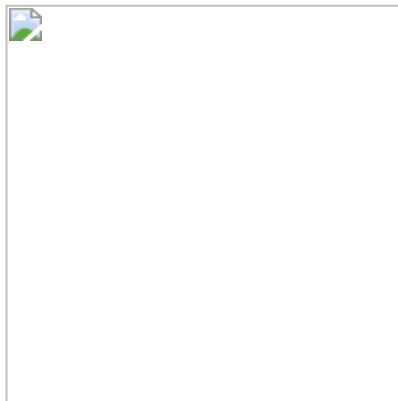


# Dart-Flutter-interview-questions

## 0-Flutter architectural overview.

- Flutter is a cross-platform UI toolkit that is designed to allow code reuse across operating systems such as iOS and Android, while also allowing applications to interface directly with underlying platform services. The goal is to enable developers to deliver high-performance apps that feel natural on different platforms, embracing differences where they exist while sharing as much code as possible.
- During development, Flutter apps run in a VM that offers stateful hot reload of changes without needing a full recompile. For release, Flutter apps are compiled directly to machine code, whether Intel x64 or ARM instructions, or to JavaScript if targeting the web. The framework is open source, with a permissive BSD license, and has a thriving ecosystem of third-party packages that supplement the core library functionality.

### Architectural layers.



- Flutter is designed as an extensible, layered system. It exists as a series of independent libraries that each depend on the underlying layer. No layer has privileged access to the layer below, and every part of the framework level is designed to be optional and replaceable.

### Embedder.

- To the underlying operating system, Flutter applications are packaged in the same way as any other native application. A platform-specific embedder provides an entrypoint; coordinates with the underlying operating system for access to services like rendering surfaces, accessibility, and input; and manages the message event loop. The embedder is written in a language that is appropriate for the platform: currently Java and C++ for Android, Objective-C/Objective-C++ for iOS and macOS, and C++ for Windows and Linux. Using the embedder, Flutter code can be integrated into an existing application as a module, or the code may be the entire content of

the application. Flutter includes a number of embedders for common target platforms, but other embedders also exist.

## Engine.

- At the core of Flutter is the Flutter engine, which is mostly written in C++ and supports the primitives necessary to support all Flutter applications. The engine is responsible for rasterizing composited scenes whenever a new frame needs to be painted. It provides the low-level implementation of Flutter's core API, including graphics (through Impeller on iOS and coming to Android, and Skia on other platforms) text layout, file and network I/O, accessibility support, plugin architecture, and a Dart runtime and compile toolchain.
- The engine is exposed to the Flutter framework through `dart:ui`, which wraps the underlying C++ code in Dart classes. This library exposes the lowest-level primitives, such as classes for driving input, graphics, and text rendering subsystems.

## Framework.

- Typically, developers interact with Flutter through the Flutter framework, which provides a modern, reactive framework written in the Dart language. It includes a rich set of platform, layout, and foundational libraries, composed of a series of layers. Working from the bottom to the top, we have:
- Basic foundational classes, and building block services such as animation, painting, and gestures that offer commonly used abstractions over the underlying foundation.
- The rendering layer provides an abstraction for dealing with layout. With this layer, you can build a tree of renderable objects. You can manipulate these objects dynamically, with the tree automatically updating the layout to reflect your changes.
- The widgets layer is a composition abstraction. Each render object in the rendering layer has a corresponding class in the widgets layer. In addition, the widgets layer allows you to define combinations of classes that you can reuse. This is the layer at which the reactive programming model is introduced.
- The Material and Cupertino libraries offer comprehensive sets of controls that use the widget layer's composition primitives to implement the Material or iOS design languages.
- The Flutter framework is relatively small; many higher-level features that developers might use are implemented as packages, including platform plugins like camera and webview, as well as platform-agnostic features like characters, http, and animations that build upon the core Dart and Flutter libraries. Some of these packages come from the broader ecosystem, covering services like in-app payments, Apple authentication, and animations.

## 1- What is a typedef ?

- In Dart , typedef or (function types alias) helps to define pointer to excute code within memory.
- typedef can be used as a pointer that references a function.
- It is a type alias allows you to provide a new name for an existing data type into your program.
- Type alias don't create a new types , they simply provide a new name to an existing type.

```
typedef IntList=List<int>;  
IntList il=[1,2,3];
```

### ChatGPT Answer:

- In Dart, typedef is used to create a named function type. It allows you to define a function signature with a custom name, which can then be used to declare variables and parameters of that function type. This can make your code more readable and expressive, especially when dealing with complex or recurring function signatures.

Here's a simple example:

```
typedef String GreetFunction(String name);  
  
String greetSomeone(GreetFunction greeter, String name) {  
    return greeter(name);  
}  
  
String englishGreeting(String name) {  
    return 'Hello, $name!';  
}  
  
String spanishGreeting(String name) {  
    return '¡Hola, $name!';  
}  
  
void main() {  
    print(greetSomeone(englishGreeting, 'John'));  
    print(greetSomeone(spanishGreeting, 'Juan'));  
}
```

- In this example, GreetFunction is a typedef representing a function that takes a String parameter and returns a String. englishGreeting and spanishGreeting are functions that match this signature, so they can be used wherever a GreetFunction is expected.

## 2- Difference between const and final ?

**final.**

- allows only a single assignment.
- It must have an initializer.
- once it has been initialized with a value, the variable cannot be reassigned.
- A final variable may rely on runtime execution of code to determine its state, but it must occur during initialization.
- :thumbsup:

```
final Datetime datetime=Datetime.now();
```

**const.**

- are compile time const.
- A Constant's entire deep state must be determinable at compile time.
- the const value will be frozen and immutable while the program is running.
- :thumbsdown:

```
const Datetime datetime=Datetime.now();      xxxxxx  
// Error
```

### 3- Does Dart Single threaded language and why ?

- Yes, Dart is a single threaded language .
- Because it use only one thread.
- It excutes one thing at a time with the support of isolates , event loop and asynchronous code.
- The event loop handles the excution of multiple chunks of your program over time.
- the event loop is what allows multiple operations in a non-blocking way.

#### **ChatGPT Answer:**

-Yes, Dart is primarily a single-threaded language. The Dart language itself is designed to be single-threaded, meaning that it doesn't inherently support concurrent execution of multiple threads within the same process. This design choice simplifies the language and makes it easier to reason about code in many cases.

- However, Dart does provide support for asynchronous programming through its Future and Stream classes, which allow you to work with asynchronous operations without the need for explicit threads. Asynchronous programming in Dart is based on the concept of event loops and callbacks, making it possible to handle I/O operations, such as network requests or file operations, without blocking the main thread.
- The Dart runtime uses an event loop to efficiently manage asynchronous tasks. This model is known as the single-threaded event loop model. It allows Dart programs to perform non-blocking I/O operations, ensuring that the application remains responsive even when waiting for external resources.
- While Dart itself is single-threaded, it's important to note that you can still achieve concurrent-like behavior through asynchronous programming without the need for explicit multi-threading. This design choice aligns with Dart's focus on simplicity, ease of use, and suitability for client-side development (e.g., Flutter for mobile and web applications).

### 4- What is the use of private keyword in dart ?

- Private keyword doesn't exist in dart.

- We use underScore before the variable name instead of private.
- It is used for encapsulating and securing your code, It makes it accessible only within their respective classes or libraries.

## 5- What is the meaning of immutable ?

- immutable constructs that cannot be mutated (altered) after they have been initialized.
- Strings, numbers and boolean can't be mutated.
- a String variable doesn't contain data itself, it is a reference to the string data's location in memory.
- non final string variables can be reassigned.

