Flutter

A study material for the students of GLS University

Prepared by : Ms. Harshita Maheshwari

Subject : Computer Networks

What is Flutter?

Flutter is an open-source UI (User Interface) framework developed by Google in 2015.

In simple words:

Flutter helps us build mobile, web, and desktop apps using a single codebase.

What is Flutter

Traditional Approach (Problem) -

- Android → Java/Kotlin
- iOS → Swift/Objective-C
- Web → HTML/CSS/JS

Different languages, different teams, more cost & time

Flutter Approach (Solution) -

- One language (Dart)
- One codebase
- Multiple platforms

What is a Framework?

A framework is a pre-built structure that provides tools, libraries, and rules to help developers build software faster and in a standard way.

Without a framework:

- You write everything from zero
- More time & more errors
- No standard structure

With a framework:

- Faster development
- Less code
- Better organization
- Built-in best practices

Why Flutter

- **Cross-platform development**
- **Hot Reload**
- **Rich UI**
- **High Performance -** Flutter apps are compiled into native machine code. No bridge like hybrid apps.

native machine code means: Code that runs directly on the device, without translation at runtime.

React native vs flutter

Feature	**React Native**	**Flutter**
Developed by	Meta	Google
Language	JavaScript	Dart
Rendering	Native UI components	Skia engine
Performance	Good	Very high
UI Consistency	Platform dependent	Same across platforms
Platform Support	Mobile only	Mobile, Web, Desktop
Hot Reload	Yes	Yes (faster)
App Size	Smaller	Larger

When to use What?

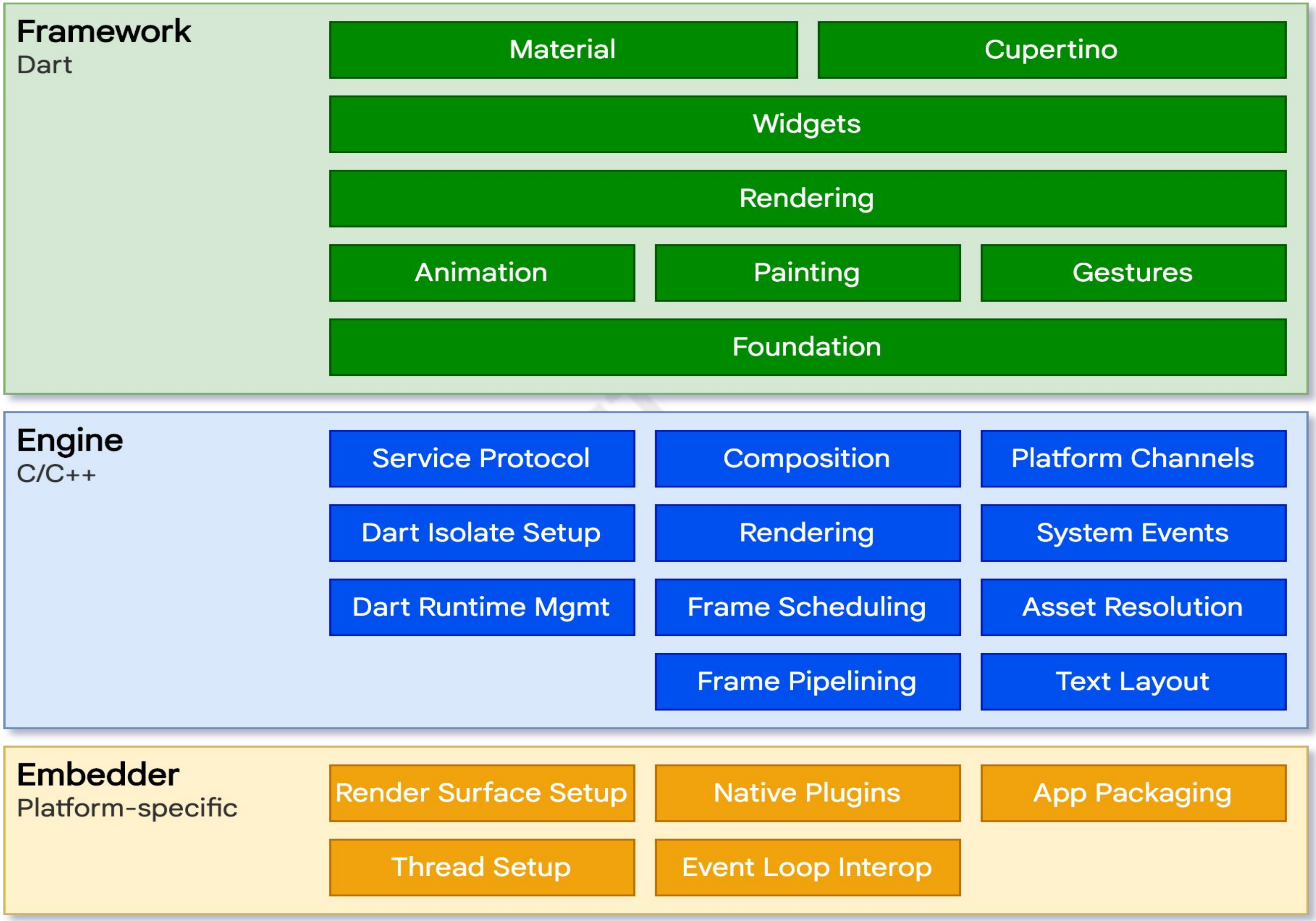
Choose React Native if:

