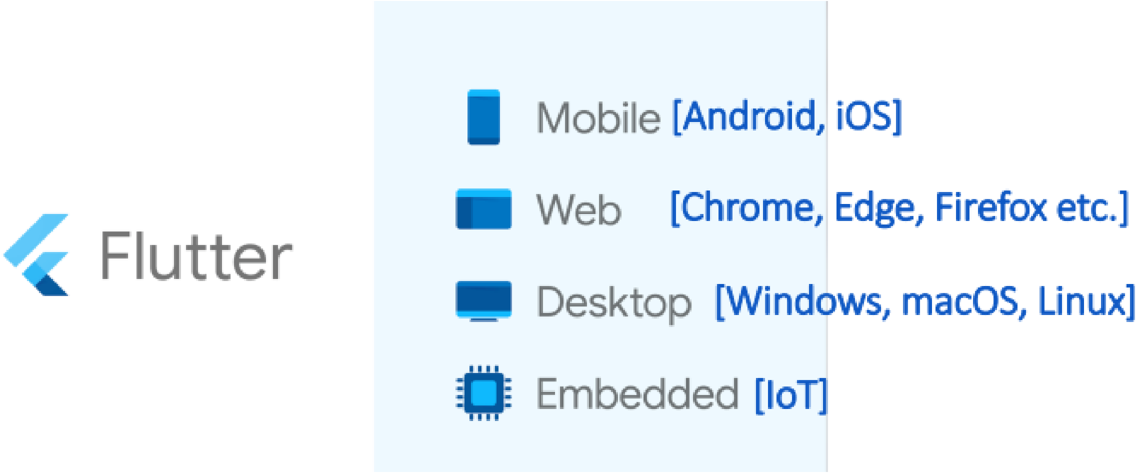
1.Give concise introduction of Flutter.

Introduction to Flutter

- **What is Flutter?**
  - Flutter is an **open-source UI framework** developed by Google.
  - It allows developers to build apps for **Android, iOS, Web, Windows, macOS, Linux, and IoT** using **a single codebase**.
- **Why Use Flutter?**
  - **Fast Development**: Features like **Hot Reload** let developers see changes instantly without restarting the app.
  - **Cross-Platform**: One codebase works on multiple platforms, reducing development time and cost.
  - **Beautiful UI**: Provides a rich set of **pre-designed widgets** following **Material Design** (for Android) and **Cupertino** (for iOS).
  - **High Performance**: Uses the **Skia 2D rendering engine** for smooth animations and fast UI rendering.
  - **Dart Language**: Flutter is built with **Dart**, a simple and powerful programming language optimized for front-end development.
- **History of Flutter**
  - Initially released in **May 2017**.
  - The first version of Flutter was called **"Sky"**, designed for Android.
  - It has since grown into a fully-featured **cross-platform development framework**.
- **How Flutter Works**
  - Unlike other frameworks that rely on **WebView** or **native components**, Flutter **directly renders** UI using its own engine.
  - This results in a more **consistent look and feel** across platforms.
- **Official Resources**
  - Official website: flutter.dev
  - Documentation: docs.flutter.dev

• Flutter is a framework and not a programming language.

- **Latest version: 3.19.5 (as on 11th April, 2024)**



Flutter

Mobile [Android, iOS]

Web [Chrome, Edge, Firefox etc.]

Desktop [Windows, macOS, Linux]

Embedded [IoT]

2.Write a short note on Widgets.

Widgets in Flutter

- **Definition**: Widgets are the building blocks of a Flutter app. Everything on the screen, such as text, buttons, and images, is a widget.
- Widgets are arranged in a **tree structure**, where each widget is inside another, forming a parent-child relationship.
- They cannot be changed directly; instead, new widgets replace the old ones when updates occur.
- There are **two types of widgets**:
  - **Stateless Widgets**: Fixed and do not change during runtime (e.g., Text, Icon, RaisedButton).
  - **Stateful Widgets**: Can update and change appearance based on user interaction (e.g., TextField, Checkbox, Slider).
- Widgets describe **what** the UI should look like rather than **how** it should be drawn.
- Every widget has properties that define its appearance and behavior.
- Widgets can be customized with **padding, margin, color, alignment, and size**.
- They follow a **hierarchical structure**, where each widget is part of a larger UI component.

- Widgets can be classified into different categories like **layout widgets, interactive widgets, styling widgets, and structural widgets**.

---

3. Write a short note on runApp() method.

❖ The runApp() Method

- `runApp()` is the entry point of a Flutter application.
- It is a function provided by the Flutter framework to launch the application.
- It takes a **widget** as an argument and makes it the root of the widget tree.
- It is **called inside the `main()` function** to start the app execution.
- The widget passed to `runApp()` is usually **`MaterialApp`** or **`CupertinoApp`**, which defines the app structure.
- It **initializes the Flutter engine** and starts rendering the UI.
- Any widget can be passed to `runApp()`, including a custom widget.
- It ensures that the app runs within the Flutter framework lifecycle.

Example usage:

```
 void main() {
  runApp(MyApp());
}
```

Example with a simple widget:

```
 void main() {
  runApp(Center(
    child: Text('Hello, Flutter!', textDirection: TextDirection.ltr),
  ));
}
```

4.Give an example of UI and corresponding widget tree.

Widget Tree in Flutter (Detailed Explanation)

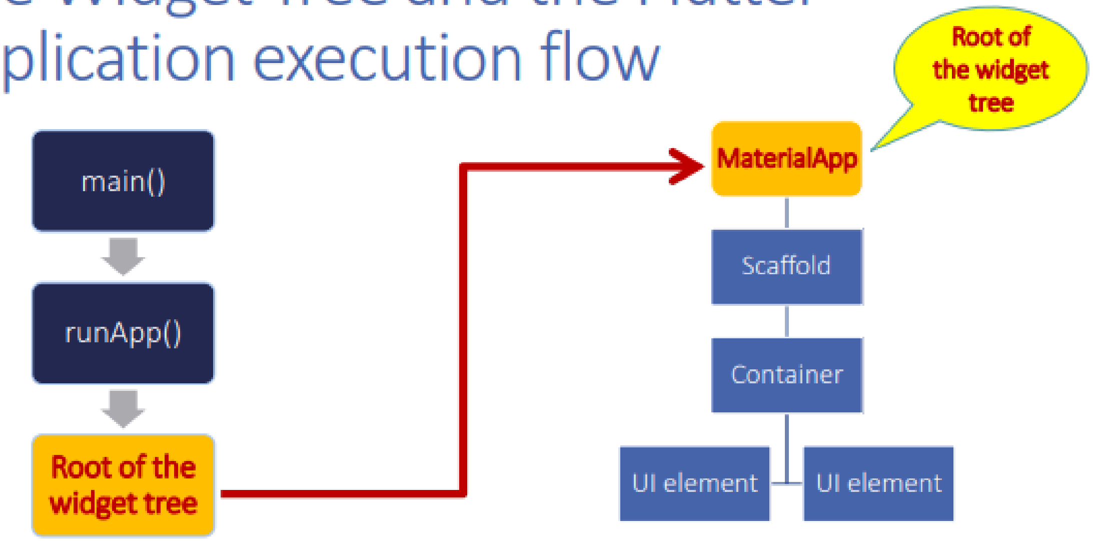
What is a Widget Tree?

In Flutter, everything you see on the screen is a **widget**. The arrangement and hierarchy of these widgets create a **Widget Tree**, which represents the structure of your UI.

A **Widget Tree** is a tree-like structure where:

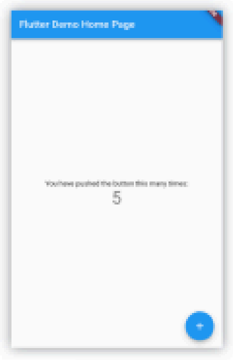
- The **root widget** is the starting point of the app (usually `MaterialApp` or `CupertinoApp`).
- **Parent widgets** contain one or more **child widgets**.
- Widgets can have multiple **nested widgets** inside them

The Widget Tree and the Flutter application execution flow



Widget Tree Example

UI

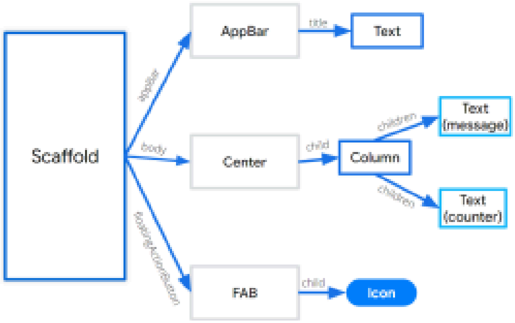


Flutter Demo Home Page

You have pushed the button this many times:

5

Breakdown



Flutter Widget Tree Structure

```
MaterialApp ← Root of the widget tree
 └─ Scaffold ← Provides app structure
     ├─ AppBar ← Displays the top navigation bar
     ├─ Body
         └─ Container ← Holds UI elements
             ├─ Row or Column ← Organizes UI elements
             ├─ Text ← Displays text
             ├─ Image ← Displays images
             ├─ Button ← Handles user interactions
             ├─ ListView ← Displays scrollable lists
             └─ Other UI elements
     └─ FloatingActionButton ← A button placed above UI
```

---

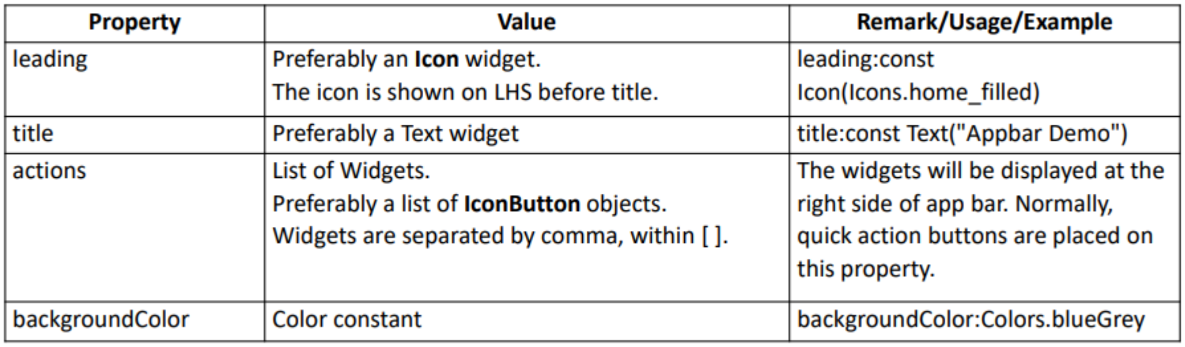
5. Explain important attributes of AppBar widget.

AppBar Widget

The `AppBar` widget is a material design toolbar that appears at the top of the screen.

Key Features:

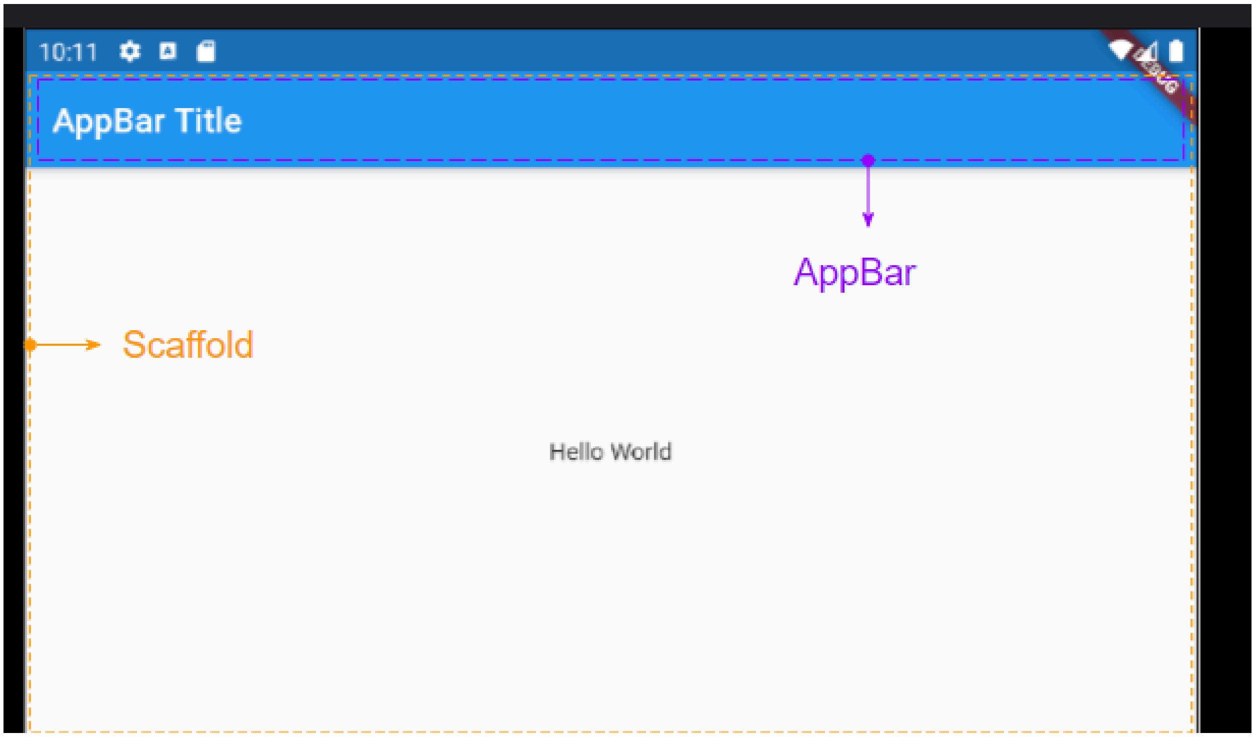
- Displays the app title, navigation, and action buttons.
- Can include icons, text, and custom widgets.
- Supports background color and elevation for styling.
- Works well with `Scaffold` to create a consistent UI.
- Can include a `leading` widget (like a back button) and `actions` (like search or settings).