### ChatGPT Answer:

- In Dart, "immutable" refers to objects whose state cannot be changed after they are created. Once an immutable object is instantiated, its values cannot be modified. This characteristic is important for creating more predictable and safer code, especially in scenarios where you want to ensure that the data remains constant throughout its lifecycle.
- Dart provides several built-in classes that are immutable, such as: 1- String: Once a Dart string is created, you cannot change its individual characters.

```
String immutableString = "Hello";
// This will create a new string, it doesn't modify the original one
immutableString += " World";
```

2- int and double: Dart's integer and double types are also immutable.

```
int immutableInt = 42;
// This will create a new int, it doesn't modify the original one
immutableInt += 10;
```

- Using immutable objects can lead to more predictable code, as you don't have to worry about unexpected changes in the object's state. It also facilitates reasoning about code in concurrent or asynchronous scenarios. Dart's final keyword is often used to create variables that, once assigned, cannot be reassigned to a different object, providing a level of immutability for the variable itself.

### advantages of using immutable data.

- it's inherently thread safe, because since no code can alter its content , it is guaranted to the same reference no matter what code is accessing it.

## 6- What is a singleton ?

- Singleton pattern ensures that a class has only one instance and also provides a global point of access to it.

```

class Singleton {
  static final Singleton _singleton = Singleton._internal();

  factory Singleton() {
    return _singleton;
  }

  Singleton._internal();
}

main() {
  var s1 = Singleton();
  var s2 = Singleton();
  print(identical(s1, s2)); // true
  print(s1 == s2);         // true
}

```

## 7- What is isolate in dart ?

- An isolate in Dart is an independent unit of execution that runs concurrently in a separate memory heap.
- It allows for parallel processing and better utilization of multi-core systems.
- Isolates don't share memory; they communicate with each other by passing messages (usually immutable data) using ports.
- This approach helps to avoid common issues related to shared-memory concurrency, like race conditions or deadlocks.
- Each isolate has its own event queue and event loop, which processes incoming messages and ensures that only one message is handled at a time, preserving sequential consistency within the isolate.
- Isolates communicate with each other by passing messages, and they are independent of each other in terms of memory. This isolation provides a level of safety when dealing with concurrent programming, as there is no shared state that can be modified by multiple isolates simultaneously.
- To create an isolate in Dart, you use the `Isolate` class. Each isolate runs its own Dart program, and communication between isolates is typically done through message passing using the `SendPort` and `ReceivePort` classes.
- Isolates are especially useful for tasks that can be parallelized, such as processing data in the background or performing heavy computations without blocking the main thread of an application.

```

import 'dart:isolate';

void main() async {
  // Create a ReceivePort for the main isolate
  ReceivePort mainReceivePort = ReceivePort();

  // Spawn a new isolate and send the mainReceivePort to it
  Isolate.spawn(isolateFunction, mainReceivePort.sendPort);

  // Receive and print messages from the spawned isolate
  mainReceivePort.listen((message) {
    print('Received message from isolate: $message');
  });

  // Send a message to the spawned isolate
  mainReceivePort.sendPort.send('Hello from the main isolate!');
}

void isolateFunction(SendPort sendPort) {
  // Create a ReceivePort for the spawned isolate
  ReceivePort isolateReceivePort = ReceivePort();

  // Send the isolateReceivePort back to the main isolate
  sendPort.send(isolateReceivePort.sendPort);

  // Receive and print messages from the main isolate
  isolateReceivePort.listen((message) {
    print('Received message from main isolate: $message');
  });

  // Send a message to the main isolate
  sendPort.send('Hello from the spawned isolate!');
}

```

## 8- Difference between function and method ?

function.

- is a top-level function which is declared outside of a class or an inline function.
- doesn't need an object as independent.

method.

- is tied to an instance of a class and has an implicit reference to this.
- is a function which is linked with an object
- we call it by the object's name.

## 9- What is a sealed class ?

- Can't be extended outside of its library "file".

## 10- What is an inherited widget in flutter ?

- Base class of widgets that efficiently propagates information down the tree ( Scaffold, Theme ).

## 11- Difference between var and dynamic ?

var.

- can't change TYPE of the variable.
- but can change the VALUE of the variable later in code.

dynamic.

- can change TYPE of the variable.
- can change VALUE of the variable later in code.

## 12- Difference between Visibility, Opacity & OffStage ?

Visibility.

- Visibility widget is to show/hide widgets.
- it has a visible property, which helps us in showing or hiding child.
- when visible is set to false, the actual child is replaced with zero- sized box instead.

```
// manage visibility
Visibility(

    // showing the child widget
    visible: true, // true to show, false to hide
    child: Text(
        "Visible/Hide Text Example",

    ),

),
```

Opacity.



- The opacity widget in flutter is used to make it's child transparent.
- it can be done by using opacity property and set it 0, (range 0.0 - 1.0).
- The child is completely invisible when set to opacity 0, but will acquire the space & widget is also interactive.

```
Opacity(  
  opacity: _visible ? 1.0 : 0.0,  
  child: const Text("Now you see me, now you don't!"),  
)
```

## OffStage.

- OffStage in flutter widget lays it child out, as if it in true but don't draw/render the child widget view on screen, and all don't take space of screen.
- most of the time offstage is used to measure the dimension of widget without showing it on screen.

```

import 'package:flutter/material.dart';

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  static const String _title = 'Flutter Code Sample';

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: _title,
      home: Scaffold(
        appBar: AppBar(title: const Text(_title)),
        body: const Center(
          child: MyStatefulWidget(),
        ),
      ),
    );
  }
}

class MyStatefulWidget extends StatefulWidget {
  const MyStatefulWidget({Key? key}) : super(key: key);

  @override
  State<MyStatefulWidget> createState() => _MyStatefulWidgetState();
}

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  final GlobalKey _key = GlobalKey();
  bool _offstage = true;

  Size _getFlutterLogoSize() {
    final RenderBox renderLogo =
      _key.currentContext!.findRenderObject()! as RenderBox;
    return renderLogo.size;
  }

  @override
  Widget build(BuildContext context) {
    return Column(

```

```

mainAxisAlignment: MainAxisAlignment.center,
children: <Widget>[
  Offstage(
    offstage: _offstage,
    child: FlutterLogo(
      key: _key,
      size: 150.0,
    ),
  ),
  Text('Flutter logo is offstage: $_offstage'),
  ElevatedButton(
    child: const Text('Toggle Offstage Value'),
    onPressed: () {
      setState(() {
        _offstage = !_offstage;
      });
    },
  ),
  if (_offstage)
    ElevatedButton(
      child: const Text('Get Flutter Logo size'),
      onPressed: () {
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
            content:
              Text('Flutter Logo size is ${_getFlutterLogoSize()}'),
          ),
        );
      },
    ),
],
);
}
}

```

- ChatGpt Answer
- In Flutter, the Offstage widget is used to conditionally hide its child widget without removing it from the widget tree. It takes a boolean parameter called offstage to determine whether its child should be visible or hidden.
- Here's a simple example:

```

Offstage(
  offstage: condition,
  child: SomeWidget(),
)

```

- In this example, if the condition is true, the SomeWidget will be hidden (offstage), and if the condition is false, the SomeWidget will be visible.
- Using Offstage can be more efficient than using other methods like setting the Visibility or Container widget to hide/show the child, because it avoids the cost of rebuilding the widget tree when hiding or showing the child.
- Keep in mind that even though the widget is offstage, it still consumes memory and resources, so it's not suitable for situations where you want to optimize performance by completely removing a widget from the tree. In such cases, you might want to consider conditional widget creation or using other techniques depending on your specific use case.
- The Visibility and Offstage widgets in Flutter serve similar purposes—they both control the visibility of their child widgets based on a condition. However, there are some key differences between them:
  - 1- Rebuilding the Widget Tree:
    - When you use Visibility and set visible to false, the child widget is still part of the widget tree, and it will be built and laid out even though it's not visible. This means it might have some performance implications in terms of widget tree construction.
    - On the other hand, Offstage avoids building and laying out its child widget when offstage is set to true. It is more efficient in terms of performance because it skips the build process for the offstage child.
  - 2-Space Occupancy:
    - With Visibility, even if the child is not visible, it still occupies space in the layout. The space is there, but the contents are just not drawn.
    - Offstage not only hides the child but also avoids it occupying any space in the layout when offstage is set to true.
- In general, if you need to toggle visibility frequently and want a more efficient approach in terms of performance, Offstage might be a better choice. If you still want the child to occupy space even when hidden, then Visibility would be more appropriate.