- You know JavaScript & React
- App is simple to medium complexity
- Need quick development

Choose Flutter if:

- You want high performance
- Complex UI & animations
- One codebase for many platforms

Flutter Architecture



Framework
Dart

Material
Cupertino
Widgets
Rendering
Animation
Painting
Gestures
Foundation

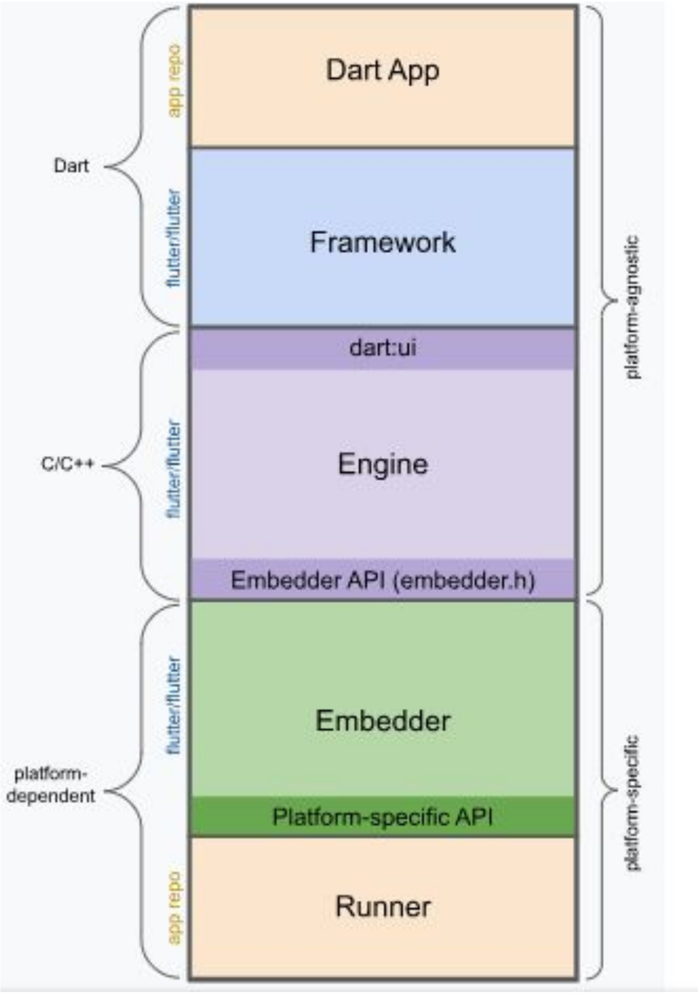
Engine
C/C++

Service Protocol
Composition
Platform Channels
Dart Isolate Setup
Rendering
System Events
Dart Runtime Mgmt
Frame Scheduling
Asset Resolution
Frame Pipelining
Text Layout

Embedder
Platform-specific

Render Surface Setup
Native Plugins
App Packaging
Thread Setup
Event Loop Interop

Flutter Architecture



1. Flutter App (What Developers Write) -

This is your code.
Written in Dart

Contains:

- UI (widgets)
- Business logic
- State management

In Flutter, **UI = Code** (no XML, no separate layout files).