Property	Value	Remark/Usage/Example
leading	Preferably an **Icon** widget. The icon is shown on LHS before title.	leading:const Icon(Icons.home_filled)
title	Preferably a Text widget	title:const Text("Appbar Demo")
actions	List of Widgets. Preferably a list of **IconButton** objects. Widgets are separated by comma, within [ ].	The widgets will be displayed at the right side of app bar. Normally, quick action buttons are placed on this property.
backgroundColor	Color constant	backgroundColor:Colors.blueGrey

Example:

```
AppBar(
  title: Text('My App'),
  backgroundColor: Colors.blue
 )
```



6.Explain important attributes of TextStyle class

What is TextStyle in Flutter?

`TextStyle` in Flutter is a class used to customize the appearance of text in a **Text widget**. It allows you to modify various text properties like **color, size, font style, weight, decoration, and spacing**.

Key Features of TextStyle

- **Changes text color** (e.g., `Colors.blue`)
- **Sets font size** (e.g., `fontSize: 20`)
- **Makes text bold or italic** (e.g., `FontWeight.bold`, `FontStyle.italic`)
- **Adds decorations** like underline or strikethrough
- **Controls spacing** between letters and word

Important Attributes of the TextStyle Class

The **TextStyle** class in Flutter is used to define the appearance of text in a widget. Below are some important attributes:

Property	Description
color	Sets the text color (e.g., `Colors.red`).
fontSize	Defines the text size in logical pixels.
fontWeight	Controls the boldness of text (e.g., `FontWeight.bold`).
fontStyle	Allows italic text using `FontStyle.italic`.
decoration	Adds text decorations like underline or strikethrough (e.g., `TextDecoration.underline`).

Example:

```
Text(
  'Hello, Flutter!',
  style: TextStyle(
    color: Colors.blue,
    fontSize: 20,
    fontWeight: FontWeight.bold,
    fontStyle: FontStyle.italic,
    decoration: TextDecoration.underline,
  ),
)
```

Output :- ***Hello, Flutter!***

---

7. Explain important attributes of InputDecoration class (4-5 marks)

InputDecoration

`InputDecoration` is a class in Flutter that helps style `TextField` and `TextFormField`. It allows customization of borders, labels, icons, hint text, and more to improve user input experience.

Important Attributes of InputDecoration

Attribute	Description
labelText	Displays a label inside the `TextField`. Moves up when text is entered.

hintText Shows a placeholder inside the `TextField`. Disappears when text is entered.

icon Adds an icon outside the input field before the text.

prefixIcon Adds an icon inside the input field before the text.

suffixIcon Adds an icon inside the input field after the text. Useful for "eye" icons in password fields.

border Sets the border style of the `TextField` (e.g., `OutlineInputBorder`).

filled Fills the background of the `TextField` with a color when set to `true`.

Example Code

```
TextField(
  decoration: InputDecoration(
    labelText: "Username",
    hintText: "Enter your username",
    icon: Icon(Icons.person),
    border: OutlineInputBorder(),
  ),
)
```

Output: A `TextField` with:
✔ **Label:** "Username"
✔ **Hint:** "Enter your username"
✔ **Icon:** Person icon
✔ **Border:** Outlined text field

---

8. Explain the following properties of TextField widget: controller, obscureText, maxLength, keyboardType, readOnly

Introduction to TextField in Flutter

`TextField` is a Flutter widget that allows users to enter text input. It is commonly used in forms, login screens, search bars, and other input fields. It provides various properties to control user input, such as validation, formatting, and styling.

Properties of the TextField Widget

controller – This property uses a `TextEditingController` to manage and retrieve the text entered in a `TextField`. It allows accessing, modifying, or clearing the text programmatically.

Example:

```
 TextEditingController myController = TextEditingController();

TextField(controller: myController);
```

1. You can get the text using `myController.text`.

obscureText – This property hides the text input, making it useful for password fields. When set to `true`, the entered text is replaced with dots (`•`).

Example:

```
TextField(obscureText: true);
```

2. This will hide the input characters.

maxLength – It limits the number of characters that can be entered in the `TextField`. Once the limit is reached, further input is restricted.

Example:

```
TextField(maxLength: 10);
```

3. Users can enter only **10 characters** in the field.

keyboardType – This defines the type of keyboard displayed for input. For example, you can set it to `TextInputType.number` for numeric input or `TextInputType.emailAddress` for email input.

Example:

```
TextField(keyboardType: TextInputType.number);
```

4. This will show a **numeric keyboard** instead of a regular one.

readOnly – When set to `true`, the `TextField` becomes non-editable, meaning users can’t type anything in it. This is useful when displaying predefined values.

Example:

```
 TextField(readOnly: true, controller: TextEditingController(text: "Read-Only Text"));
```

5. The field will display `"Read-Only Text"`, but users cannot change it.

---

9. How can we retrieve value from a TextField? (4 marks)

How to Retrieve Value from a TextField in Flutter?

In Flutter, the **TextEditingController** is used to retrieve the text entered in a `TextField`. The `.text` property of the controller allows access to the input. This is useful for handling user inputs in forms, login screens, and search fields.

Key Points:

- `TextEditingController` stores the text entered in a `TextField`.
- `.text` property is used to retrieve the current value.
- The controller helps in modifying or clearing the text programmatically.
- It is commonly used in form validation and real-time updates.
- Helps in managing multiple text fields efficiently.

Example:

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


void main() {

  runApp(MyApp());

}


class MyApp extends StatelessWidget {
```

```dart
  final TextEditingController myController = TextEditingController();


  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Column(
          children: [
            TextField(controller: myController),
            ElevatedButton(
              onPressed: () {
                print(myController.text); // Retrieve text
              },
              child: Text("Get Value"),
            ),
          ],
        ),
      ),
    );
  }
}
```

Explanation:

- The user types in the `TextField`.
- Clicking the button retrieves the text using `myController.text`.
- The text is printed in the console.

---

10. Explain Row and Column widgets. (4-5 marks) 48-53

Row and Column Widgets in Flutter

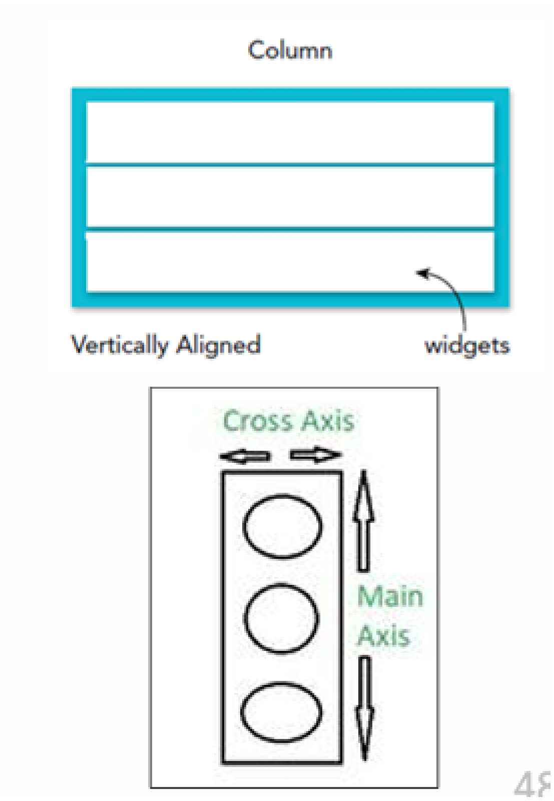
In Flutter, **Row** and **Column** are layout widgets used to arrange multiple child widgets **horizontally** and **vertically**, respectively.

---

1. Row Widget

The **Row** widget arranges its children **horizontally** in a single line.

- The **Row** widget arranges child widgets **horizontally** from left to right.
- It does **not scroll**, so if the content overflows, an error occurs.
- Uses `mainAxisAlignment` to align children horizontally.
- Uses `crossAxisAlignment` to align children vertically.
- Each child inside a `Row` gets **equal space** unless wrapped with `Expanded` or `Flexible`.
- It takes **full width** of the screen but height depends on its children.

- 

-

Example:

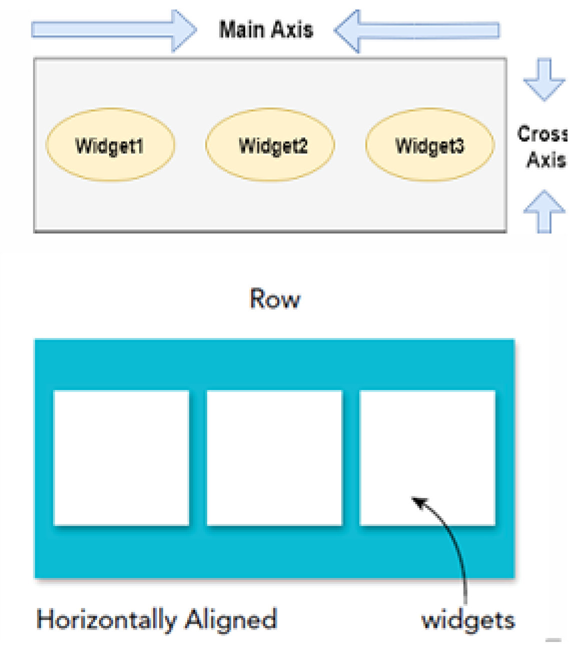
```
Row(
  mainAxisAlignment: MainAxisAlignment.center, // Align children in center
  children: [
    Text("A"),
    Text("B"),
    Text("C"),
  ],
)
```

This will display: **A B C** in a single line.

2. Column Widget

The **Column** widget arranges its children **vertically** in a single column.

- The **Column** widget arranges child widgets **vertically** from top to bottom.
- It automatically **scrolls** if the content overflows the screen.
- Uses `mainAxisAlignment` to align children vertically.
- Uses `crossAxisAlignment` to align children horizontally.
- Each child inside a `Column` gets **equal space** unless wrapped with `Expanded` or `Flexible`.
- It takes **full height** of the screen but width depends on its children.



Example:

Column(

```
  mainAxisAlignment: MainAxisAlignment.center, // Align children in center
  children: [
    Text("A"),
    Text("B"),
    Text("C"),
  ],
)
```

This will display:
A
B
C
(one below the other).

---

11. Discuss Stateless vs. Stateful widgets. (4-5 marks) 54

Stateless vs. Stateful Widgets in Flutter

Feature	Stateless Widget	Stateful Widget
Definition	A widget that **does not change** once built.	A widget that **can change** dynamically during runtime.
State Changes	Cannot change state after creation.	Can change state using `setState()`.

Use Case	Used for **static UI** (e.g., `Text`, `Icons`).	Used for **interactive UI** (e.g., `Forms`, `Buttons`).
Extends	`StatelessWidget` class.	`StatefulWidget` class and has a separate `State` class.
Rebuilds	Built **only once**.	Rebuilds whenever `setState()` is called.
Performance	**Faster** and uses fewer resources.	**Slightly slower** due to frequent UI updates.
Memory Usage	Takes up less memory since no state is stored.	Uses more memory as state is stored in the widget.
Widget Lifecycle	Created and removed only when needed.	Exists throughout the app runtime with state updates.
Example	A **static** text or icon.	A **counter** that updates when a button is pressed.

Example of Stateless Widget

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


class MyStatelessWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text("I am Stateless!"),
    );
  }
}
```

✅ **Output:** Displays a **static** message that does not change.

Example of Stateful Widget

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


class MyStatefulWidget extends StatefulWidget {
  @override
  _MyStatefulWidgetState createState() => _MyStatefulWidgetState();
}


class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  int count = 0;


  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text("Count: $count"),
        ElevatedButton(
          onPressed: () {
            setState(() {
              count++; // Updates the UI
            });
          },
          child: Text("Increment"),
        ),
      ],
    );
  }
}
```

✅ **Output:** Displays a counter that **increments** when the button is pressed.

---

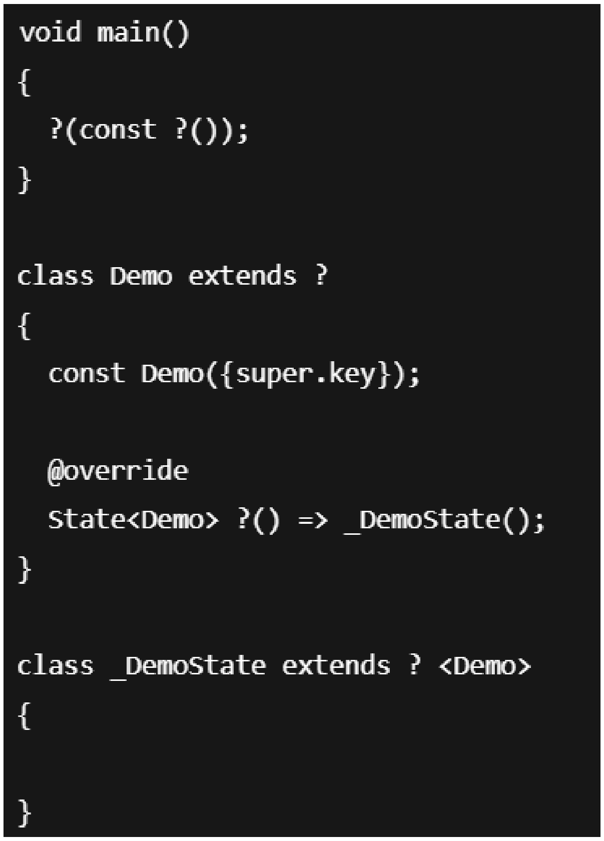
12. Write steps to create and use Stateful widgets. (4-6 marks) 56

---

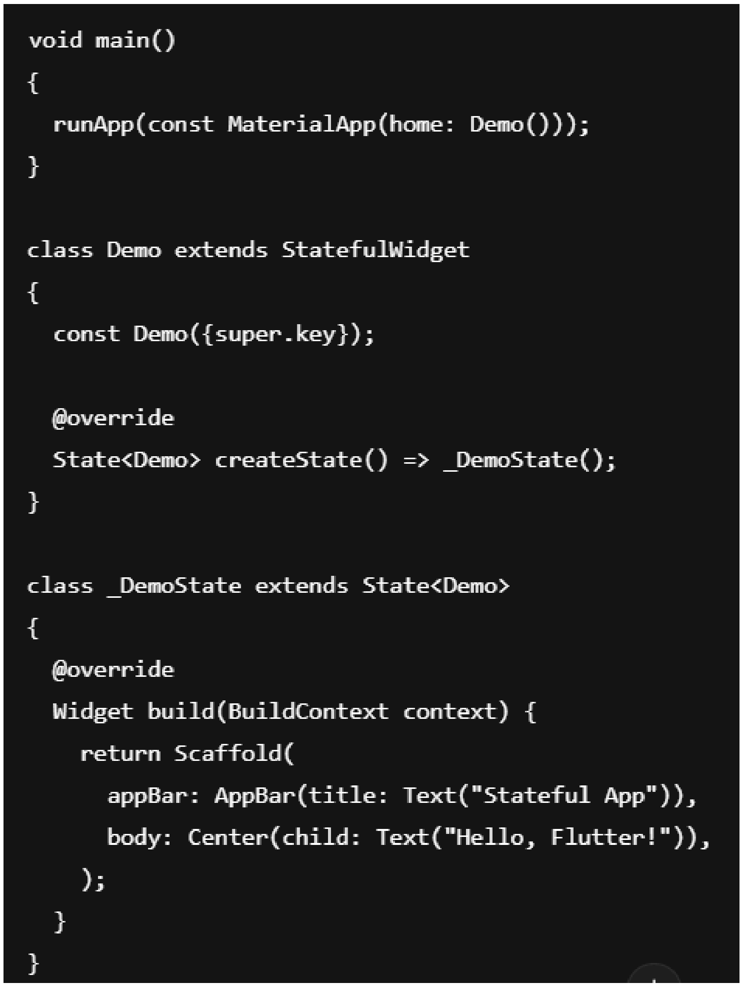
13. Replace ? in the following code with the correct words to make it a Stateful application.