## 13- Difference between Expanded and Flexible ?

- Flexible takes only the needed space, and Expanded takes all available space, respecting the flex factor.
- The Expanded widget simply takes all available space.
- One important distinction between the Flexible and Expanded widgets is that Flexible does not force its children to take the same size as itself where the Expanded widget forces its child to take the same size.
- Widget under Flexible are by default WRAP\_CONTENT although you can change it using parameter fit.
- Widget under Expanded is MATCH\_PARENT you can change it using flex.

## 14- What id generic programming ?

- Style of computer programming in which algorithms are written in terms of types to be specified later that are then instantiated when needed for specific types provided.

## 15- Solid Principles ?

S=> Single responsibility principle .

- A class should only be responsible for one thing that means a class could change for one reason.

## O=> Opened-Closed Principle .

- An entity should be open for extension but closed for modification.
- Adding new features without modifying the existing code.

## L=>Liskov substitution principle.

- Objects of a superclass should be replaceable with objects of its subclass without breaking the application.
- Objects of our subclass behaves the same way as objects of our superclass.

## I=>Interface Segregation Principle .

- It states that no client should be forced to depend on methods it does not use.
- Client should never depend on any thing more than the method it's calling.

## D=>Dependency inversion principle .

- Abstractions should not depend on details (Concrete implementations), they should depend on abstractions.
- You will be able to change the implementation without altering the high level code.
- Depending on abstractions gives the freedom to be independent of the implementations.

## 16- What is Records in dart ?

- Records are an anonymous, immutable, aggregate type. Like other collection types, they let you bundle multiple objects into a single object. Unlike other collection types, records are fixed-sized, heterogeneous, and typed.
- Records are real values; you can store them in variables, nest them, pass them to and from functions, and store them in data structures such as lists, maps, and sets.

```
var record = ('first', a: 2, b: true, 'last');
```

## 17- What is the difference between .map() and .foreach() methods in Dart while iterating over an array ?

- The major difference is forEach doesn't returns anything while map() does return another array which satisfies the conditions.
- forEach iterates over each element in the array, it's a function which won't return.

```
xyz = [1,2,3,4,5]

newArray = xyz.map( (x) => x == 3)

xyz.forEach( (x) {
  if(x == 3) print("Yo");
})
```

## 18- What is Spread Operator and Null-aware Spread Operator in dart ?

- In Dart, Spread Operator (...) and Null-aware Spread Operator (...?) are used for inserting multiple elements in a collection like Lists, Maps, etc.
- Spread Operator: Used to bundle a list into another list.

```
...Data_structure
```

- The Null-aware Spread Operator (...?) in Dart is used to perform a spread operation only if the operand is non-null. It's a convenient way to conditionally spread elements in a collection, such as a list or a map, without causing a null reference error.

```
...?Data_structure
```

```
List<int>? numbers = [1, 2, 3];
List<int> moreNumbers = [4, 5, ...?numbers];

print(moreNumbers); // Output: [4, 5, 1, 2, 3]
```

```
void main() {

    // initialise a List l1
    List? l1 = ["Geeks","For","Geeks"];

    // initialize another List l2 using l1
    List? l2=["Wow",...l1,"is","amazing"];

    // print List l2
    print(l2);
}
```

## 19- Difference between .then() and .whenCompleted() methods when working with Futures ?

- .whenComplete will fire a function either when the Future completes with an error or not, instead .then will fire a function after the Future completes without an error.
- catchError runs if the future fails.

```
someFuture().then((value) {  
    print('Future finished successfully i.e. without error');  
}).catchError((error) {  
    print('Future finished with error');  
}).whenComplete(() {  
    print('Either of then or catchError has run at this point');  
});
```

## 20- What is a mixin class in dart ?

- A way of defining code that can be reused in multiple class hierarchies.

```
mixin Mixin{}  
  
class Me extends Person with Mixin{}
```

## 21- What is a stream in dart ?

- Define asynchronous programming in dart.
- A stream can be thought of as series of asynchronous occurrence.
- Could be thought of as a pipe and the water inside it is the data.

## 22- Explain OOP principles in programming ?

### Abstraction.

- The process of hiding the internal data and implementation details from the outer world.
- Its used to describe things in simple terms.
- It's used to create a boundary between the application and the client programs.

### Abstract class.

- Cannot be instantiated and can contain abstract and non-abstract methods.

### Interface class.

- Specifies a set of methods that a class must implement.

### Benefits.

- Reduces boilerplate code (Code sections repeated throughout a program or software).
- Increase reuseability, maintainability and readability.

## Inheritance.

- One class inherits the attributes and methods of another class.
- Child class from parent class.

## Benefits.

- Code reusability, save time and reduce code boilerplate.

## Types.

- Single Inheritance.
- Multiple Inheritance (Dart doesn't support multiple inheritance but we can use mixins to inherit multiple classes).
- Multilevel Inheritance.

## Polymorphism.

- Multi form => the concept that you can access objects of different types through the same interface.
- The ability of any data to be processed in more than one form.
- the ability of an object to take many forms.
- Allows to compensate any reference of the super class by any object of the sub class.
- Allows to perform a single action in different ways.
- Reduces Coupling.
- Increase Reusability.

```
abstract class Person{}  
class Me extends Person{}  
class Him extends Person{}  
  
Person me=new Me();  
  
Person him=new Him();
```

## Encapsulation.

- bundling of data and methods that operate on that data within single unit (class).
- The method of hiding the implementation details of a class from outside access and only exposing a public interface.
- It's a protective shield that prevent data from being accessed by code outside this shield.



- In dart we use `_var` instead of private keyword.
- Getters => Read only.
- Setters => Write only.

### **Benefits.**

- Data Hiding, increases flexibility and reusability.

## 23- Introduce the Class modifiers in dart "3"! ?

No modifier.

**To allow unrestricted permission to construct or subtype from any library, use a class or mixin declaration without a modifier. By default, you can:**

- Construct new instances of a class.
- Extend a class to create a new subtype.
- Implement a class or mixin's interface.
- Mix in a mixin or mixin class.

### **abstract.**

- To define a class that doesn't require a full, concrete implementation of its entire interface, use the abstract modifier.
- Abstract classes cannot be constructed from any library, whether its own or an outside library. Abstract classes often have abstract methods.
- If you want your abstract class to appear to be instantiable, define a factory constructor.

```

// Library a.dart
abstract class Vehicle {
    void moveForward(int meters);
}

// Library b.dart
import 'a.dart';

// Error: Cannot be constructed
Vehicle myVehicle = Vehicle();

// Can be extended
class Car extends Vehicle {
    int passengers = 4;
    // ...
}

// Can be implemented
class MockVehicle implements Vehicle {
    @override
    void moveForward(int meters) {
        // ...
    }
}

```

base.