Flutter Architecture

2. Flutter Framework (Dart Layer) -

The Flutter Framework is a set of Dart libraries that help you build apps easily.

It sits between your app and the engine.

Main Responsibilities -

- Widget system
- Layout system
- Animation
- Gestures
- Themes
- Navigation
- State management

Framework Sub-Layers

A. Widget Layer -

Widgets are the building blocks of Flutter UI.
Examples: Text, Container, Row, Column, Button

Key concepts:
Everything is a widget
Widgets are immutable
UI is rebuilt when state changes

Flutter uses a reactive UI model: Change data → UI updates automatically

Framework Sub-Layers

B. Element Layer -

- Acts as a bridge between widgets and render objects
- Manages: Widget lifecycle, Tree structure
- Widgets → Elements → Render Objects

C. Rendering Layer -

- Converts widgets into visual instructions
- Responsible for: Size, Position, Layout
- Creates the Render Tree

D. Foundation & Services Layer -

Provides: Animations, Gestures (tap, swipe, drag), Painting, Text handling, Themes

Flutter Architecture

3. Flutter Engine (C++ Core) -

The engine is the heart of Flutter, written in C++.

What the Engine Does -

- Runs Dart code
- Converts UI instructions into pixels
- Handles: Graphics, Text rendering, Accessibility, Input events

- Flutter uses Skia Graphic Engine for drawing.
- JIT (Just-In-Time) → during development (Hot Reload)
- AOT (Ahead-Of-Time) → for release builds

Flutter Architecture

4. Platform Embedder (Platform-Specific Layer) -
Each platform has its own embedder:
Android → Java / Kotlin
iOS → Objective-C / Swift
Windows / Linux / macOS → C++

Responsibilities
App entry point
Event loop
Input handling (touch, keyboard, mouse)
Surface creation for drawing
Plugin integration

Embedder is the bridge between Flutter and OS.

How UI Rendering Actually Works

1. build() method is called
2. Widget tree is created
3. Element tree is updated
4. Render tree calculates layout
5. Engine paints pixels
6. OS displays the frame

MVVM Pattern

MVVM (Model-View-ViewModel) is a proven architectural pattern that separates concerns, improves testability, and scales gracefully as applications grow in complexity.

View Layer - UI widgets that display data and capture user interactions. Views are declarative and reactive, automatically updating when data changes.

ViewModel Layer - Contains business logic and state management. Exposes data streams and commands to Views while remaining UI-agnostic for better testability.

Model Layer - Data layer with repositories and services. Handles data sources, API calls, caching, and persistence logic independent of UI concerns.

Layered Architecture in Flutter Apps

UI Layer - Views and ViewModels working together. Reactive and declarative, responding to state changes automatically. This layer never directly accesses data sources.

Logic Layer - Optional domain/business logic layer containing use cases and business rules. Orchestrates data flow between UI and Data layers while keeping business logic testable.

Data Layer - Repositories and services managing all data sources including REST APIs, GraphQL, local databases, and file systems. Provides clean abstractions to upper layers.

Setting Up Flutter SDK

Flutter SDK is a collection of:

- Flutter framework
- Dart SDK
- Compiler
- CLI tools
- Debugging tools

Steps to Set Up Flutter SDK -

Step 1: Download Flutter SDK
Step 2: Set Environment Variable (PATH)
Step 3: Verify Installation (flutter doctor)

Visit https://docs.flutter.dev/install for the steps for setting up flutter manually and with VS code.

Recommended IDEs:
Android Studio
Visual Studio Code

Required plugins:
Flutter
Dart

Emulator / Device Setup:
Android Emulator
Physical device (USB debugging enabled)

VERIFY FLUTTER INSTALLATION: flutter doctor

Assets in Flutter

- Assets are external resources used in an app:

Ex- Images, Fonts, Audio. JSON files etc.