```
void main()

{

  ?(const ?());

}


class Demo extends ?

{

  const Demo({super.key});


  @override

  State<Demo> ?() => _DemoState();

}


class _DemoState extends ? <Demo>

{


}
```

Answer:



```
void main()
{
  ?(const ?());
}

class Demo extends ?
{
  const Demo({super.key});

  @override
  State<Demo> ?() => _DemoState();
}

class _DemoState extends ? <Demo>
{

}
```



```
void main()
{
  runApp(const MaterialApp(home: Demo()));
}

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

  @override
  State<Demo> createState() => _DemoState();
}

class _DemoState extends State<Demo>
{
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Stateful App")),
      body: Center(child: Text("Hello, Flutter!")),
    );
  }
}
```

Replaced Words:

Symbol (?)	**Replaced with**

`?` in `main()`	`runApp`
`?` in `const ?()`	`MaterialApp(home: Demo())`
`?` in `class Demo extends ?`	`StatefulWidget`
`?` in `State<Demo> ?()`	`createState`
`?` in `class _DemoState extends ?<Demo>`	`State`

Explanation:

1. **`runApp()`** starts the Flutter app.
2. **`MaterialApp(home: Demo())`** sets `Demo` as the main widget.
3. **`StatefulWidget`** is used because the UI can change.
4. **`createState()`** links the widget with its state class.
5. **`State<Demo>`** is the base class that manages state.

---

14. Explain the setState() method. (4-5 marks) 63

setState() Method in Flutter

The `setState()` method is used in **Stateful Widgets** to update the UI dynamically when the widget's state changes. It notifies the Flutter framework that the state has changed, triggering a **rebuild** of the widget.

Key Features of setState()

1. **Triggers UI Updates** – When called, Flutter **rebuilds** the widget to reflect the latest state.
2. **Works Only in Stateful Widgets** – `setState()` is available only inside the `State` class of a **StatefulWidget**.
3. **Requires a Callback Function** – The state-changing logic must be placed inside the `setState()` method.
4. **Efficient UI Updates** – It only rebuilds the affected part of the UI, improving performance.
5. **Does Not Update Immediately** – The UI updates **after** the method completes execution.

Example of `setState()` Usage

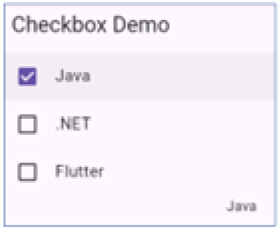
```
void incrementCounter() {

  setState(() {

    count++; // Updating state

  });

}
```

Explanation:

- `setState()` **is used to increase the count and update the UI.**
- **The UI rebuilds automatically to reflect the new value.**

---

15. Explain any five import properties/callback of CheckboxListTile. (4 5 marks)65



Checkbox Demo

☑ Java

□ .NET

□ Flutter

Java

CheckboxListTile in Flutter

The `CheckboxListTile` widget is a **combination of a checkbox and a label** (title). It allows users to **toggle selections** in a structured list format. This widget is useful for **settings screens, forms, and preference selections** where users need to enable or disable options easily.

Important Properties/Callbacks of CheckboxListTile

1.value – Defines whether the checkbox is **checked (true)** or **unchecked (false)**.

```
CheckboxListTile(value: true, onChanged: (bool? value) {}, title: Text("Option"));
```

2.onChanged – A callback function that gets triggered when the user **taps** the checkbox.

```
onChanged: (bool? newValue) {
  setState(() {
    isChecked = newValue!;
  });
}
```

3.title – Used to display the main text label beside the checkbox.

```
 title: Text("Enable Notifications"),
```

4.subtitle – Displays a **secondary label** below the main title.

```
 subtitle: Text("Receive updates via email"),
```

5.**controlAffinity** – Defines the position of the checkbox relative to the text.

```
 controlAffinity: ListTileControlAffinity.leading, // Checkbox before text
```

6.secondary – Adds a widget (e.g., an icon) before or after the checkbox.

```
 secondary: Icon(Icons.settings),
```

7.selected – Highlights the text and icon when set to `true`.

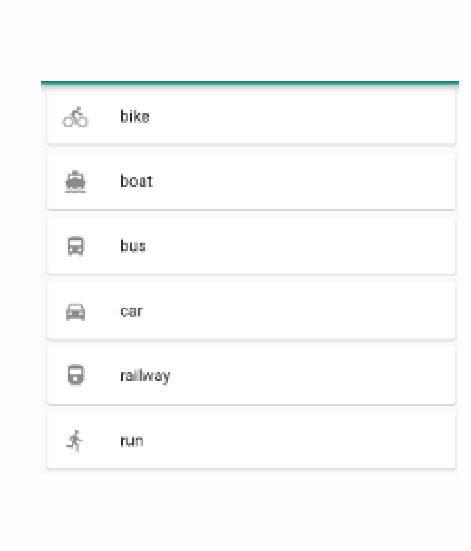
```
 selected: true,
```

---

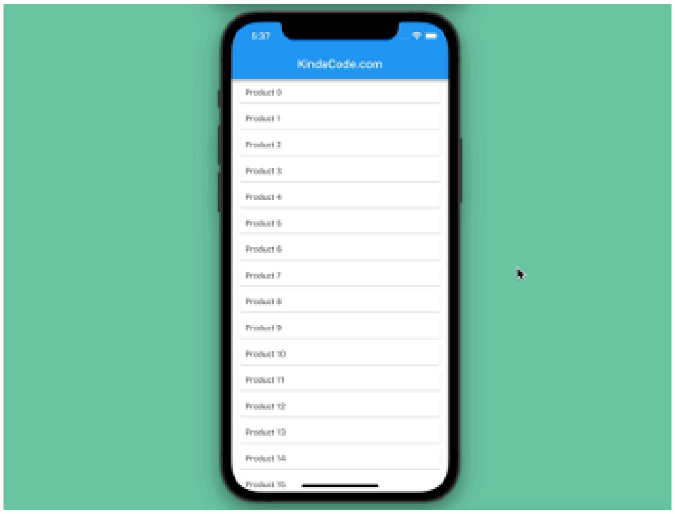
16. Write a short-note on ListView widget. (4-5 marks) 87, 90

Short Note on ListView Widget

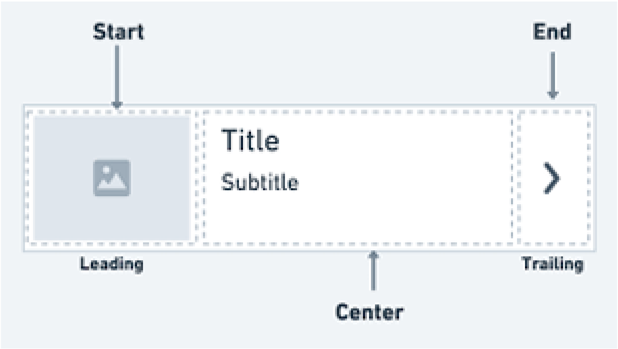
The `ListView` widget in Flutter is used to create a **scrollable list** of items, which can be **fixed** or **dynamic** in length. It is commonly used for displaying multiple widgets in a vertical arrangement.



bike
boat
bus
car
railway
run



5:37
KindaCode.com
Product 0
Product 1
Product 2
Product 3
Product 4
Product 5
Product 6
Product 7
Product 8
Product 9
Product 10
Product 11
Product 12
Product 13
Product 14
Product 15



Key Points:

1. **Types of ListView:**
   - `ListView()` – Creates a static list.
   - `ListView.builder()` – Creates list items dynamically when needed.
   - `ListView.separated()` – Adds separators between list items.

   - `ListView.custom()` – Offers full customization.

2. **Fixed vs. Dynamic List:**

   - **Fixed List:** Created using `ListView(children: [])`.
   - **Dynamic List:** Uses `ListView.builder()` to load items as they appear on screen, improving performance.

3. **Commonly Used with ListTile:**

   - `ListTile` is a built-in widget that helps display structured list items with icons, titles, and subtitles.

Example of ListView:

```
ListView(
  children: [
    ListTile(title: Text("Bike")),
    ListTile(title: Text("Car")),
    ListTile(title: Text("Bus")),
  ],
)
```

This creates a **scrollable list** with predefined items.

Example of ListView.builder():

```
ListView.builder(
  itemCount: 5,
  itemBuilder: (context, index) {
    return ListTile(title: Text("Item $index"));
  },
)
```

---

17. Discuss Navigation with Named Routes in brief.

(4-5 marks) 116 117

Navigation with Named Routes

- **Used when navigating to a specific screen instead of just moving forward or backward.**
- **Helps in managing multiple screens efficiently in large apps.**
- **Defined inside `MaterialApp` using the `routes` property.**
- **Each route has a unique name (path) and target widget.**
- **Uses `Navigator.pushNamed()` to navigate between screens.**
- **Uses `Navigator.pop()` to return to the previous screen.**
- **✔ Improves Code Readability – No need to create `MaterialPageRoute` every time.**
- **✔ Centralized Route Management – All routes are defined in one place.**
- **✔ Useful in Large Apps – Avoids hardcoding screen navigation logic everywhere.**
- **✔ Easy to Modify – Just update the route in `MaterialApp` instead of changing code in multiple places.**

```
MaterialApp(
  home: const Screen1(),
  routes: {
    "/s1": (context) => const Screen1(),
    "/s2": (context) => const Screen2(),
    "/s3": (context) => const Screen3(),
    "/s4": (context) => const Screen4(),
  },
);
```

Passing Data with Named Routes

1.On Sender Screen:

Use `arguments` while calling `pushNamed()`

```
Navigator.pushNamed(context, "/s2", arguments: "Hello, Flutter!");
```

2.On Receiver Screen:

Retrieve the data using `ModalRoute`

String data = ModalRoute.of(context)!.settings.arguments.toString();

OR

int number = ModalRoute.of(context)!.settings.arguments as int;

Key Points:

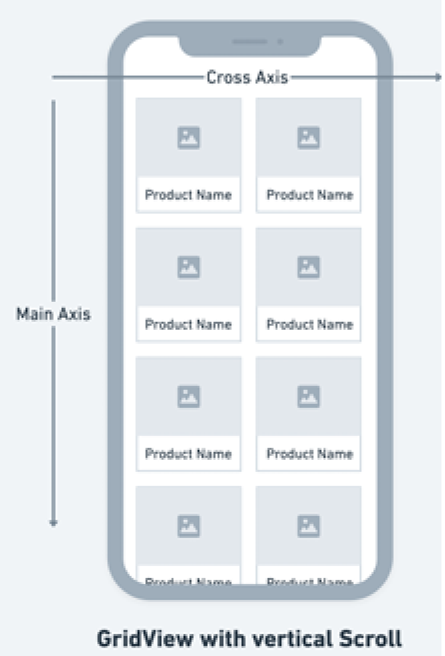
✔ Use `arguments` in `pushNamed()` to pass data
✔ Retrieve data in the target screen using `ModalRoute.of(context)`
✔ Use `.toString()` for strings and `as Type` for other data types

---

18. Explain GridView.builder().(4-5 marks) 138 141

GridView.builder()

`GridView.builder()` is a dynamic way to create grid layouts in Flutter. Unlike `GridView.count()`, it builds items lazily, loading only what is visible on the screen.



GridView with vertical Scroll

Key Points:

- ✅ **Efficient for large datasets** – loads only visible items to optimize performance.
- ✅ **Uses lazy loading**, meaning it builds items as needed.
- ✅ **Uses `SliverGridDelegateWithFixedCrossAxisCount`** to define the grid layout.
- ✅ **`crossAxisCount`** determines the number of columns.
- ✅ **`itemBuilder`** is called for each grid item dynamically.
- ✅ **Supports spacing and alignment adjustments** using `mainAxisSpacing` and `crossAxisSpacing`.
- ✅ **Flexible and customizable**, allowing dynamic content like images, cards, or buttons.

Common Attributes of GridView.builder()

1. **`itemCount`** – Defines the number of items to display.
2. **`gridDelegate`** – Controls the grid layout and spacing (Required).

3. `itemBuilder` – Creates grid items dynamically (Required).
4. `scrollDirection` – Sets scrolling direction (default: vertical).

Syntax:

```
GridView.builder(
  gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
    crossAxisCount: 2, // Number of columns
    crossAxisSpacing: 10, // Space between columns
    mainAxisSpacing: 10, // Space between rows
  ),
  itemCount: 10, // Total items
  itemBuilder: (context, index) {
    return Container(
      color: Colors.blue,
      child: Center(child: Text("Item $index")),
    );
  },
);
```

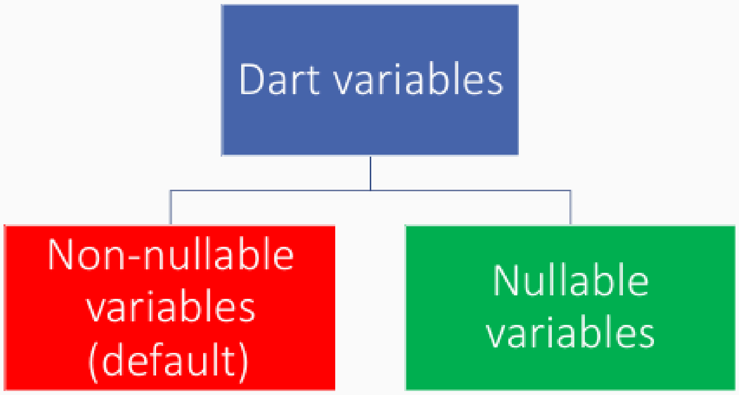
---

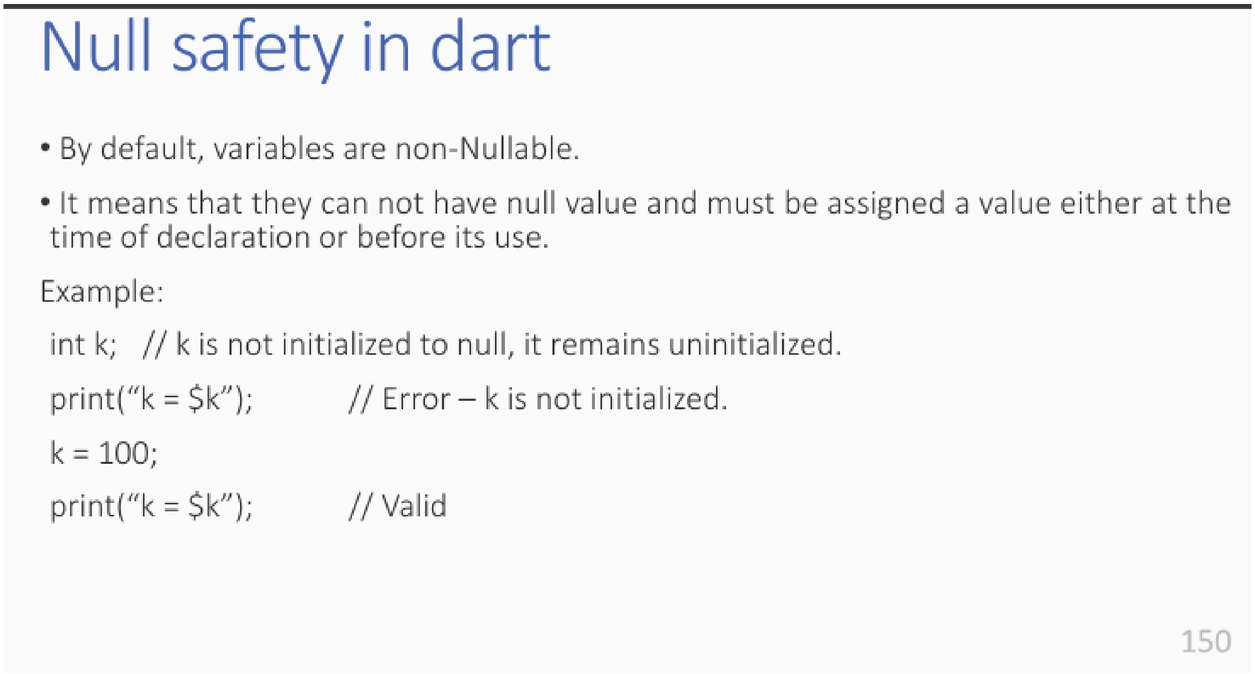
19. Explain Null Safety in Dart. (4-5 marks) 148, 150, 151

Null safety in dart

- Null safety is a technique to prevent an error which occurs due to accessing a variables/property/method which has/returns null value.
- A **null dereference error** occurs when you
  - access a variable with null value
  - access a property of a widget/object which has null value
  - call a method on a widget/an object which returns null value.
- With null safety, the Dart compiler detects these potential errors at **compile time**.

Null safety in dart



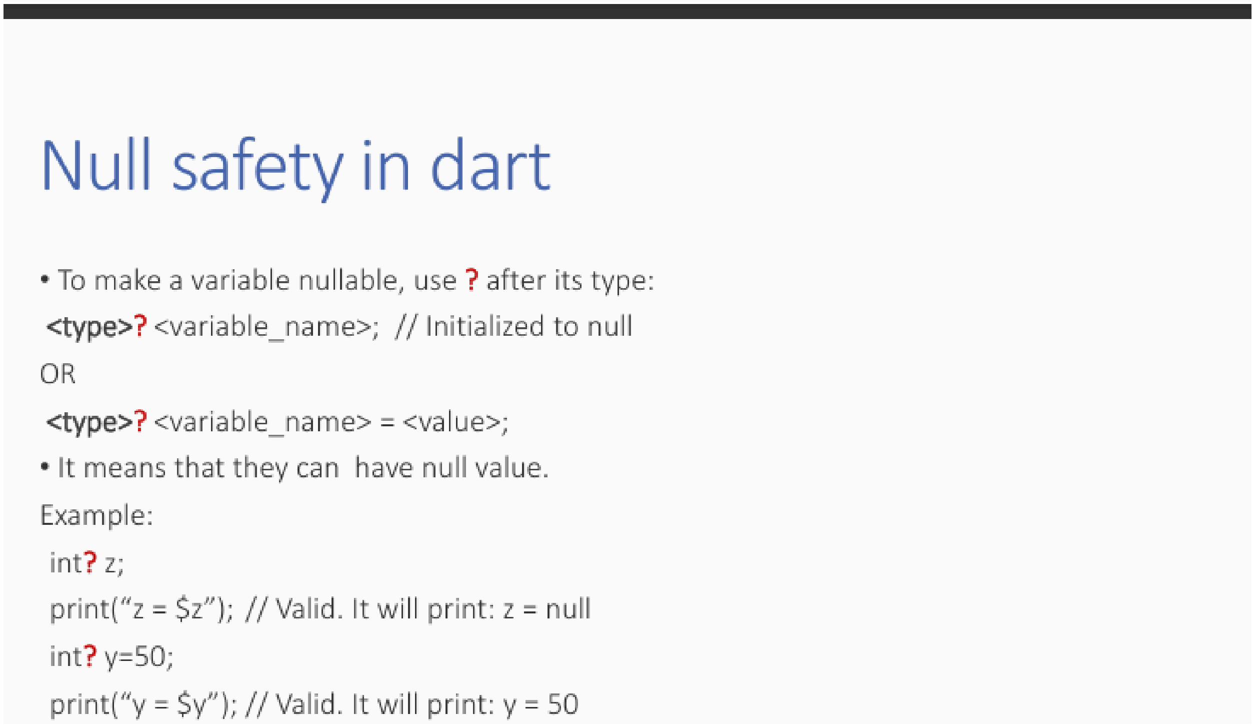


Null safety in dart

- By default, variables are non-Nullable.
- It means that they can not have null value and must be assigned a value either at the time of declaration or before its use.

Example:

```
int k;   // k is not initialized to null, it remains uninitialized.
print("k = $k");          // Error – k is not initialized.
k = 100;
print("k = $k");          // Valid
```



Null safety in dart

- To make a variable nullable, use **?** after its type:

```
<type>? <variable_name>;  // Initialized to null
```

OR

```
<type>? <variable_name> = <value>;
```

- It means that they can have null value.

Example:

```
int? z;
print("z = $z");  // Valid. It will print: z = null
int? y=50;
print("y = $y"); // Valid. It will print: y = 50
```

20. Differentiate between const and final in Dart. (4-5 marks) 155

Difference Between `const` and `final` in Dart

Feature	`const`	`final`
Mutability	Immutable at compile time	Immutable at runtime
Assignment	Must be assigned a value at compile time	Can be assigned a value at runtime
Reassignment	Not allowed	Not allowed
Usage	Used for constant values known at compile time	Used for values that may be set at runtime but remain unchanged
Memory Allocation	Allocated at compile time	Allocated at runtime
Usage in Objects	Creates a completely immutable object	Can reference a mutable object
Inheritance	Cannot be overridden in subclasses	Can be overridden if used inside a class
Performance	Faster execution due to compile-time evaluation	Slightly slower due to runtime allocation

Example (Valid)	`const pi = 3.14;`	`final date = DateTime.now();`
Example (Invalid)	`const x = DateTime.now();` ❌	`final y; y = 10;` ✅ **(only once)**

Deferred Loading of a Library in Dart

Deferred loading (also known as **lazy loading**) is a technique in Dart that allows a library to be loaded only when needed, rather than at the start of the program. This helps improve the app's startup time and reduces memory usage.

Key Points:

- **Used for large libraries** to reduce initial load time.
- **The `deferred` keyword** is used when importing the library.
- **`loadLibrary()` method** is used to load the library at runtime.
- Improves **performance and efficiency** by loading resources only when required.

Example:

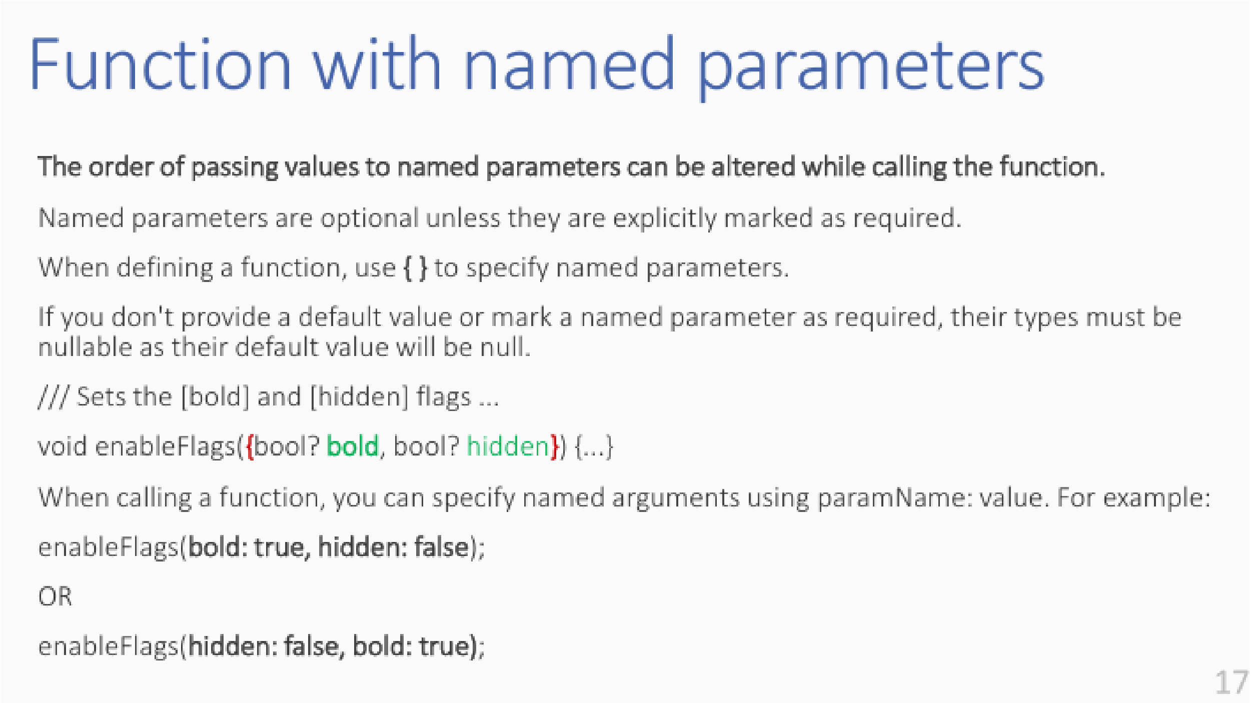
```
import 'my_library.dart' deferred as myLib;

void main() {
  print("App Started");

  // Load the library only when needed
  myLib.loadLibrary().then((_) {
    myLib.someFunction(); // Call function after loading
  });
}
```

Advantages:

✅ Faster app startup
✅ Saves memory
✅ Reduces unnecessary resource loading

---

22. Discuss function with named parameters. (4-5 marks) 175 176



Function with named parameters

The order of passing values to named parameters can be altered while calling the function.

Named parameters are optional unless they are explicitly marked as required.

When defining a function, use **{ }** to specify named parameters.

If you don't provide a default value or mark a named parameter as required, their types must be nullable as their default value will be null.