**To enforce inheritance of a class or mixin's implementation, use the base modifier. A base class disallows implementation outside of its own library. This guarantees:**

- The base class constructor is called whenever an instance of a subtype of the class is created.
- All implemented private members exist in subtypes.
- A new implemented member in a base class does not break subtypes, since all subtypes inherit the new member. This is true unless the subtype already declares a member with the same name and an incompatible signature.
- You must mark any class which implements or extends a base class as base, final, or sealed. This prevents outside libraries from breaking the base class guarantees.

```

// Library a.dart
base class Vehicle {
  void moveForward(int meters) {
    // ...
  }
}

// Library b.dart
import 'a.dart';

// Can be constructed
Vehicle myVehicle = Vehicle();

// Can be extended
base class Car extends Vehicle {
  int passengers = 4;
  // ...
}

// ERROR: Cannot be implemented
base class MockVehicle implements Vehicle {
  @override
  void moveForward() {
    // ...
  }
}

```

interface.

**To define an interface, use the interface modifier. Libraries outside of the interface's own defining library can implement the interface, but not extend it. This guarantees:**

- When one of the class's instance methods calls another instance method on this, it will always invoke a known implementation of the method from the same library.
- Other libraries can't override methods that the interface class's own methods might later call in unexpected ways. This reduces the fragile base class problem.

```
// Library a.dart
interface class Vehicle {
  void moveForward(int meters) {
    // ...
  }
}

// Library b.dart
import 'a.dart';

// Can be constructed
Vehicle myVehicle = Vehicle();

// ERROR: Cannot be inherited
class Car extends Vehicle {
  int passengers = 4;
  // ...
}

// Can be implemented
class MockVehicle implements Vehicle {
  @override
  void moveForward(int meters) {
    // ...
  }
}
```

## abstract interface.

- The most common use for the interface modifier is to define a pure interface. Combine the interface and abstract modifiers for an abstract interface class.
- Like an interface class, other libraries can implement, but cannot inherit, a pure interface. Like an abstract class, a pure interface can have abstract members.

## final.

To close the type hierarchy, use the final modifier. This prevents subtyping from a class outside of the current library. Disallowing both inheritance and implementation prevents subtyping entirely. This guarantees:

- You can safely add incremental changes to the API.
- You can call instance methods knowing that they haven't been overwritten in a third-party subclass.

- Final classes can be extended or implemented within the same library. The final modifier encompasses the effects of base, and therefore any subclasses must also be marked base, final, or sealed.

```
// Library a.dart
final class Vehicle {
  void moveForward(int meters) {
    // ...
  }
}

// Library b.dart
import 'a.dart';

// Can be constructed
Vehicle myVehicle = Vehicle();

// ERROR: Cannot be inherited
class Car extends Vehicle {
  int passengers = 4;
  // ...
}

class MockVehicle implements Vehicle {
  // ERROR: Cannot be implemented
  @override
  void moveForward(int meters) {
    // ...
  }
}
```

**sealed.**

- To create a known, enumerable set of subtypes, use the sealed modifier. This allows you to create a switch over those subtypes that is statically ensured to be exhaustive.

The sealed modifier prevents a class from being extended or implemented outside its own library. Sealed classes are implicitly abstract.

- They cannot be constructed themselves.
- They can have factory constructors.
- They can define constructors for their subclasses to use.

- Subclasses of sealed classes are, however, not implicitly abstract.
- The compiler is aware of any possible direct subtypes because they can only exist in the same library. This allows the compiler to alert you when a switch does not exhaustively handle all possible subtypes in its cases:

```
sealed class Vehicle {}

class Car extends Vehicle {}

class Truck implements Vehicle {}

class Bicycle extends Vehicle {}

// ERROR: Cannot be instantiated
Vehicle myVehicle = Vehicle();

// Subclasses can be instantiated
Vehicle myCar = Car();

String getVehicleSound(Vehicle vehicle) {
    // ERROR: The switch is missing the Bicycle subtype or a default case.
    return switch (vehicle) {
        Car() => 'vroom',
        Truck() => 'VROOOOMM',
    };
}
```

- If you don't want exhaustive switching, or want to be able to add subtypes later without breaking the API, use the `final` modifier.

## 24- What is Enum ?

- Enum stands for enumerated type, a type of data where only a set of predefined values exist.
- In Dart, Enum is simply a special kind of class used to represent a fixed number of constant values.
- All enums automatically extend the Enum class. They are also sealed, meaning they cannot be subclassed, implemented, mixed in, or otherwise explicitly instantiated.
- Abstract classes and mixins can explicitly implement or extend Enum, but unless they are then implemented by or mixed into an enum declaration, no objects can actually implement the type of that class or mixin.
- makes the code readable and reusable.

```
enum OperatingSystem { macOS, windows, linux }

final favsystem=OperatingSystem.linux;

if(favsystem==OperatingSystem.linux){}
```

## 25- Difference Override and overload in programming ?

- Firstly, Dart doesn't support overloading.
- Overriding occurs when the method signature is the same in the superclass and the child class.
- Overloading occurs when two or more methods in the same class have the same name but different parameters.
- Method overloading deals with the notion of having two or more methods in the same class with the same name but different arguments.

```
void foo(int a)
void foo(int a, float b)
```

- Method overriding means having two methods with the same arguments, but different implementations. One of them would exist in the parent class, while another will be in the derived, or child class. The `@Override` annotation, while not required, can be helpful to enforce proper overriding of a method at compile time.

```
class Parent {
    void foo(double d) {
        // do something
    }
}

class Child extends Parent {

    @Override
    void foo(double d){
        // this method is overridden.
    }
}
```

## 26- Extensions in dart?

- Extensions methods add functionality to existing libraries.

```
extension NumberParsing on String {}
```

## 27- Named Constructor?

- Giving your constructor different names allows your class to have many constructors and also to be better represent their use cases outside of the class.
- `Image.asset()`.
- `Image.network()`.

## 28- Factory constructor?

- factory keyword is used when implementing a constructor that doesn't always create a new instance of its class.
- it might return an instance of a subtype.
- it's used to initialize a final variable using logic that can't be handled in the initializer list.

## 29- Design patterns?

### 1- Creational design patterns?

- Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.
- Singleton => Ensure a class only has one instance, and provide a global point of access to it.
- One of the main advantages of using the singleton pattern is that it can reduce the memory usage of your application. By creating only one instance of a class, you can avoid allocating and deallocating memory for multiple objects of the same type.
- Instance control: Singleton prevents other objects from instantiating their own copies of the Singleton object, ensuring that all objects access the single instance.
- Instance control: Singleton prevents other objects from instantiating their own copies of the Singleton object, ensuring that all objects access the single instance.
- The singleton class provides a global access point to get the instance of the class.

### 2- Structural patterns?

- Adapter or Wrapper. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Decorator or Wrapper. Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

### 3- Behavioral patterns?

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.
- Observer. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

## 30- Iterables?

- Iterable is a collection of elements that can be accessed sequentially.
- Iterable is an abstract class .
- You can't instantiate it directly. You can create a new Iterable by creating new list.
- List => Read elements by their indexes.



- Set => Contain elements that occur once.
- Map => Read elements using key.