- Flutter does not automatically detect assets
- Assets must be declared explicitly

Asset Folder Structure (Recommended) -

```
assets/
├── images/
├── icons/
├── fonts/
```

Declaring Assets in pubspec.yaml -

```
flutter:
  assets:
    - assets/images/
    - assets/icons/
```

Using Image Asset in Flutter -

```
Image.asset('assets/images/logo.png')
```

Flutter DevTools

Flutter DevTools is a suite of performance and debugging tools for Flutter apps.

It helps developers:

- Debug UI
- Analyze performance
- Inspect widgets
- Monitor memory

Key Features of Flutter DevTools -

- Widget Inspector
- Performance Tab
- Memory Tab
- Debugger

Dart Programming

Dart is an open-source general-purpose programming language developed by Google. It supports application development on both the client and server side. However, it is widely used for the development of Android apps, iOS apps, IoT(Internet of Things), and web applications using the Flutter Framework.

Why Dart Programming

Fast & Smooth: Dart compiles to native code for speedy performance, ideal for mobile apps.

Easy to Learn: Similar to familiar languages like Java or Javascript, making it approachable for new developers.

Flutter Power: Dart is the heart of Flutter, a popular framework for building beautiful and functional mobile apps.

One Code, Many Places: Develop for mobile, web, and even desktop with a single codebase (primarily with Flutter).

Data Types

Dart supports statically typed and type-inferred variables.

- int
- double
- num
- String
- bool
- List(Array)
- Set
- Map
- Dynamic

Variable and Constant

var (type inference) -

```
var city = "Ahmedabad";   // String
```

final (runtime constant) -

```
final int x = 10;
```

const (compile-time constant) -

```
const double pi = 3.14;
```

Null Safety

Null safety prevents null reference crashes at runtime.

Nullable vs Non-Nullable -

```
String name = "Flutter";   // Non-nullable
String? city;              // Nullable
```

? → variable can hold null

Without ?, null is NOT allowed

late Keyword

```
late String token;
token = getToken();
```

Used when: Variable initialized later but guaranteed before use.

Operators

Category	Operators
Arithmetic Operators	+ , - , * , / , ~/ , %
Relational (Comparison) Operators	== , != , > , < , >= , <=
Logical Operators	`&& , || , !
Assignment Operators	= , += , -= , *= , /= , ~/=
Unary Operators	++ , -- , -
Conditional Operator	?:
Type Test Operators	is , is! , as
Null-Aware Operators	?? , ??= , ?. , !.

Conditional Statements

if Statement -

Syntax -

```
if (condition) {
  // statements
}
```

if–else Statement -

Syntax -

```
if (condition) {
  // true block
} else {
  // false block
}
```

Conditional Statements

if–else if Ladder -

Syntax -

```
if (condition1) { // block 1}
else if (condition2) {// block 2}
else { // default block }
```

switch Statement -

Syntax -

```
switch (expression) {
  case value1:
    // code
    break;
  case value2:
    // code
    break;
  default:
    // code
}
```

Loops

Loops enable you to execute code blocks repeatedly, automating repetitive tasks and processing collections of data with precision and control.

For Loop -
Iterate with a known count. Perfect when you know exactly how many times to repeat.

```
for (int i = 1; i <= 10; i++) {
  print(i);
}
```

Loops

While Loop -
Repeat while a condition remains true. Tests condition before each iteration.

```
int i = 1;
while (i <= 10) {
  print(i);
  i++;
}
```

Do-While Loop -
Execute at least once, then repeat while condition is true. Tests condition after iteration.

```
int i = 1;
do {
 print(i);
 i++;
} while (i <= 10);
```

Loops

for–in Loop -

Used to iterate over collections.

Syntax -

```
for (var item in collection) {
  // code
}

List<String> names = ["Aman", "Riya", "Kunal"];

for (var name in names) {
  print(name);
}
```

Loop Control Statements

Break Statement -
Immediately exit the loop, terminating all remaining iterations regardless of the condition.