```
/// Sets the [bold] and [hidden] flags ...
void enableFlags({bool? bold, bool? hidden}) {...}
```

When calling a function, you can specify named arguments using paramName: value. For example:

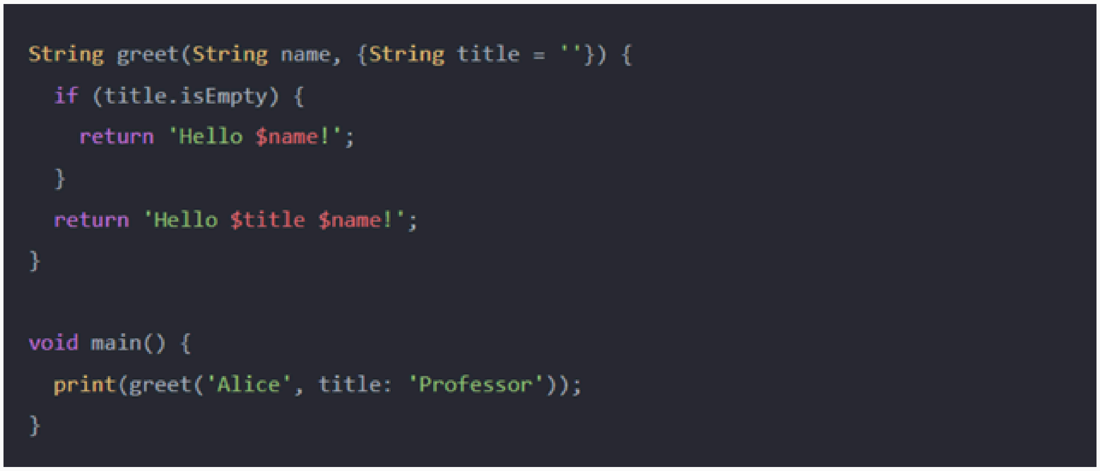
```
enableFlags(bold: true, hidden: false);
```

OR

```
enableFlags(hidden: false, bold: true);
```

17

Function with named parameters

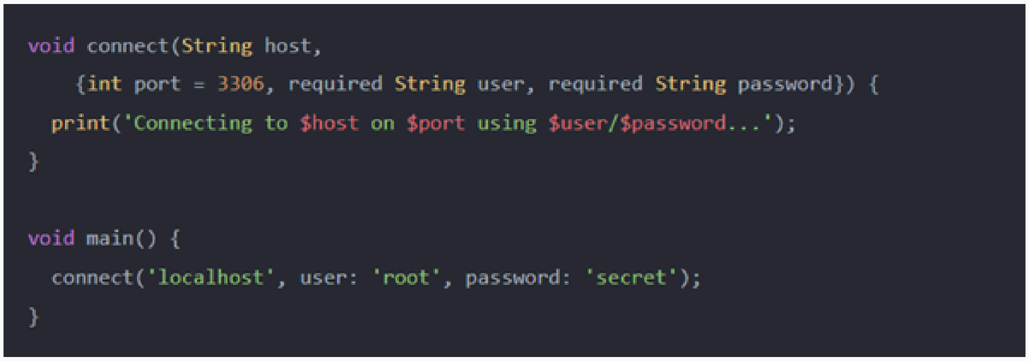


```
String greet(String name, {String title = ''}) {
  if (title.isEmpty) {
    return 'Hello $name!';
  }
  return 'Hello $title $name!';
}

void main() {
  print(greet('Alice', title: 'Professor'));
}
```

Function with required parameters

To make a named parameter required, you add the **required** keyword in front and remove the default value.

The following example makes the user and password parameters required:



```
void connect(String host,
    {int port = 3306, required String user, required String password}) {
  print('Connecting to $host on $port using $user/$password...');
}

void main() {
  connect('localhost', user: 'root', password: 'secret');
}
```

23. Difference Between Hot Reload, Hot Restart, and Full Restart

Feature	Hot Reload	Hot Restart	Full Restart
Definition	Updates UI without restarting the app	Restarts the app but keeps the initial state	Completely restarts the app and clears all states
Code Changes Reflected	UI changes, widget tree updates	UI, state reset, and global variables	Entire app reloaded from scratch
Global Variables	Not reset	Reset	Reset
Time Taken	Fastest	Medium	Slowest
Use Case	UI changes like text, colors, layouts	Logic changes like new variables, methods	Major changes in dependencies or configurations
State Preservation	Maintains app state	Clears app state	Clears app state and memory
Rebuild Required?	No	Yes	Yes

Example Use	Changing button color	Adding new function	Updating dependencies

🔹 **Hot Reload** → Best for UI changes.
🔹 **Hot Restart** → Use when changing business logic.
🔹 **Full Restart** → Needed for deep changes like dependency updates.

Q.24: I need to get the text which the user enters in a TextField named tf. I use the following code. Replace A, B, and C with appropriate words to make the code correct.

```
TextEditingController tec = A();

TextField tf = TextField(B: tec);

String data = C.text;
```

Answer:

```
TextEditingController tec = TextEditingController();  // A

TextField tf = TextField(controller: tec);           // B

String data = tec.text;                              // C
```

Explanation:

- **A** → `TextEditingController`: Creates an instance to control the text field.
- **B** → `controller`: Assigns the controller to the TextField.
- **C** → `tec`: Refers to the controller to access the entered text.

Q.25: I need to show a DatePicker with the following requirements:

1. The earliest selectable date should be **1st April 2025**.
2. The last selectable date should be **15th May 2025**.
3. The user must respond to the DatePicker.
4. Instead of the **OK** button, it should display **"Book my appointment on selected date."**

Replace **$** in the following code with correct property names.

Corrected Code:

```
Future<void> _selectDate(BuildContext context) async {
  final DateTime? picked = await showDatePicker(
    context: context,
    initialDate: DateTime.now(),
    firstDate: DateTime(2025, 4),  // $
    lastDate: DateTime(2025, 5, 15),  // $
    helpText: "Select date of appointment",
    confirmText: "Book my appointment on selected date",  // $
    cancelText: "I will do it later",
    barrierDismissible: false,  // $
  );
}
```

Explanation:

- **firstDate** → Sets the minimum selectable date to **1st April 2025**.
- **lastDate** → Sets the maximum selectable date to **15th May 2025**.
- **confirmText** → Customizes the **OK** button text.
- **barrierDismissible** → Ensures the user cannot dismiss the picker without selecting a date.

Q.26: How will you use a group of `CheckboxListTile` widgets to list food items with prices? Also, explain how to manage the total price of selected items.

Using `CheckboxListTile` for Food Selection

To create a screen listing five food items with checkboxes and manage the total price, we can use `CheckboxListTile` widgets inside a `Column`. A state variable will track the selected items and update the total price dynamically.

Simple Code:

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


void main() {
  runApp(MaterialApp(home: FoodScreen()));
}


class FoodScreen extends StatefulWidget {
  @override
  _FoodScreenState createState() => _FoodScreenState();
}

class _FoodScreenState extends State<FoodScreen> {
  Map<String, int> foodItems = {
    "Samosa": 40,
    "Tea/Coffee": 10,
    "Ice-cream": 40,
    "Cold drink": 30,
  };

  Map<String, bool> selectedItems = {
    "Samosa": false,
    "Tea/Coffee": false,
    "Ice-cream": false,
    "Cold drink": false,
  };

  int totalPrice = 0;

  void updatePrice() {
    totalPrice = foodItems.entries
```

```
      .where((e) => selectedItems[e.key]!)
      .map((e) => e.value)
      .fold(0, (sum, price) => sum + price);
  setState(() {});
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text("Food Selection")),
    body: Column(
      children: [
        ...foodItems.keys.map((item) => CheckboxListTile(
              title: Text("$item (Rs. ${foodItems[item]})"),
              value: selectedItems[item],
              onChanged: (value) {
                selectedItems[item] = value!;
                updatePrice();
              },
            )),
        Text("Total: Rs. $totalPrice",
            style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold)),
    ],
   ),
  );
 }
}
```

Explanation:

✔ `CheckboxListTile` → Displays food items with price.
✔ `selectedItems` **Map** → Tracks selected checkboxes.
✔ `updatePrice()` → Calculates the total price dynamically.
✔ **User-friendly UI** → Updates price when items are selected/unselected. 🚀

---

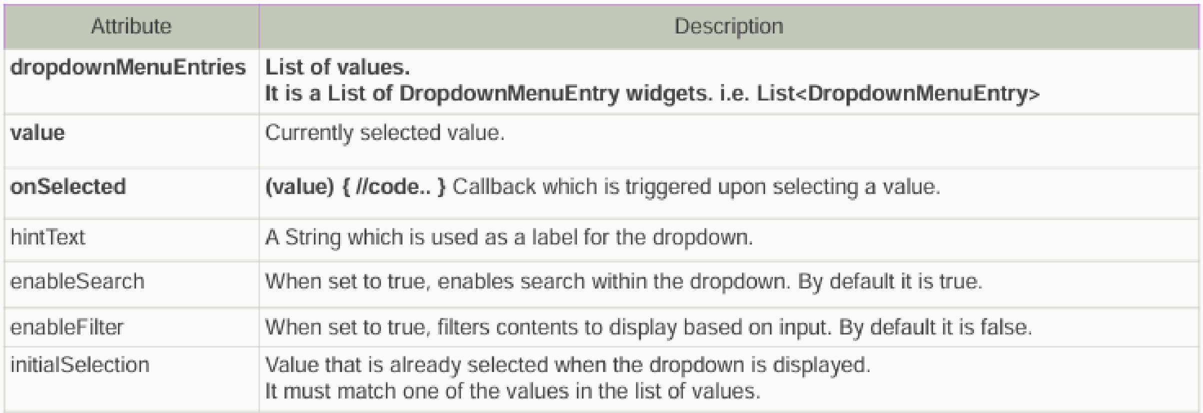
27. Explain important properties of DropdownMenu widget. 70

DropDownMenu

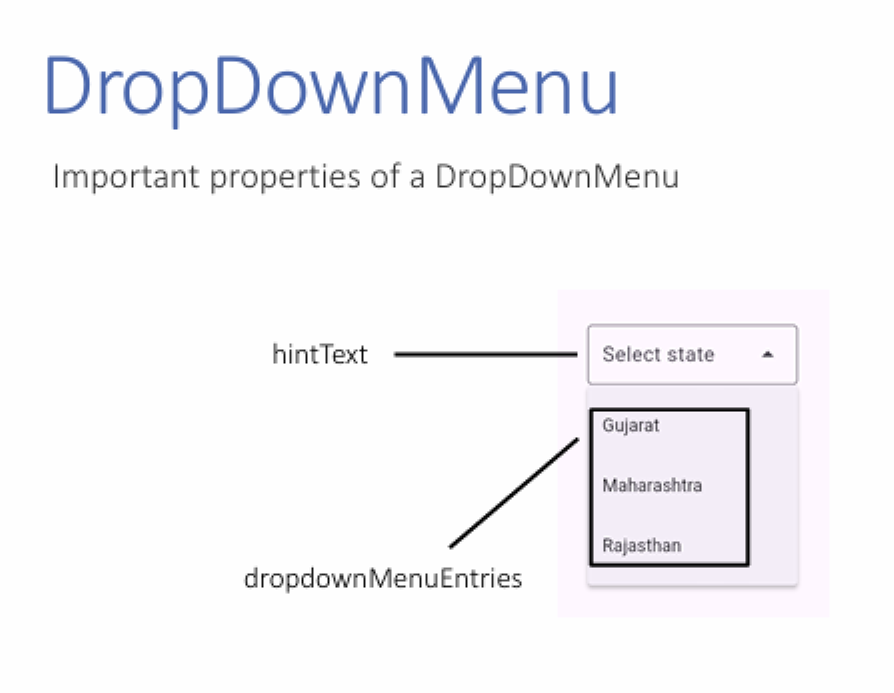
- A better alternative to group of radiobuttons.
- It allows the user to select a single value from the list of the values.
- The **onChanged()** callback is triggered when a value is selected.
- The **value** property of the widget returns the current selection.

DropDownMenu

Important properties of a DropDownMenu



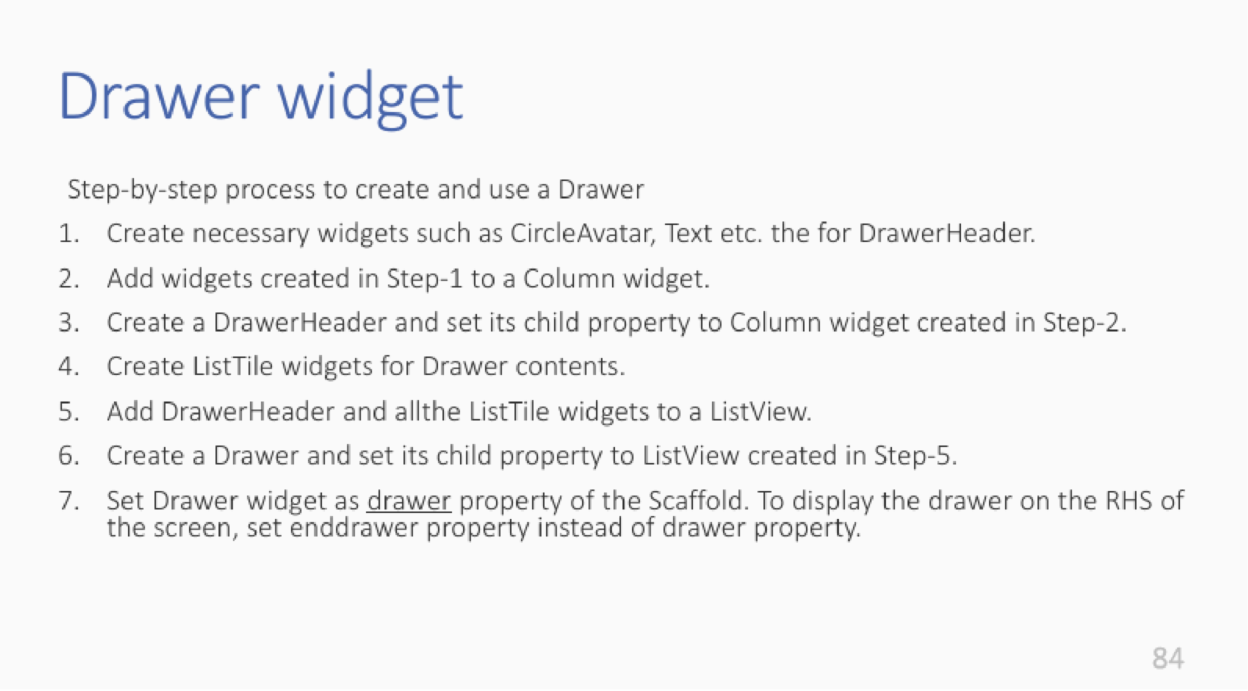
Attribute	Description
dropdownMenuEntries	**List of values.** **It is a List of DropdownMenuEntry widgets. i.e. List<DropdownMenuEntry>**
value	Currently selected value.
onSelected	**(value) { //code.. }** Callback which is triggered upon selecting a value.
hintText	A String which is used as a label for the dropdown.
enableSearch	When set to true, enables search within the dropdown. By default it is true.
enableFilter	When set to true, filters contents to display based on input. By default it is false.
initialSelection	Value that is already selected when the dropdown is displayed. It must match one of the values in the list of values.