## 31- this keyword?

- Is used to remove ambiguity that can be caused if the class attributes and the parameters have the same name.
- represent an implicit object pointing to the current class object.
- The this keyword is used to refer to the current class instance. this can be used to invoke the current class's constructors or methods. It also helps access the current class's instance variables. The this keyword can be used to set the instance variable's values or get the current instance of the class.
- In Dart, the this keyword is a reference to the current instance of the class. It is often used to differentiate between instance variables and local variables when they have the same name. Here are a few examples to illustrate the usage of this:

```
// Differentiating between instance and local variables
class Person {
  String name;

  Person(this.name); // Constructor

  void printName() {
    print('Name: $name');
  }

  void updateName(String newName) {
    // Using 'this' to refer to the instance variable
    this.name = newName;
  }
}

void main() {
  Person person = Person('John Doe');
  person.printName(); // Output: Name: John Doe

  person.updateName('Jane Doe');
  person.printName(); // Output: Name: Jane Doe
}
```

- In this example, the updateName method takes a parameter with the same name as the instance variable name. The this keyword is used to refer to the instance variable, making it clear that we want to update the name of the current instance.

```
// Using 'this' in a constructor
class Rectangle {
  double width;
  double height;

  Rectangle(this.width, this.height); // Constructor

  double calculateArea() {
    // Using 'this' to access instance variables in a method
    return this.width * this.height;
  }
}

void main() {
  Rectangle rectangle = Rectangle(5.0, 10.0);
  print('Area: ${rectangle.calculateArea()}'); // Output: Area: 50.0
}
```

- In this example, the Rectangle class has a constructor that uses the this keyword to initialize the instance variables width and height. The calculateArea method also uses this to access the instance variables within the method.
- The this keyword is optional in many cases in Dart, and you can often omit it. However, using this can make your code more explicit and readable, especially in situations where variable names might be ambiguous.

## 32- Types of streams?

- Streams provide an asynchronous sequence of data .
- data sequences include use-generated events and data read from files.

### Types of Straems:

- Single Subscription Stream => Contains Sequence of events that are parts of a larger whole , Events need to be delivered in the correct order and without missing any of them. (read file , recieve web request). Such a stream can only be listened to once , listening again later could mean missing out on initial events. when you start listening, the data will be fetched and provided in chuncks.
- Broad Cast Stream => Intended for individual messages that can be handled one at a time . (Mouse events in a browser. You can start listening at any time and you get the events that are fired while you listen, more than a alistner can listen at the same time. and you can listen again after canceling a previous subscription.

## 33- Get, Post, Delete, update?

- The GET method is used to retrieve data from the server. This is a read-only method, so it has no risk of mutating or corrupting the data. For example, if we call the get method on our API, we'll get back a list of all to-dos.

- The POST method sends data to the server and creates a new resource. The resource it creates is subordinate to some other parent resource. When a new resource is POSTed to the parent, the API service will automatically associate the new resource by assigning it an ID (new resource URI). In short, this method is used to create a new data entry.
- The PUT method is most often used to update an existing resource. If you want to update a specific resource (which comes with a specific URI), you can call the PUT method to that resource URI with the request body containing the complete new version of the resource you are trying to update.
- The PATCH method is very similar to the PUT method because it also modifies an existing resource. The difference is that for the PUT method, the request body contains the complete new version, whereas for the PATCH method, the request body only needs to contain the specific changes to the resource, specifically a set of instructions describing how that resource should be changed, and the API service will create a new version according to that instruction.
- The DELETE method is used to delete a resource specified by its URI.
- `readBytes =>` Sends an HTTP GET request with the given headers to the given URL and returns a Future that completes to the body of the response as a list of bytes.

```
Future<Uint8List> readBytes(
  Uri url,
  {Map<String, String>? headers}
)
```

## 34- Blocprovider, bloclistener, blocbuilder, blocconsumer ?

- `BlocProvider =>` Takes a Create function that is responsible for creating the Bloc or Cubit and a child which will have access to the instance via `BlocProvider.of(context)`. It is used as a dependency injection (DI) widget so that a single instance of a Bloc or Cubit can be provided to multiple widgets within a subtree.

```
BlocProvider(
  create: (BuildContext context) => BlocA(),
  child: ChildA(),
);
```

- `BlocBuilder =>` `BlocBuilder` handles building a widget in response to new states. `BlocBuilder` is analogous to `StreamBuilder` but has simplified API to reduce the amount of boilerplate code needed as well as bloc-specific performance improvements.

```
BlocBuilder<BlocA, BlocAState>(
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

- **BlocListener** => Takes a **BlocWidgetListener** and an optional **bloc** and invokes the listener in response to state changes in the bloc. It should be used for functionality that needs to occur only in response to a state change such as navigation, showing a **SnackBar**, showing a **Dialog**, etc... The listener is guaranteed to only be called once for each state change unlike the builder in **BlocBuilder**. If the **bloc** parameter is omitted, **BlocListener** will automatically perform a lookup using **BlocProvider** and the current **BuildContext**.

```
BlocListener<BlocA, BlocAState>(
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container(),
)
```

- The builder is run upon state change. But the builder function is also run when the framework deems necessary to rebuild.
- The listener function is not affected by the frameworks need to rebuild.
- **BlocConsumer** => **BlocConsumer** exposes a builder and listener in order react to new states. **BlocConsumer** is analogous to a nested **BlocListener** and **BlocBuilder** but reduces the amount of boilerplate needed. **BlocConsumer** should only be used when it is necessary to both rebuild UI and execute other reactions to state changes in the bloc. **BlocConsumer** takes a required **BlocWidgetBuilder** and **BlocWidgetListener** and an optional **bloc**, **BlocBuilderCondition**, and **BlocListenerCondition**. If the **bloc** parameter is omitted, **BlocConsumer** will automatically perform a lookup using **BlocProvider** and the current **BuildContext**.

```
BlocConsumer<BlocA, BlocAState>(
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

## 35- listview, listview.builder?

- The **ListView** constructor requires us to create all items at once. This is good when list items are less and all will be seen on the screen, but if not then for long list items its not the good practice.
- the **ListView.Builder** constructor will create items as they are scrolled onto the screen like on-demand. This is the best practice for development for List widget where items will only render only when items are going seen on the screen.

## 36- AssetImage, Image.asset?

- Image is a StatefulWidget and Image.asset is just a named constructor, you can use it directly on your widget tree.
- AssetImage is an ImageProvider which is responsible for obtaining the image of the specified path.
- AssetImage => Image provider => provides an image from the pass , It is not a widget.
- Image.asset => Creates a widget.

## 37- What Does WidgetsFlutterBinding.ensureInitialized() do ?

- The WidgetsFlutterBinding is used to interact with the Flutter engine.
- Ex. Firebase.initializeApp() needs to call native code to initialize Firebase, and since the plugin needs to use platform channels to call the native code, which is done asynchronously therefore you have to call ensureInitialized() to make sure that you have an instance of the WidgetsBinding.
- Returns an instance of the WidgetsBinding, creating and initializing it if necessary. If one is created, it will be a WidgetsFlutterBinding. If one was previously initialized, then it will at least implement WidgetsBinding.
- You only need to call this method if you need the binding to be initialized before calling runApp.

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}
```

## 38- Json?

- Javascript object notation => It is a standard text-based format for representing structured data based on javascript object syntax.
- It is commonly used for transmitting data in web applications.

## 39- Difference between Bloc and Cubit?

- Cubit is a subset of the BLoC Pattern package that does not rely on events and instead uses methods to emit new states.
- Cubit is a subset of Bloc; so, it reduces complexity. Cubit eliminates the event classes. Cubit uses emit rather than yield to emit state. Since emit works synchronously, you can ensure that the state is updated in the next line.
- Cubit is not event driven.
- Bloc is event driven.

## 40- What is a context?

- A handle to the location of a widget in the widget tree.
- In Flutter, the context refers to the location of a widget in the widget tree. It provides information about the surrounding environment and services that the widget might need. For example, the context can contain

information such as the theme of the app, the locale, the screen size, and more.

## 41- What is a key?

- Preserves state when widget move around in the widget tree.
- Keys can be found in almost every widget as named parameters. It is useful when we need to store the widget's state having the same type of data. It's used with a stateful widget where states change.
- You don't need to use Keys most of the time, the framework handles it for you and uses them internally to differentiate between widgets. There are a few cases where you may need to use them though.
- A common case is if you need to differentiate between widgets by their keys, `ObjectKey` and `ValueKey` can be useful for defining how the widgets are differentiated.
- Another example is that if you have a child you want to access from a parent, you can make a `GlobalKey` in the parent and pass it to the child's constructor. Then you can do `globalKey.state` to get the child's state (say for example in a button press callback). Note that this shouldn't be used excessively as there are often better ways to get around it.
- In Flutter, the term "key" refers to a way of identifying and associating widgets in the widget tree. Keys are used to maintain state across widget rebuilds and efficiently update the UI. Here are the key-related concepts in Flutter:
- **Widget Key:** In Flutter, a key is an object that you can assign to a widget to uniquely identify it. Keys are optional, and if you don't provide a key, Flutter will use default mechanisms to identify and update widgets. However, in certain situations, using keys can be beneficial for performance optimization and state management.