Continue Statement -
Skip the current iteration and jump directly to the next one, bypassing remaining code in the loop body.

Function

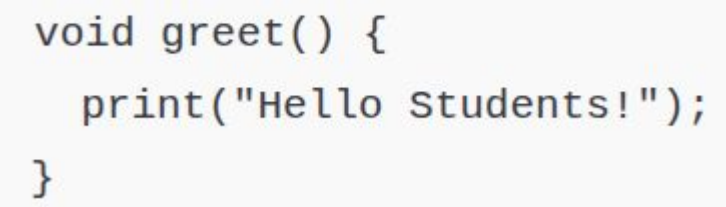
A function is a block of code that:

- Performs a specific task
- Can be reused
- Improves readability and modularity

Example: Instead of writing calculation logic again and again, write it once in a function.

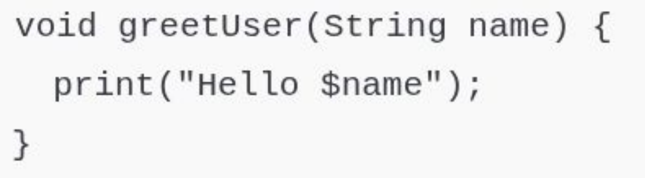
Types -

1. Simple Function (No Return, No Parameter) -



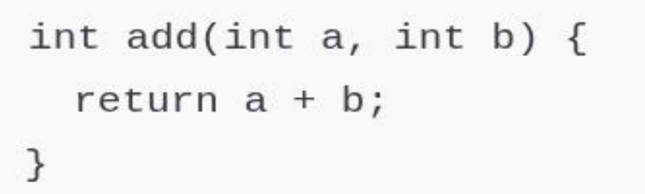
```
void greet() {
  print("Hello Students!");
}
```

2. Function with Parameters -



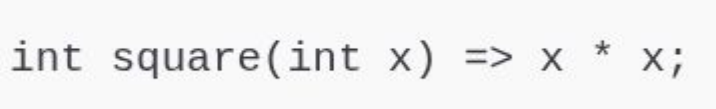
```
void greetUser(String name) {
  print("Hello $name");
}
```

3. Function with Return Value -



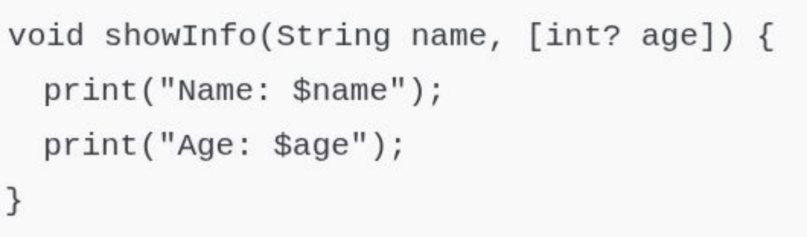
```
int add(int a, int b) {
  return a + b;
}
```

4. Arrow (Fat Arrow) Function -



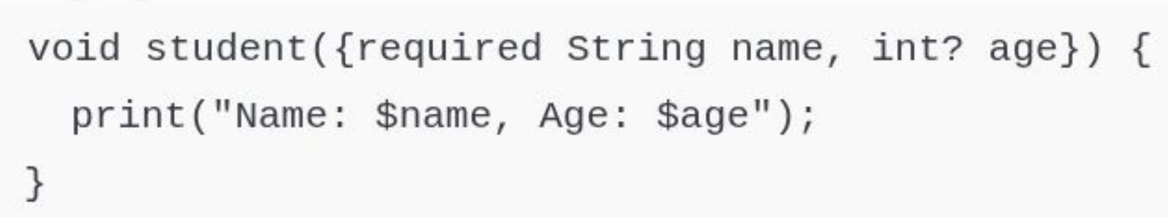
```
int square(int x) => x * x;
```

5. Optional Positional Parameters -



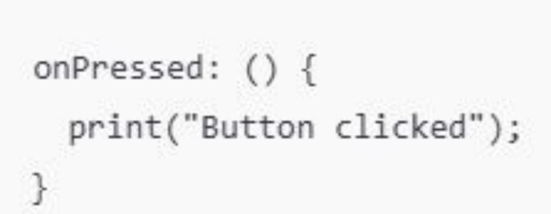
```
void showInfo(String name, [int? age]) {
  print("Name: $name");
  print("Age: $age");
}
```