DropDownMenu

Important properties of a DropDownMenu

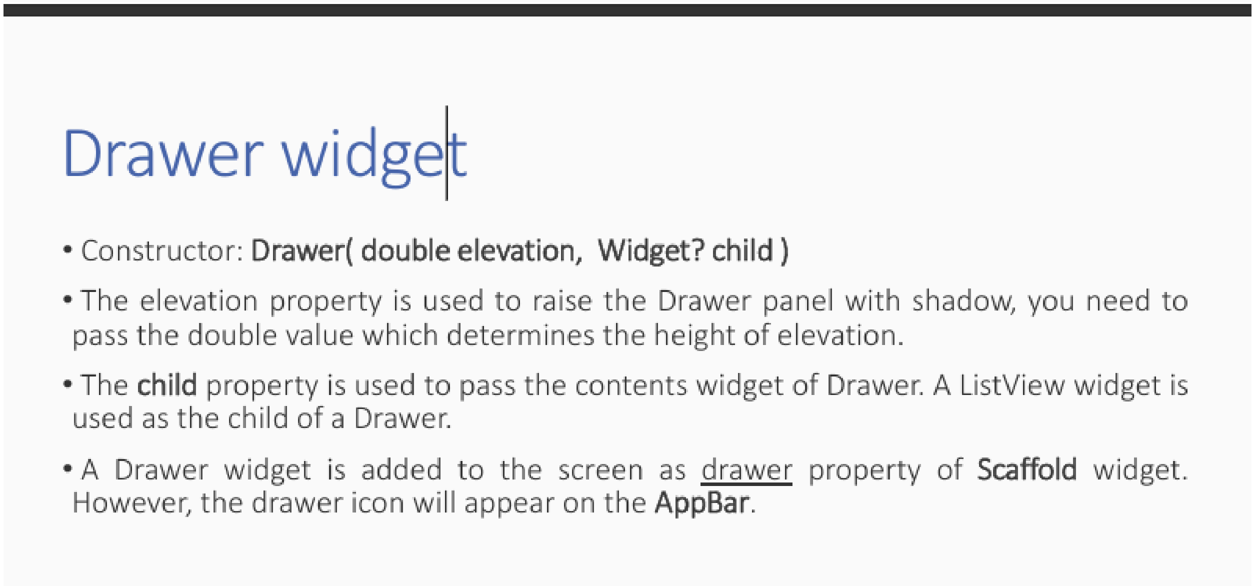
28. Write the Step by step process to create and use a Drawer. 84



Drawer widget

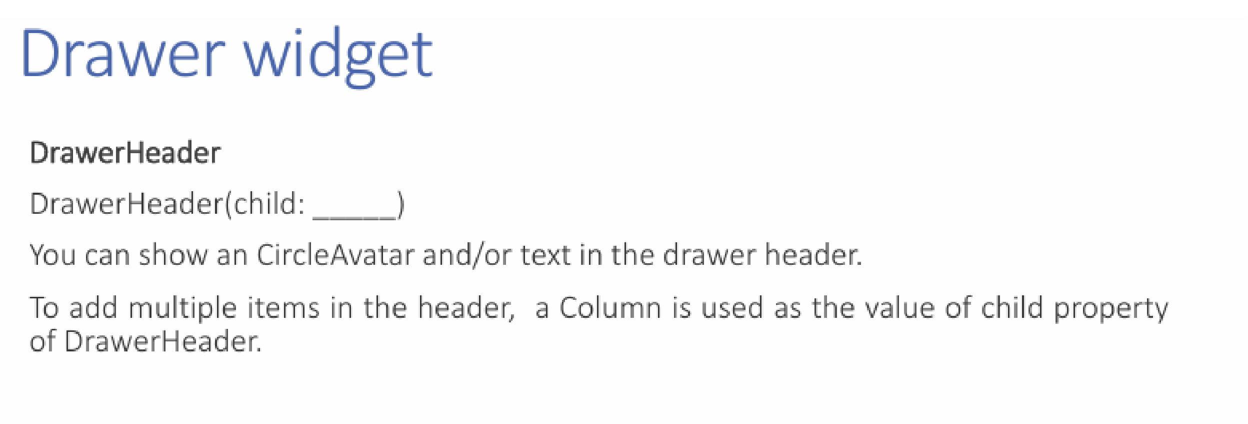
Step-by-step process to create and use a Drawer

1. Create necessary widgets such as CircleAvatar, Text etc. the for DrawerHeader.
2. Add widgets created in Step-1 to a Column widget.
3. Create a DrawerHeader and set its child property to Column widget created in Step-2.
4. Create ListTile widgets for Drawer contents.
5. Add DrawerHeader and allthe ListTile widgets to a ListView.
6. Create a Drawer and set its child property to ListView created in Step-5.
7. Set Drawer widget as drawer property of the Scaffold. To display the drawer on the RHS of the screen, set enddrawer property instead of drawer property.



Drawer widget

- Constructor: **Drawer(double elevation, Widget? child)**
- The elevation property is used to raise the Drawer panel with shadow, you need to pass the double value which determines the height of elevation.
- The **child** property is used to pass the contents widget of Drawer. A ListView widget is used as the child of a Drawer.
- A Drawer widget is added to the screen as drawer property of **Scaffold** widget. However, the drawer icon will appear on the **AppBar**.



Drawer widget

DrawerHeader

DrawerHeader(child: _____)

You can show an CircleAvatar and/or text in the drawer header.

To add multiple items in the header, a Column is used as the value of child property of DrawerHeader.

29. Discuss how images can be used in Flutter applications. 92

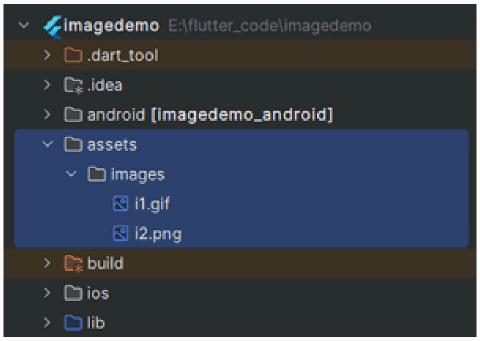
Using images in Flutter

- Displaying an image is a common requirement in almost all the Flutter apps.
- Flutter supports many graphic file formats including JPG, PNG, WEBP, BMP and GIF.
- An image can be present as an image file on the device or on the Internet.
- To display an image from an image file available on the device, follow the steps given below:

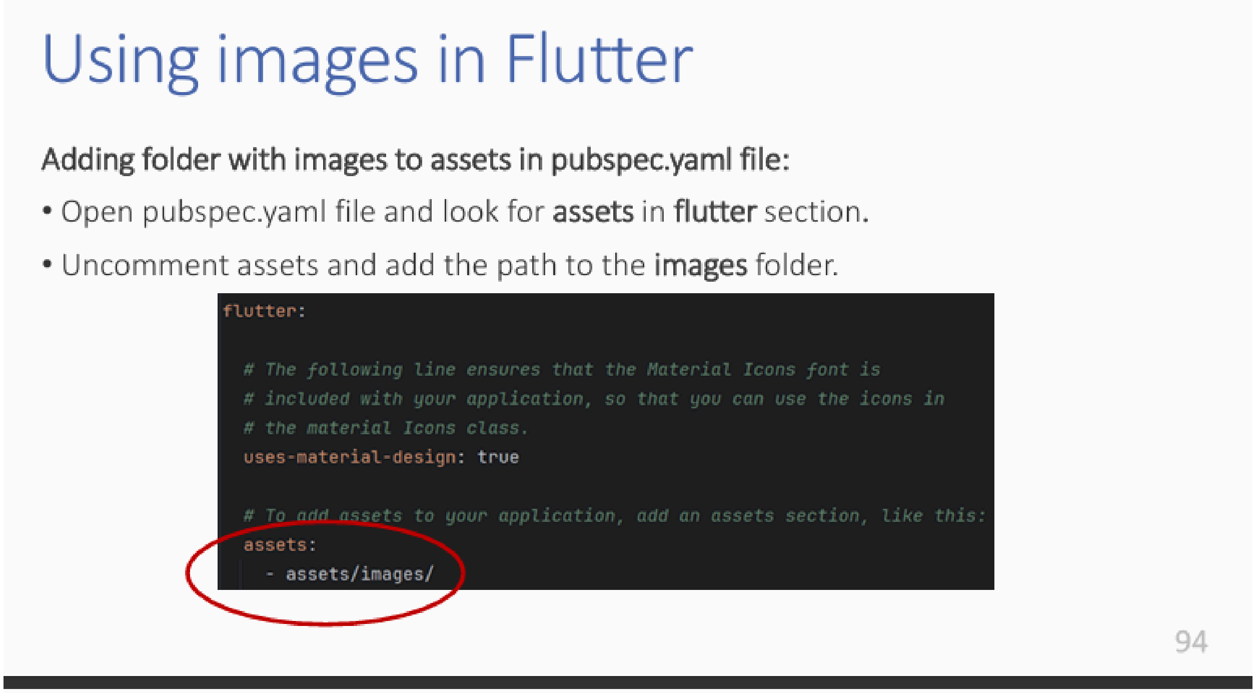
1. Create assets folder in the project folder and then images folder in assets folder.
2. Add image files to the images folder.
3. Add path to images folder in pubspec.yaml file.
4. Use Image widget.

Using images in Flutter

- An image file can be used to display an image in a Flutter app.
- The file must be stored in the assets folder.
- By default, this folder does not exist. Therefore, you must create a folder named **assets** in the project folder.
- Then, create a folder named images in the assets folder.
- Next, copy all the image files in the **images** folder.



```
imagedemo  E:\flutter_code\imagedemo
  .dart_tool
  .idea
  android [imagedemo_android]
  assets
    images
      i1.gif
      i2.png
  build
  ios
  lib
```

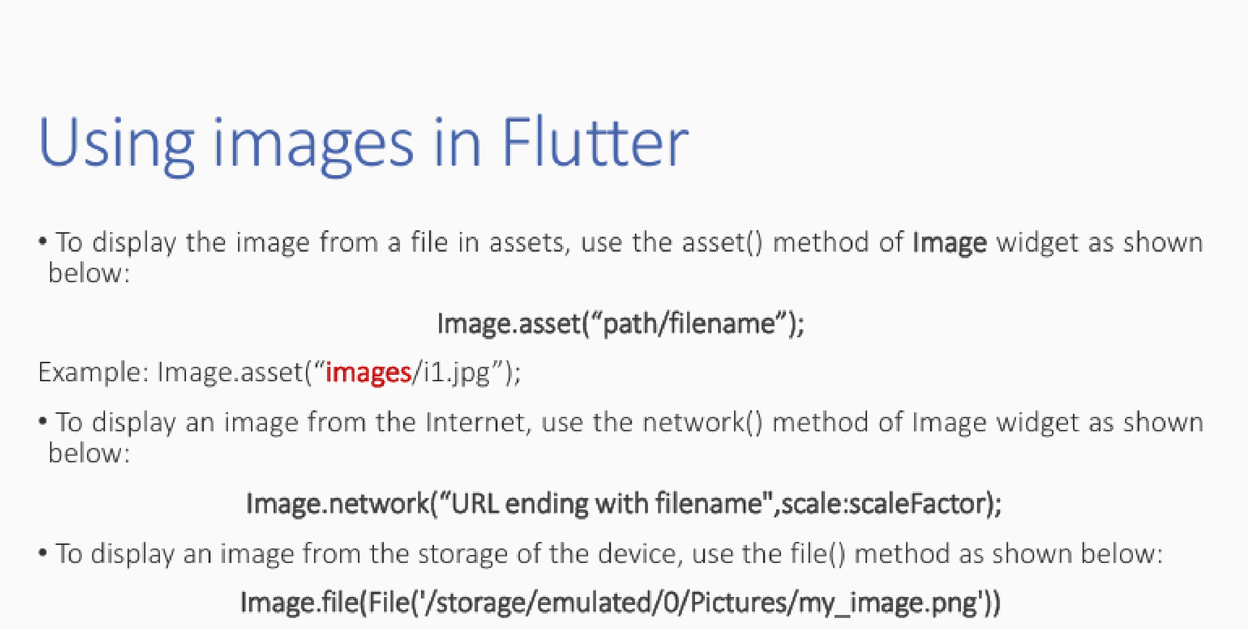


Using images in Flutter

Adding folder with images to assets in pubspec.yaml file:

- Open pubspec.yaml file and look for **assets** in **flutter** section.
- Uncomment assets and add the path to the **images** folder.

```
flutter:

  # The following line ensures that the Material Icons font is
  # included with your application, so that you can use the icons in
  # the material Icons class.
  uses-material-design: true

  # To add assets to your application, add an assets section, like this:
  assets:
    - assets/images/
```