```
Widget myWidget = Container(  
  key: Key('myUniqueKey'),  
  // other widget properties  
);
```

### Types of keys:

- Flutter provides several types of keys, and each serves a specific purpose. The commonly used key types include:
- **Key:** The base type for all keys. It is generally used when a simple, unique identifier is needed.

```
Widget myWidget = Container(  
  key: Key('myUniqueKey'),  
  // other widget properties  
);
```

- **ValueKey:** A key that uses the actual value as its identifier. It's useful when you have a set of widgets that represent a collection of unique values.

```
Widget myWidget = Container(  
  key: ValueKey<String>('john_doe'),  
  // other widget properties  
);
```

- **ObjectKey:** A key that uses the identity of the given object. It's commonly used when dealing with custom objects.

```
Widget myWidget = Container(  
  key: ObjectKey(myCustomObject),  
  // other widget properties  
);
```

- **GlobalKey:** A key that is globally unique across the entire application. It's often used when you need to access the state of a widget from outside its parent.

```
GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();  
  
Widget build(BuildContext context) {  
  return Scaffold(  
    key: scaffoldKey,  
    // other widget properties  
  );  
}
```

### Key Usage Scenarios:

- **ListViews and ListItems:** When dealing with dynamic lists of widgets, using keys helps Flutter efficiently update the list items.
- **Stateful Widgets:** Keys are often used with GlobalKey to maintain state in stateful widgets and access the widget's state directly.
- **Animations and Page Transitions:** Keys can be useful when managing animations and page transitions to ensure that Flutter can properly update the UI.

**Using keys judiciously can enhance the performance and maintainability of your Flutter applications, especially in scenarios where the widget tree dynamically changes or widgets need to be uniquely identified.**

## 42- What is a widget?

- A blueprint or recipe of what this part of the ui should be, And they appear with other widgets to complete the whole ui.
- Widgets are the central class hierarchy in the Flutter framework. A widget is an immutable description of part of a user interface. Widgets can be inflated into elements, which manage the underlying render tree.

## 43- The use of static keyword in dart ?

- static used for memory management of global data members.
- the static variables and methods are part of the class instead of a specific instance.
- used for class-level variable and method that is the same for every instance of a class.
- can't be overridden.
- can be accessed without the class prefix only in the class where they are declared.
- class name.static variable.

## 44- What is SDK ?

- Software development Kit => that includes a comprehensive set of development tools ( Emulator, Libraries, Debugger ).

## 45- What is an Api ?

- Application Programming Interface => Software intermediary that allows two applications to talk to each other .
- Apis are an accessible way to extract and share data within and across organizations .
- let your product or service communicate with other without having to know how they were implemented.

## 46- What is dependency injection ?

- Design Pattern used to implement inversion of control .
- It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways.
- Is one of subtypes of inversion of control => decoupling.

## 47- What is PreferredSizeWidget ?

- An interface for widgets that can return the size this widget would prefer if it were otherwise unconstrained.
- There are a few cases, notably AppBar and TabBar, where it would be undesirable for the widget to constrain its own size but where the widget needs to expose a preferred or "default" size.
- Widgets that need to know the preferred size of their child can require that their child implement this interface by using this class rather than Widget as the type of their child property.
- Use PreferredSize to give a preferred size to an arbitrary widget.

## 48- What is WillPopScope class?

- Registers a callback to veto attempts by the user to dismiss the enclosing ModalRoute.



- Whenever the back button is pressed, you will get a callback at `onWillPop`, which returns a `Future`. If the `Future` returns true, the screen is popped.

```

class WillPopScopeExample extends StatefulWidget {
  const WillPopScopeExample({super.key});

  @override
  State<WillPopScopeExample> createState() => _WillPopScopeExampleState();
}

class _WillPopScopeExampleState extends State<WillPopScopeExample> {
  bool shouldPop = true;
  @override
  Widget build(BuildContext context) {
    return WillPopScope(
      onWillPop: () async {
        return shouldPop;
      },
      child: Scaffold(
        appBar: AppBar(
          title: const Text('Flutter WillPopScope demo'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              OutlinedButton(
                child: const Text('Push'),
                onPressed: () {
                  Navigator.of(context).push<void>(
                    MaterialPageRoute<void>(
                      builder: (BuildContext context) {
                        return const WillPopScopeExample();
                      },
                    ),
                  );
                },
              ),
              OutlinedButton(
                child: Text('shouldPop: $shouldPop'),
                onPressed: () {
                  setState(
                    () {
                      shouldPop = !shouldPop;
                    },
                  );
                },
              ),
            ],
          ),
        ),
      ),
    );
  }
}

```

```

        },
    ),
    const Text('Push to a new screen, then tap on shouldPop '
        'button to toggle its value. Press the back '
        'button in the appBar to check its behavior '
        'for different values of shouldPop'),
    ],
),
),
),
);
}
}

```

## 49- Spacer class ?

- Spacer creates an adjustable, empty spacer that can be used to tune the spacing between widgets in a Flex container, like Row or Column.
- The Spacer widget will take up any available space, so setting the Flex.mainAxisAlignment on a flex container that contains a Spacer to MainAxisAlignment.spaceAround, MainAxisAlignment.spaceBetween, or MainAxisAlignment.spaceEvenly will not have any visible effect: the Spacer has taken up all of the additional space, therefore there is none left to redistribute.

## 50 - callback and anonymous functions ?

- callback => a function or method that we pass as an argument into another function or method.
- VoidCallback typedef => Signature of callbacks that have no arguments and return no data.

```
typedef VoidCallback = void Function();
```

- anonymous => doesn't have a name

```
list.forEach((item) {});
```

## 51- Scaffold ?

- is responsible for implementing the basic material design , visual layout structure of an app.

## 52- super keyword ?

- is used to denote the instant parent class object of the current child class .
- It's used to invoke super class methods , super class constructor in its child class.
- to refer the super class properties and methods.

## 53- Materialapp ?

- A convenience widget that wraps a number of widgets that are commonly required for Material Design applications. It builds upon a WidgetsApp by adding material-design specific functionality, such as AnimatedTheme and GridPaper.
- MaterialApp configures its WidgetsApp.textStyle with an ugly red/yellow text style that's intended to warn the developer that their app hasn't defined a default text style. Typically the app's Scaffold builds a Material widget whose default Material.textStyle defines the text style for the entire scaffold.
- Material widgets implements the Material design language for iOS, Android, and web.
- Cupertino widgets implements the current iOS design language based on Apple's Human Interface Guidelines.

## 54- runApp ?