6. Named Parameters -



```
void student({required String name, int? age}) {
  print("Name: $name, Age: $age");
}
```

7. Anonymous Functions (Callbacks) -



```
onPressed: () {
  print("Button clicked");
}
```

OOP (OBJECT-ORIENTED PROGRAMMING)

What is OOP - OOP is a programming approach based on real-world objects.

Example:
Object → Student
Properties → name, rollNo
Behavior → study(), exam()

Core Concepts -
Class - Blueprint
Object - Instance of class
Encapsulation - Data hiding
Inheritance - Parent → Child
Polymorphism - One name, many forms
Abstraction - Hide implementation

Class and Object

```
class Student {
  String name = "";
  int rollNo = 0;

  void study() {
    print("$name is studying");
  }
}

void main() {
  Student s1 = Student();
  s1.name = "Rahul";
  s1.rollNo = 101;
  s1.study();
}
```

Constructor

Default Constructor -

```
class Student {
  Student() {
    print("Student object created");
  }
}
```

Parameterized Constructor -

```
class Student {
  String name;
  int rollNo;

  Student(this.name, this.rollNo);
}
```

this → refers to current object

Encapsulation (Getters & Setters)

```
class Bank {
  double _balance = 0; // private variable

  double get balance => _balance;

  void deposit(double amount) {
    _balance += amount;
  }
}
```

_ → private

Inheritance

```
class Animal {
  void eat() {
    print("Animal eats");
  }
}

class Dog extends Animal {
  void bark() {
    print("Dog barks");
  }
}

void main() {
  Dog d = Dog();
  d.eat();
  d.bark();
}
```

Method Overriding (Polymorphism)

```
class Animal {
  void sound() {
    print("Animal sound");
  }
}

class Cat extends Animal {
  @override
  void sound() {
    print("Cat meows");
  }
}
```

Abstraction (Abstract Class)

```
abstract class Shape {
  void draw();
}

class Circle extends Shape {
  @override
  void draw() {
    print("Drawing Circle");
  }
}
```

Abstract class cannot be instantiated
Forces child to implement methods

Mixins

```
mixin Logger {
  void log(String msg) => print(msg);
}
```

Used for:

Code reuse

Utility logic

Exception Handling

An **exception** is a **runtime error** that crashes the program if not handled.

Example: Divide by zero, Invalid input, File not found etc

Why Exception Handling?

- Prevent app crash
- Show user-friendly message
- Handle unexpected situations

```
void main() {
  try {
    int result = 10 ~/ 0;
    print(result);
  } catch (e) {
    print("Error occurred: $e");
  }
}
```

Exception Handling

```
try {
  int x = int.parse("abc");
} on FormatException {
  print("Invalid number
format");
}
```

```
try {
  int result = 10 ~/ 0;
} on
IntegerDivisionByZeroException
{
  print("Cannot divide by zero");
} catch (e) {
  print("Unknown error");
}
```

```
class InvalidAgeException implements
Exception {
  String errorMsg() => "Age must be above
18";
}

void checkAge(int age) {
  if (age < 18) {
    throw InvalidAgeException();
  }
}

void main() {
  try {
    checkAge(15);
  } catch (e) {
    print(e);
  }
}
```

Collections

List -

```
List<String> items = ["A", "B"];
```

Map - Used for: JSON, API responses

```
Map<String, dynamic> user = {
  "name": "Amit",
  "age": 22
};
```

Collection for & if -

```
children: [
  for (var item in items) Text(item),
  if (isAdmin) AdminWidget()
]
```

Asynchronous Programming

Future -

```
Future<String> fetchData() async {
  return "Data loaded";
}
```

async & await -

```
var data = await fetchData();
```

Streams (Real-Time Data) -

```
Stream<int> counter() async* {
  for (int i = 0; i < 5; i++) {
    yield i;
  }
}
```

Thank You