Using images in Flutter

- To display the image from a file in assets, use the asset() method of **Image** widget as shown below:

Image.asset(“path/filename”);

Example: Image.asset(“**images**/i1.jpg”);

- To display an image from the Internet, use the network() method of Image widget as shown below:

Image.network(“URL ending with filename",scale:scaleFactor);

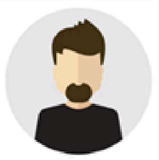
- To display an image from the storage of the device, use the file() method as shown below:

Image.file(File('/storage/emulated/0/Pictures/my_image.png'))

30. Explain the CircleAvatar widget. 96-98

The CircleAvatar widget

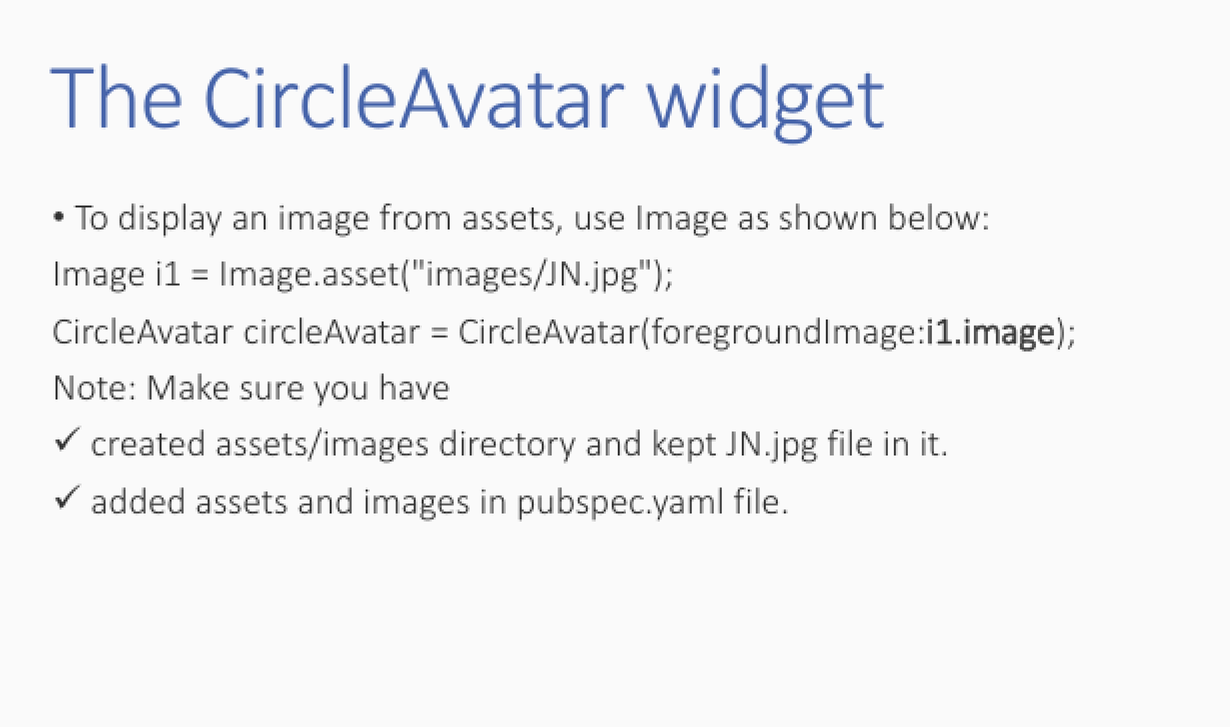
- The CircleAvatar widget is used to display an image, icon, or text inside a circular frame.
- The image can be from assets or Internet.
- It is commonly used for user profile pictures, initials, or decorative avatars.



The CircleAvatar widget

Important properties:

Attribute	Description
radius	Controls the size of the avatar. Default value is 40.
foregroundImage	Image to display inside the circle. ***AssetImage, NetworkImage*** etc is used here.
child	It is used to display text or icon inside the circle. Can be a Text, Icon, or another widget inside the avatar.
backgroundColor	Sets the background color if no image is used.
foregroundColor	Sets the color of the text/icon inside the avatar.



The CircleAvatar widget

- To display an image from assets, use Image as shown below:

Image i1 = Image.asset("images/JN.jpg");

CircleAvatar circleAvatar = CircleAvatar(foregroundImage:**i1.image**);

Note: Make sure you have

✓ created assets/images directory and kept JN.jpg file in it.

✓ added assets and images in pubspec.yaml file.

Sure! Here's the **detailed solution for Question 31** from your question bank, with proper formatting and the **reference slide numbers** included:

✅ 31. List four types Alert dialogs available in Flutter. Also, explain important properties of an AlertDialog.

📘 Reference: Slide 126–127, Revised PPT CAUC513 AMP

🧩 Types of AlertDialogs in Flutter

Flutter provides several types of alert dialogs to interact with users. Here are four commonly used types:

1. Basic AlertDialog

- A simple dialog with a **title**, **message**, and **OK** button.

- Used to give short notifications or acknowledgments.

2. Confirmation AlertDialog

- Used to ask the user for confirmation, typically with **Yes** and **No** buttons.
- Suitable for delete confirmations or exiting the app.

3. Select AlertDialog

- Allows the user to select from multiple options.
- Used when multiple choices are available like selecting a color or mode.

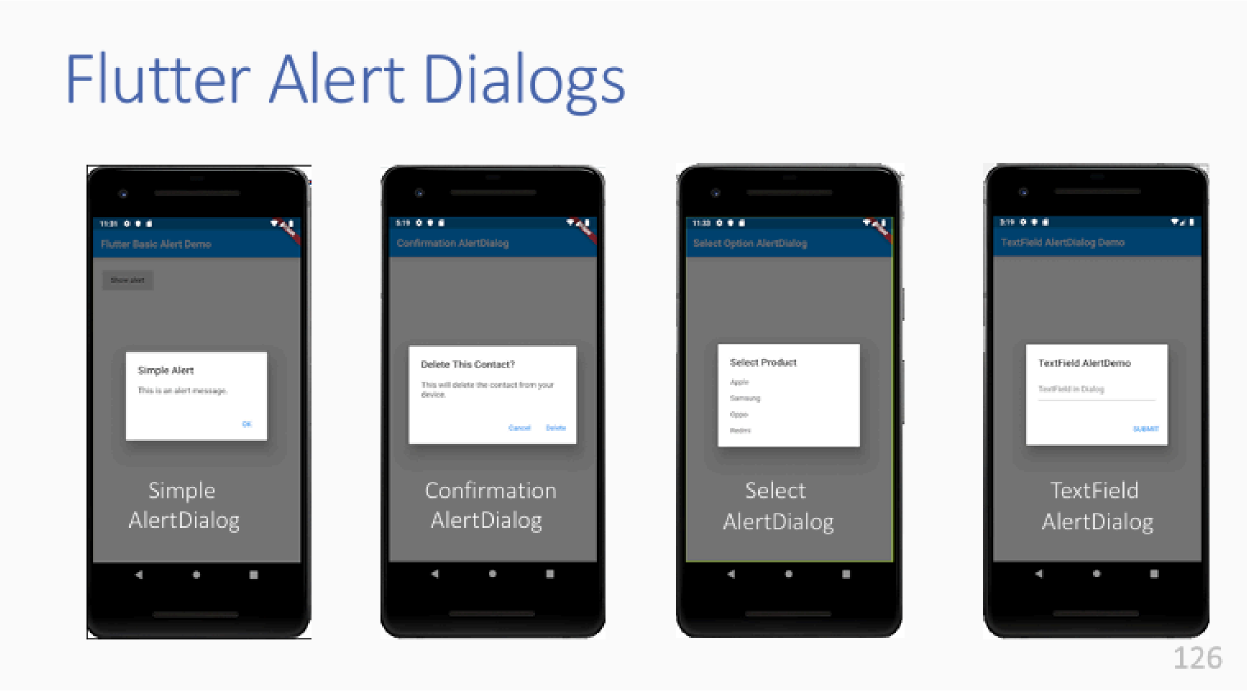
4. TextField AlertDialog

- Contains a `TextField` inside the dialog's `content`.
- Used to take user input such as a name, comment, or email.

🛠️ Important Properties of AlertDialog Widget

Property	Description
`title`	A widget (commonly `Text`) displayed at the top of the dialog.
`content`	The main content of the dialog. Can be a `Text`, `TextField`, or any widget.
`actions`	List of widgets (usually buttons like `TextButton` or `ElevatedButton`).
`backgroundColor`	Sets background color of the dialog.

`shape`	Sets the shape of the dialog, like rounded corners using `RoundedRectangleBorder`.



Flutter Alert Dialogs

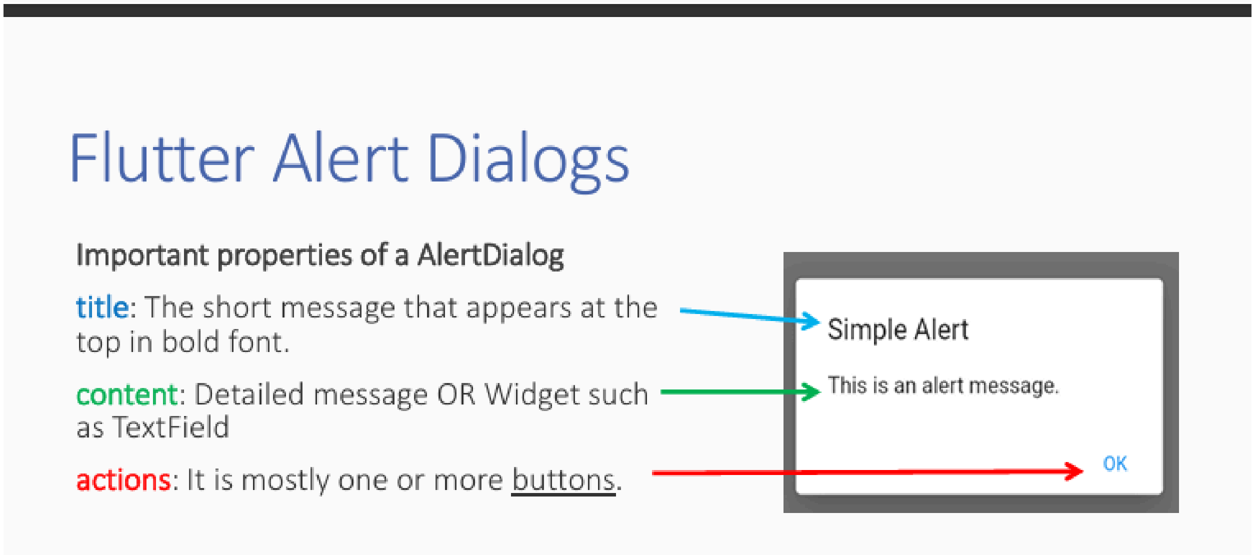
Simple AlertDialog

Confirmation AlertDialog

Select AlertDialog

TextField AlertDialog

126



Flutter Alert Dialogs

Important properties of a AlertDialog

title: The short message that appears at the top in bold font.

content: Detailed message OR Widget such as TextField

actions: It is mostly one or more buttons.

💻 Example Code: Basic AlertDialog

```
void showAlertDialog(BuildContext context) {
  // Step 1: Create OK button
  Widget okButton = TextButton(
    child: Text("OK"),
    onPressed: () {
      Navigator.of(context).pop(); // Closes the dialog
    },
  );

  // Step 2: Create AlertDialog object
  AlertDialog alert = AlertDialog(
    title: Text("Simple Alert"),
    content: Text("This is an alert message."),
    actions: [okButton],
  );

  // Step 3: Call showDialog()
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return alert;
    },
  );
}
```

📝 Summary

- Flutter offers multiple types of `AlertDialog` to serve different UI needs.
- Properties like `title`, `content`, and `actions` allow customization.
- Dialogs are displayed using `showDialog()` function.

Here is the **detailed solution for Question 32** from the question bank:

✅ 32. Write step-by-step procedure to create and use a Form widget.

📘 Reference: Slide 147, Revised PPT CAUC513 AMP

🧩 What is a Form Widget in Flutter?

- The `Form` widget in Flutter is used to group and validate multiple form fields (like `TextField`, `Dropdown`, `Checkbox`, etc.).
- It simplifies managing form state, validations, and submissions using a `GlobalKey<FormState>`.

🪜 Step-by-Step Procedure to Create and Use a Form Widget

🔹 Step 1: Create a GlobalKey for Form

Used to uniquely identify the `Form` and access form state for validation and submission.

```
final _formKey = GlobalKey<FormState>();
```

🔹 Step 2: Create a Form Widget

Wrap your input widgets (e.g., `TextFormField`) inside a `Form` widget and assign the key.

```
Form(
  key: _formKey,
  child: Column(
    children: <Widget>[
      // Add input fields here
    ],
  ),
)
```

🔹 Step 3: Use `TextFormField` Widgets

Use `TextFormField` instead of `TextField` to enable built-in validation.

```
TextFormField(
  decoration: InputDecoration(labelText: 'Enter your name'),
  validator: (value) {
    if (value == null || value.isEmpty) {
      return 'Please enter your name';
    }
    return null;
  },
),
```

🔹 Step 4: Add Submit Button

Use a button to trigger form validation and submission.

```
ElevatedButton(
  onPressed: () {
    if (_formKey.currentState!.validate()) {
      // Process data if valid
      print("Form is valid");
    }
  },
  child: Text('Submit'),
),
```

💡 **Full Example:**

```
class MyForm extends StatelessWidget {
  final _formKey = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    return Form(
      key: _formKey,
      child: Column(
        children: <Widget>[
          TextFormField(
            decoration: InputDecoration(labelText: 'Email'),
            validator: (value) {
              if (value == null || value.isEmpty) {
                return 'Please enter email';
              }
              return null;
            },
          ),
          ElevatedButton(
            onPressed: () {
              if (_formKey.currentState!.validate()) {
                print("Valid Form Submitted");
              }
            },
            child: Text('Submit'),
          ),
        ],
      ),
    );
  }
}
```

📝 Key Points:

- Use `Form` for grouping fields and handling validation.
- Each `TextFormField` can have its own `validator` function.
- Use `_formKey.currentState!.validate()` to trigger validation.
- `FormState` allows form reset and validation.