```
void runApp(  
Widget app  
)
```

- Inflate the given widget and attach it to the screen.
- The widget is given constraints during layout that force it to fill the entire screen. If you wish to align your widget to one side of the screen (e.g., the top), consider using the Align widget. If you wish to center your widget, you can also use the Center widget.
- Calling runApp again will detach the previous root widget from the screen and attach the given widget in its place. The new widget tree is compared against the previous widget tree and any differences are applied to the underlying render tree, similar to what happens when a StatefulWidget rebuilds after calling State.setState.
- The main() function is the entry point of the app and it is the first method that is executed when the app starts.
- The runApp() method is responsible for creating the WidgetsFlutterBinding which is the binding between the framework and the Flutter engine.

## 55- Changenotifier?

- A class that can be extended or mixed in that provides a change notification API using VoidCallback for notifications.
- A data structure can extend or mix in ChangeNotifier to implement the Listenable interface and thus become usable with widgets that listen for changes to Listenables, such as ListenableBuilder.

## 56- ListenableBuilder class?

- A general-purpose widget for building a widget subtree when a Listenable changes.
- ListenableBuilder is useful for more complex widgets that wish to listen to changes in other objects as part of a larger build function. To use ListenableBuilder, construct the widget and pass it a builder function.
- Any subtype of Listenable (such as a ChangeNotifier, ValueNotifier, or Animation) can be used with a ListenableBuilder to rebuild only certain parts of a widget when the Listenable notifies its listeners. Although they

have identical implementations, if an Animation is being listened to, consider using an AnimatedBuilder instead for better readability.

```

import 'package:flutter/material.dart';

/// Flutter code sample for a [ChangeNotifier] with a [ListenableBuilder].

void main() {
  runApp(const ListenableBuilderExample());
}

class CounterModel with ChangeNotifier {
  int _count = 0;
  int get count => _count;

  void increment() {
    _count += 1;
    notifyListeners();
  }
}

class ListenableBuilderExample extends StatefulWidget {
  const ListenableBuilderExample({super.key});

  @override
  State<ListenableBuilderExample> createState() =>
    _ListenableBuilderExampleState();
}

class _ListenableBuilderExampleState extends State<ListenableBuilderExample> {
  final CounterModel _counter = CounterModel();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text('ListenableBuilder Example')),
        body: CounterBody(counterNotifier: _counter),
        floatingActionButton: FloatingActionButton(
          onPressed: _counter.increment,
          child: const Icon(Icons.add),
        ),
      ),
    );
  }
}

```

```

class CounterBody extends StatelessWidget {
  const CounterBody({super.key, required this.counterNotifier});

  final CounterModel counterNotifier;

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          const Text('Current counter value:'),
          // Thanks to the ListenableBuilder, only the widget displaying the
          // current count is rebuilt when counterValueNotifier notifies its
          // listeners. The Text widget above and CounterBody itself aren't
          // rebuilt.
          ListenableBuilder(
            listenable: counterNotifier,
            builder: (BuildContext context, Widget? child) {
              return Text('${counterNotifier.count}');
            },
          ),
        ],
      ),
    );
  }
}

```

## 57 - ValueNotifier ?

- A ChangeNotifier that holds a single value.
- When value is replaced with something that is not equal to the old value as evaluated by the equality operator ==, this class notifies its listeners.
- Because this class only notifies listeners when the value's identity changes, listeners will not be notified when mutable state within the value itself changes.
- For example, a ValueNotifier<List> will not notify its listeners when the contents of the list are changed. As a result, this class is best used with only immutable data types. For mutable data types, consider extending ChangeNotifier directly.

## 58- notifylisteners ?

- Call all the registered listeners.

- Call this method whenever the object changes, to notify any clients the object may have changed. Listeners that are added during this iteration will not be visited. Listeners that are removed during this iteration will not be visited after they are removed.

## 59- Difference between inkwell and gesturedetector ?

- They both provide many common features like onTap, onLongPress etc. The main difference is GestureDetector provides more controls like dragging etc. on the other hand it doesn't include ripple effect tap, which InkWell does.
- You can use either of them according to your needs, you want ripple effects go with InkWell, need more controls go with GestureDetector or even combine both of them.

## 60- ColorScheme ?

- primary => The color displayed most frequently across your app's screens and components.
- onPrimary => A color that's clearly legible when drawn on primary.
- secondary => An accent color used for less prominent components in the UI, such as filter chips, while expanding the opportunity for color expression.
- error => The color to use for input validation errors, e.g. for InputDecoration.errorText.
- surface => The background color for widgets like Card.
- background => A color that typically appears behind scrollable content.

## 61- Textfield and textformfield ?

- If you making a Form where you require save, reset, or validate operations- use TextFormField. Else For Simple user input capture TextField is sufficient.
- TextFormField, which integrates with the Form widget.
- A Form ancestor is not required. The Form simply makes it easier to save, reset, or validate multiple fields at once.
- TextField, which is the underlying text field without the Form integration.
- Form class An optional container for grouping together multiple form field widgets (e.g. TextField widgets). Each individual form field should be wrapped in a FormField widget, with the Form widget as a common ancestor of all of those. Call methods on FormState to save, reset, or validate each FormField that is a descendant of this Form. To obtain the FormState, you may use Form.of with a context whose ancestor is the Form, or pass a GlobalKey to the Form constructor and call GlobalKey.currentState.



```
import 'package:flutter/material.dart';

/// Flutter code sample for [Form].

void main() => runApp(const FormExampleApp());

class FormExampleApp extends StatelessWidget {
  const FormExampleApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text('Form Sample')),
        body: const FormExample(),
      ),
    );
  }
}

class FormExample extends StatefulWidget {
  const FormExample({super.key});

  @override
  State<FormExample> createState() => _FormExampleState();
}

class _FormExampleState extends State<FormExample> {
  final GlobalKey<FormState> _formKey = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    return Form(
      key: _formKey,
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
          TextFormField(
            decoration: const InputDecoration(
              hintText: 'Enter your email',
            ),
            validator: (String? value) {
              if (value == null || value.isEmpty) {

```

```

        return 'Please enter some text';
    }
    return null;
  },
),
Padding(
  padding: const EdgeInsets.symmetric(vertical: 16.0),
  child: ElevatedButton(
    onPressed: () {
      // Validate will return true if the form is valid, or false if
      // the form is invalid.
      if (_formKey.currentState!.validate()) {
        // Process data.
      }
    },
    child: const Text('Submit'),
  ),
),
],
),
);
}
}

```

## 62- Flutter clip image?

- ClipRect - Clips the image in rectangle.

```

ClipRect(
  child: Container(
    child: Align(
      alignment: Alignment.center,
      widthFactor: 0.4,
      heightFactor: 1.0,
      child: Image.network(
        'https://images.unsplash.com/photo-1473992243898-fa7525e652a5'
      ),
    ),
  ),
)

```

ClipRect prevents the child from painting outside the box. The size and location of the clipper can be customized by changing the arguments in the clipper. The code above shows how to clip an image with a rectangular shape. You can wrap the image widget with an Align widget to easily position the image inside the clipper. The widthFactor and heightFactor properties are used to decide the size of the clipper and alignment is used to decide the position of the clipper. Here we are clipping the image at the center.

- ClipRRect - Clips the image with rounded corners or clip circle.

```
ClipRRect(  
  borderRadius: BorderRadius.circular(300.0),  
  child: Image.network(  
    "https://images.unsplash.com/photo-1523285367489-d38aec03b6bd"  
  ),  
)
```

ClipRRect can be used to clip image circle or clip the edges with a circular radius. The extra R stands for rounded. The borderRadius property can be used to change the radius of the rounded corners.

- ClipOval - Clips the image in oval.

```
ClipOval(  
  child: Container(  
    child: Image.network(  
      'https://images.unsplash.com/photo-1523965671143-ac717f1cb928'  
    ),  
  ),  
)
```

ClipOval can be used to clip the child widget in an oval shape. The clipper uses the widgets bounding box to determine the width and height of the oval. Therefore, if the width and height of the child widget are equal, then it will be a circle.

- ClipPath - Clips image in custom shape.

