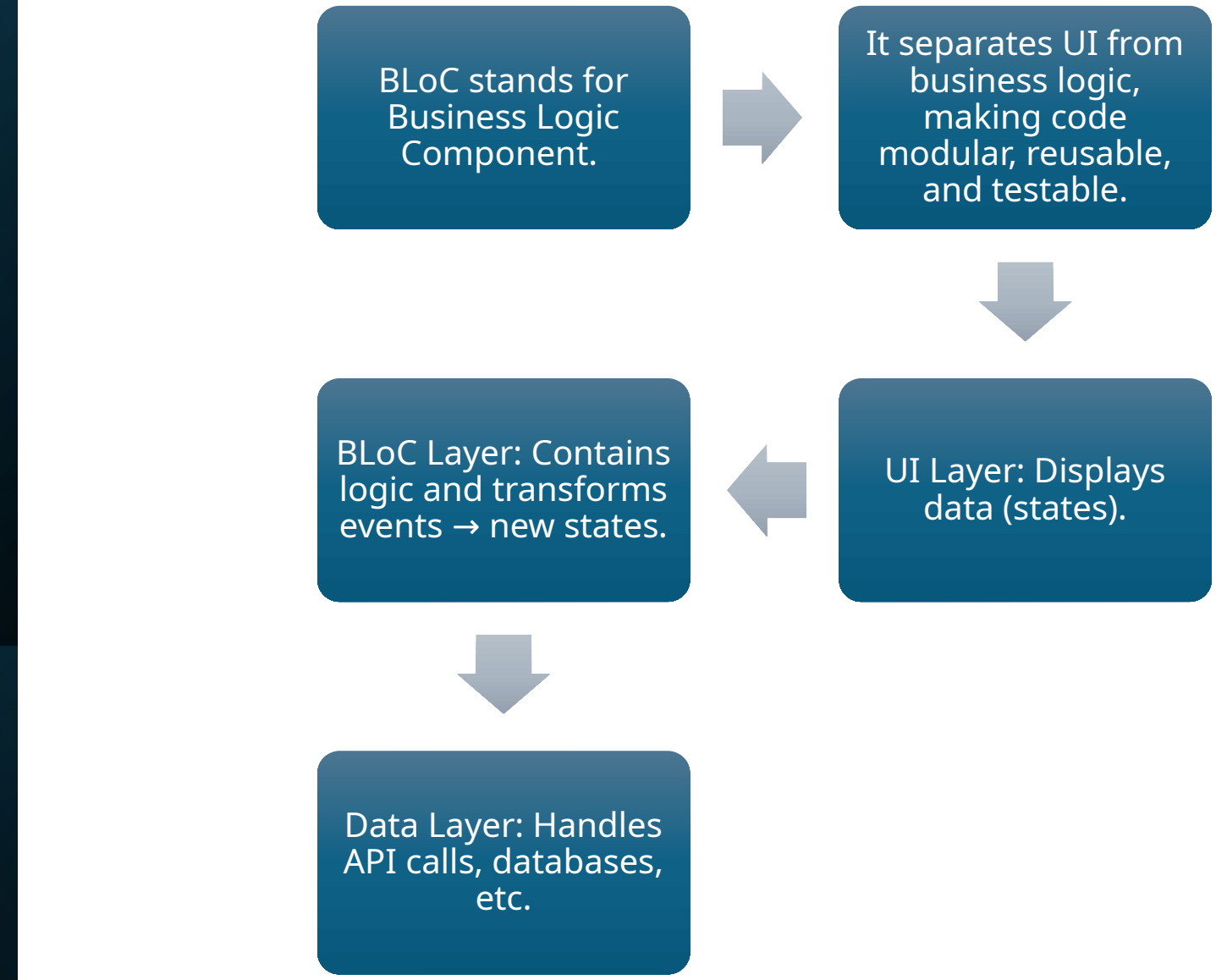
```dart
class ChildWidget extends StatelessWidget {
  final bool active;
  final ValueChanged<bool> onChanged;

  const ChildWidget({super.key, required this.active, requi

  void _handleTap() {
    onChanged(!active);
  }

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: _handleTap,
      child: Container(
        width: 150,
        height: 150,
        color: active ? Colors.green : Colors.grey,
        child: Center(
          child: Text(
            active ? 'Active' : 'Inactive',
            style: TextStyle(fontSize: 24, color: Colors.wh
          ),  Text
        ),  Center
      ),  Container
    );  GestureDetector
  }
}
```

BLoC (& Cubit pattern)

- Separates UI from business logic with a stream-based approach
- BLoC: Uses events and states via Stream and Sink.
- Cubit: A simplified BLoC that emits states directly.



BLoC


BLoC stands for Business Logic Component.
It separates UI from business logic, making code modular, reusable, and testable.
UI Layer: Displays data (states).
BLoC Layer: Contains logic and transforms events → new states.
Data Layer: Handles API calls, databases, etc.

Cubit

Cubit is a simpler version of BLoC.

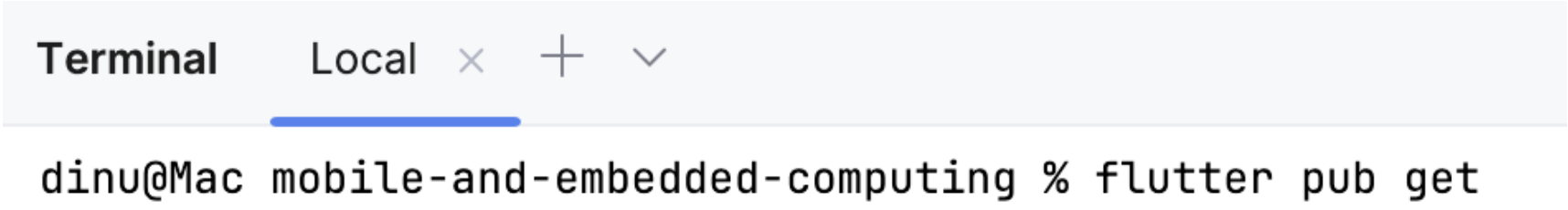
Cubit: Emits new states directly (no separate events).

BLoC: Works with **events** and **states**, offering more structure.

Bloc & Cubit Required dependencies

```
dependencies:
  flutter_bloc: ^9.1.1
  equatable: ^2.0.7
```

- You can add new dependencies in pubspec.yaml
- Then you will need to run flutter pub get to fetch them
- Upgrading dependencies is done using flutter pub upgrade



Equatable

- In Flutter BLoC, the UI only rebuilds when the state changes. But how does Flutter know if the new state is different from the previous one?
- Without Equatable, Dart's default equality check compares object references, not values. So even if two CounterState(1) objects have the same count, Flutter thinks they're different, causing unnecessary rebuilds. Equatable fixes this by comparing objects by value.

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

// without equatable - and Flutter won't know
// that the state has changed
class CounterStateBad {
  final int count;
  CounterStateBad(this.count);
}

// with equatable - Flutter knows that the state has changed
class CounterState extends Equatable {
  final int count;
  const CounterState(this.count);

  @override
  List<Object> get props => [count];
}   You, Moments ago • Uncommitted changes
```

Best Practices and Recommendations

KEEP UI (WIDGETS) AND LOGIC SEPARATE.

USE IMMUTABLE DATA MODELS WHEN POSSIBLE.

DISPOSE STREAMS AND CONTROLLERS PROPERLY.

START SIMPLE (SETSTATE → RIVERPOD/BLOC).

PICK WHAT FITS YOUR PROJECT.