```

ClipPath(
  clipper: TriangleClipper(),
  child: Image.network(
    "https://images.unsplash.com/photo-1513775192371-1b9d33760c3f"
  ),
)

class TriangleClipper extends CustomClipper {
  @override
  Path getClip(Size size) {
    final path = Path();
    path.moveTo(size.width/2, 0.0);
    path.lineTo(size.width, size.height);
    path.lineTo(0.0, size.height);
    path.close();
    return path;
  }
  @override
  bool shouldReclip(TriangleClipper oldClipper) => false;
}

```

ClipPath can be used to clip the child widget in custom shape using a custom Clipper defined. In this code, we clip the widget in a triangle shape. Here we are supplying a custom TriangleClipper class to the clipper attribute. If you want to know more about how a custom clipper can be defined to achieve the desired shape, follow the detailed tutorial on [custom clippers here](#).

## 63- showBottomSheet vs showModalBottomSheet?

- A modal bottom sheet is an alternative to a menu or a dialog and prevents the user from interacting with the rest of the app. A closely related widget is a persistent bottom sheet, which shows information that supplements the primary content of the app without preventing the user from interacting with the app. Persistent bottom sheets can be created and displayed with the showBottomSheet function or the ScaffoldState.showBottomSheet method.
- DraggableScrollableSheet class A container for a Scrollable that responds to drag gestures by resizing the scrollable until a limit is reached, and then scrolling.

```

Future<T?> showModalBottomSheet<T>(
  {required BuildContext context,
  required WidgetBuilder builder,
  Color? backgroundColor,
  String? barrierLabel,
  double? elevation,
  ShapeBorder? shape,
  Clip? clipBehavior,
  BoxConstraints? constraints,
  Color? barrierColor,
  bool isScrollControlled = false,
  bool useRootNavigator = false,
  bool isDismissible = true,
  bool enableDrag = true,
  bool? showDragHandle,
  bool useSafeArea = false,
  RouteSettings? routeSettings,
  AnimationController? transitionAnimationController,
  Offset? anchorPoint}
)

PersistentBottomSheetController<T> showBottomSheet<T>(
  {required BuildContext context,
  required WidgetBuilder builder,
  Color? backgroundColor,
  double? elevation,
  ShapeBorder? shape,
  Clip? clipBehavior,
  BoxConstraints? constraints,
  bool? enableDrag,
  AnimationController? transitionAnimationController}
)

```

## 64- Uint8List class?

- A fixed-length list of 8-bit unsigned integers.
- For long lists, this implementation can be considerably more space- and time-efficient than the default List implementation.
- Integers stored in the list are truncated to their low eight bits, interpreted as an unsigned 8-bit integer with values in the range 0 to 255.
- It is a compile-time error for a class to attempt to extend or implement Uint8List.

## 65- How to upload images to server in Flutter?

- Use `MultipartRequest` class Such a request has both string fields, which function as normal form fields, and (potentially streamed) binary files. This request automatically sets the Content-Type header to `multipart/form-data`. This value will override any value set by the user.

```
Upload(File imageFile) async {
  var stream = new http.ByteStream(DelegatingStream.typed(imageFile.openRead()));
  var length = await imageFile.length();

  var uri = Uri.parse(uploadURL);

  var request = new http.MultipartRequest("POST", uri);
  var multipartFile = new http.MultipartFile('file', stream, length,
    filename: basename(imageFile.path));
    //contentType: new MediaType('image', 'png'));

  request.files.add(multipartFile);
  var response = await request.send();
  print(response.statusCode);
  response.stream.transform(utf8.decoder).listen((value) {
    print(value);
  });
}
```

## 66- override toString?

```

class BankAccount {
    double _balance = 0;

    BankAccount({double balance = 0}) : _balance = balance;

    double get balance => _balance;

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

    bool withdraw(double amount) {
        if (amount <= _balance) {
            _balance -= amount;
            return true;
        }
        return false;
    }

    @override
    String toString() {
        return 'The balance is $balance USD.';
    }
}

void main() {
    var account = BankAccount(balance: 100);
    print(account);
}

// output
The balance is 100.0 USD.

```

67- get\_it?

```

final sl = GetIt.instance;

Future<void> init() async {

  sl.registerLazySingleton<QuranCubit>(
    () => QuranCubit(quranUsecase: sl(), quranAudioUsecase: sl()));
  sl.registerLazySingleton<SurahCubit>(() => SurahCubit(surahUsecase: sl()));
  sl.registerLazySingleton<RecitationsCubit>(
    () => RecitationsCubit(recitationsUsecase: sl()));
  sl.registerLazySingleton<AudioCubit>(() => AudioCubit(audioUsecase: sl()));

}

void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await di.init();
  runApp(const MyApp());
}

providers: [
  BlocProvider(create: (context) => sl<QuranCubit>()),
  BlocProvider(create: (context) => sl<SurahCubit>()),
  BlocProvider(create: (context) => sl<RecitationsCubit>()),
],

```

## 68- How is flutter considered native ?

- Flutter is often described as providing a "native-like" experience because it uses a compiled programming language (Dart) and doesn't rely on web views. This allows Flutter apps to interact directly with the device's hardware, providing performance and user experience similar to native applications.
- Unlike some cross-platform frameworks that use web views to render content, Flutter doesn't rely on them. It compiles Dart code to native machine code, allowing Flutter apps to run directly on the device without the need for an intermediary web-based rendering engine. This contributes to a more efficient and responsive user experience.
- Frameworks like Apache Cordova or React Native often use web views to render the user interface. This involves embedding a web browser component within the app, interpreting JavaScript, HTML, and CSS to display the content. In contrast, Flutter doesn't use web views, opting for a direct compilation approach for native performance.
- Web views are components in mobile app development that display web content within an application. They essentially embed a web browser within the app, allowing the rendering of HTML, CSS, and JavaScript. This approach is common in cross-platform frameworks that leverage web technologies to create mobile apps.

## 69- difference between sealed class and enum?



- Sealed classes and enums are both constructs used in programming languages for modeling and representing a fixed set of values or types, but they have some key differences.

## **Sealed Class:**

### **Hierarchy and Subtypes:**

- Sealed classes allow you to define a hierarchy of types, similar to regular classes.
- Subtypes of a sealed class can be created and extended, providing more flexibility in terms of structure.

### **Properties and Methods:**

- Sealed classes can have properties and methods, making them more versatile when you need to associate behavior or additional data with each subtype.

### **Instance Creation:**

- You can create instances of sealed class subtypes using constructors.

### **Pattern Matching:**

- Sealed classes are often used in languages that support pattern matching, enabling more complex and expressive control flow based on the type of the sealed class instance.

## **Enum:**

### **Fixed Set of Values:**

- Enums represent a fixed set of named values, often used to define a set of constants or options.

### **No Hierarchy:**

- Enums do not have a subtype hierarchy. Each enum value is equal to every other value within the enum.

### **No Additional Properties or Methods:**

- Enum values typically don't have associated properties or methods. They are simple, distinct values.

### **Limited Use Cases:**

- Enums are suitable for situations where you have a predefined set of constant values, such as days of the week or status codes.

```
sealed class Result {  
    data class Success(val data: String) : Result()  
    data class Error(val message: String) : Result()  
}
```

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;  
}
```

- In summary, sealed classes are more flexible and versatile, allowing you to define hierarchies with associated properties and methods, while enums are simpler and more focused on representing a fixed set of distinct values without hierarchy or additional features. The choice between them depends on the specific needs of your program or system.