Here is the **detailed solution for Question 33** from your question bank:

✅ 33. Explain how you will create three different Badge widgets to fulfil the following requirements:

📘 Reference: Slide 153, Revised PPT CAUC513 AMP

🧩 Badge Widget in Flutter

The `Badge` widget is used to visually notify the user of events like unread messages, missed calls, or alerts, often with or without a number.

Flutter does not have a built-in `Badge` widget in the standard library, but badges can be created using external packages like badges or custom widgets using `Stack`, `Positioned`, etc.

✨ Requirements and Solutions

✅ 1) Notify the user (without specifying the count) in case one or more calls are missed

- You can use a `Badge` with no count, just a red dot or icon.

```
Badge(
  child: Icon(Icons.call),
  badgeStyle: BadgeStyle(
    badgeColor: Colors.red,
    shape: BadgeShape.circle,
  ),
)
```

✅ 2) Notify the user about the number of unread messages without using the `count()` function

- You can manually specify the number.

```
Badge(
  label: Text("5"),  // Replace 5 with actual unread count
  child: Icon(Icons.message),
)
```

✅ 3) Notify the user about the number of unread messages using the `count()` function

- If the unread messages are stored in a list, use `.length` to dynamically show the count.

```
List<String> unreadMessages = ["Hi", "Hello", "Check this"];
Badge(
  label: Text("${unreadMessages.length}"),
  child: Icon(Icons.message),
)
```

🧪 Example Using badges Package (Recommended)

Add to `pubspec.yaml`:

```
dependencies:
  badges: ^3.1.1
```

Then use:

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


Badge(
  label: Text('3'),
  child: Icon(Icons.notifications),
);
```

📝 Key Points:

- Badges improve user engagement by showing actionable notifications.
- You can use text, dots, or icons as badge indicators.
- For dynamic counts, use variables like `.length`.

Here is the **detailed solution for Question 34** from your question bank:

✅ 34. Explain important arguments of `showTimePicker()` function.

📘 Reference: Slide 161, Revised PPT CAUC513 AMP

🧩 What is `showTimePicker()` in Flutter?

- `showTimePicker()` is a built-in Flutter function used to display a time picker dialog.
- It allows users to select a time (hours and minutes) using either a dial or text input.
- It returns a `Future<TimeOfDay?>` which completes when the user selects or cancels the picker.

🧾 Syntax:

```
Future<TimeOfDay?> showTimePicker({
  required BuildContext context,
  required TimeOfDay initialTime,
  bool useRootNavigator = true,
  Widget? helpText,
  EntryMode initialEntryMode = TimePickerEntryMode.dial,
  bool use24HourFormat = false,
  TransitionBuilder? builder,
  String? cancelText,
  String? confirmText,
});
```

🛠️ Important Arguments Explained

Argument	Description
`context`	**(Required)**: The build context used to locate the widget in the tree.
`initialTime`	**(Required)**: The time that is initially selected when the picker opens.

`initialEntryMode`	Defines how the time picker opens – **dial** (default) or **input** mode.
`use24HourFormat`	If `true`, shows time in 24-hour format (e.g., 18:30). Default is `false`.
`cancelText`	Label for the Cancel button. Default is `"Cancel"`.
`confirmText`	Label for the OK button. Default is `"OK"`.
`helpText`	Optional widget shown at the top of the picker (e.g., `"Select Appointment"`).
`builder`	Allows customization of the dialog appearance.

💻 Example Code:

```
Future<void> _selectTime(BuildContext context) async {
  final TimeOfDay? picked = await showTimePicker(
    context: context,
    initialTime: TimeOfDay.now(),
    helpText: "Choose time for your appointment", // Top label
    confirmText: "Set Time",                      // Confirm button
    cancelText: "Cancel",                         // Cancel button
    use24HourFormat: true,                    // 24-hour format
    initialEntryMode: TimePickerEntryMode.dial,   // Entry mode
  );

  if (picked != null) {
    print("Selected Time: ${picked.format(context)}");
  }
}
```

📝 Key Points:

- Always provide `context` and `initialTime`.
- Customize appearance and behavior using optional parameters.
- Returns `null` if user cancels the dialog.

Here is the **detailed solution for Question 35** from your question bank:

---

35. Write code of a function with the following requirement:

- The function takes three arguments of double type. Their names are length, breadth and rate.

- Rate means rate of painting the shape on both sides.

- If the value for rate is passed, that value must be used. Otherwise take 5 as its value.

- Calculate and return cost of painting using the following formula: cost = 2 X length X breadth X rate

✅ 35. Write code of a function with the following requirement:

📘 Reference: Slide 199, Revised PPT CAUC513 AMP

🧩 Requirements Recap:

Write a function that:

- Takes **three arguments** of `double` type: `length`, `breadth`, and `rate`.
- `rate` is **optional**; if not passed, use default value `5`.
- Calculates cost of painting **both sides** using the formula: cost=2×length×breadth×rate\text{cost} = 2 \times \text{length} \times \text{breadth} \times \text{rate}
- Returns the calculated **cost**.

💻 Dart Function Code Using Named Parameter with Default Value:

```
double calculatePaintingCost({
  required double length,
  required double breadth,
  double rate = 5,  // default value if not provided
}) {
  return 2 * length * breadth * rate;
}
```

✅ Example Usage:

```
void main() {
  double cost1 = calculatePaintingCost(length: 10, breadth: 5);
  print("Cost1: $cost1"); // Uses default rate = 5

  double cost2 = calculatePaintingCost(length: 10, breadth: 5, rate: 7);
  print("Cost2: $cost2"); // Uses rate = 7
}
```

📝 Key Concepts Used:

- **Named parameters** using `{}`.
- **Default value** assignment for optional parameter (`rate = 5`).
- **Required keyword** to enforce non-null inputs (`length`, `breadth`).

Here is the **detailed solution for Question 36** from your question bank:

✅ 36. Explain the following terms in the context of database connectivity: `snapshot`, `stream`, `StreamBuilder`

📘 Reference: Firebase & Firestore concepts (commonly covered under Slide Range: 260+, Revised PPT CAUC513 AMP)

🔹 1) `snapshot` (or `DocumentSnapshot` / `QuerySnapshot`)

- A **snapshot** is a read-only copy of data retrieved from a database (e.g., Firestore).
- It represents the **current state** of the data at a specific point in time.
- There are two types:
    - `DocumentSnapshot`: Represents a single document.
    - `QuerySnapshot`: Represents a collection of documents.

✅ Example:

```
DocumentSnapshot snapshot = await FirebaseFirestore.instance
  .collection('users')
  .doc('userId')
  .get();

String name = snapshot['name'];
```

🔹 2) `stream`

- A **stream** is a sequence of asynchronous events.
- In Firebase, it continuously listens to real-time updates in Firestore.
- Useful for reflecting live updates in your UI.

✅ **Example:**

```
Stream<QuerySnapshot> userStream = FirebaseFirestore.instance
  .collection('users')
  .snapshots();
```

🔹 3) `StreamBuilder`

- A **widget** that builds itself based on the latest snapshot of interaction with a stream.

- It is used to connect a **Stream** (like from Firestore) to the **UI**, and auto-updates the widget when the stream emits a new value.

✅ Example:

```
StreamBuilder(
  stream: FirebaseFirestore.instance.collection('users').snapshots(),
  builder: (context, AsyncSnapshot<QuerySnapshot> snapshot) {
    if (!snapshot.hasData) return CircularProgressIndicator();

    return ListView(
      children: snapshot.data!.docs.map((doc) => ListTile(
        title: Text(doc['name']),
      )).toList(),
    );
  },
)
```

📝 Summary Table:

Term	Meaning
`snapshot`	A static read of the current data from Firestore

`stream`	A continuous flow of real-time data updates from Firestore
`StreamBuilder`	Flutter widget that listens to a stream and rebuilds UI on new data

---

Here is the **detailed solution for Question 37** from your question bank:

---

37. The followings are steps to fetch data from a Firestore database. They are not in the correct order. Arrange them in the correct order.

- Import required packages.

- Initialize a Firebaseapp instance in main().

- Add firebase_core and cloud_firestore dependencies to pubspec.yaml.

- Configure Flutter app for Firebase.

- Create a Stream.

- Create a StreamBuilder and a DocumentSnapshot.

- Access data from DocumentSnapshot.

✅ **37. The followings are steps to fetch data from a Firestore database. They are not in the correct order. Arrange them in the correct order.**

📘 **Reference: Slide 260, Revised PPT CAUC513 AMP**

🧩 Steps Given (Unordered):

- Import required packages.
- Initialize a FirebaseApp instance in `main()`.
- Add `firebase_core` and `cloud_firestore` dependencies to `pubspec.yaml`.
- Configure Flutter app for Firebase.
- Create a Stream.
- Create a `StreamBuilder` and a `DocumentSnapshot`.
- Access data from `DocumentSnapshot`.

🪜 Correct Order to Fetch Data from Firestore:

1. ✅ **Add `firebase_core` and `cloud_firestore` dependencies to `pubspec.yaml`**
    - Required to use Firebase in Flutter.

```
dependencies:
  firebase_core: latest
  cloud_firestore: latest
```

2.

✅ **Import required packages**

```
import 'package:firebase_core/firebase_core.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
```

3.

✅ Initialize a FirebaseApp instance in `main()`

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

4.
5. ✅ **Configure Flutter app for Firebase**
   - Set up `firebase_options.dart` using Firebase CLI.
   - Link your Firebase project to the app (done outside Dart code via `flutterfire configure`).

✅ Create a Stream

```
 Stream<QuerySnapshot> userStream = FirebaseFirestore.instance
  .collection('users')
  .snapshots();
```

6.

✅ Create a `StreamBuilder` and a `DocumentSnapshot`

```
 StreamBuilder(
  stream: userStream,
  builder: (context, AsyncSnapshot<QuerySnapshot> snapshot) {
    if (!snapshot.hasData) return CircularProgressIndicator();
    // access docs here
  },
)
```

7.

✅ Access data from `DocumentSnapshot`

```
 final docs = snapshot.data!.docs;
final name = docs[0]['name'];  // Access document field
```

8.

📝 Summary (Final Sequence):

Step	Action
1	Add dependencies in `pubspec.yaml`
2	Import Firebase packages
3	Initialize Firebase in `main()`

4	Configure app for Firebase (outside Dart)
5	Create a Stream from Firestore
6	Use a `StreamBuilder` to consume the stream
7	Access data from `DocumentSnapshot`

✅ 38. I need to get 15 records from an HTTP service. Replace A, B, C, D and E in the following code to make it correct.

📘 Reference: Slide 303, Revised PPT CAUC513 AMP

🧩 Question Statement Code (with placeholders):

```
Uri url = Uri.parse("https://randomuser.me/api/?A");

var response = B   C.D(url, headers: {"Accept": "application/json"});

var listData = E.decode(response.body);
data = listData['results'];
```

🛠️ Correct Answer (Replacements):

Placeholder	Correct Word	Explanation
A	`results=15`	Tells the API to return 15 random users.
B	`await`	Wait for the response asynchronously.
C	`http`	Refers to the HTTP package.
D	`get`	Performs a GET request.
E	`json`	Used to decode JSON response from the server.

✅ Corrected Code:

```
import 'package:http/http.dart' as http;
import 'dart:convert';

void fetchData() async {
  Uri url = Uri.parse("https://randomuser.me/api/?results=15");

  var response = await http.get(url, headers: {"Accept": "application/json"});

  var listData = json.decode(response.body);
  var data = listData['results'];

  print(data); // Prints list of 15 users
}
```

📝 Summary:

- You must use `await` with asynchronous network calls.

- `http.get()` is the correct method to fetch data.
- `json.decode()` parses the response into a Dart object.
- Always import both `http` and `dart:convert`.

---

39. The followings are steps to get the location of a device. They are not in the correct order. Arrange them in the correct order.

- Check if the location service is enabled or not.

- Check if permission to access location service is available or not.

- Create a Location object.

- Request permission to access location.

- Get value of longitude and latitude.

- Create a Location object. - Call getLocation().

✅ 39. The following are steps to get the location of a device. They are not in the correct order. Arrange them in the correct order.

📘 Reference: Slide 311–313, Revised PPT CAUC513 AMP

🧩 Steps Given (Unordered):

- Check if the location service is enabled or not.
- Check if permission to access location service is available or not.
- Create a Location object.

- Request permission to access location.
- Get value of longitude and latitude.
- Create a Location object.
- Call `getLocation()`.

(Note: "Create a Location object" appears twice—merge as one step.)

🪜 Correct Order to Get Device Location Using `location` Package:

✅ Step 1: Create a `Location` object

```
Location location = Location();
```

✅ Step 2: Check if location service is enabled

```
bool serviceEnabled = await location.serviceEnabled();

if (!serviceEnabled) {

  serviceEnabled = await location.requestService();

  if (!serviceEnabled) {

    return; // Exit if user denies service

  }

}
```

✅ Step 3: Check for permission

```
PermissionStatus permissionGranted = await location.hasPermission();

if (permissionGranted == PermissionStatus.denied) {

  permissionGranted = await location.requestPermission();

  if (permissionGranted != PermissionStatus.granted) {

    return; // Exit if permission not granted

  }

}
```

✅ Step 4: Get location (latitude and longitude)

```
LocationData locationData = await location.getLocation();

print("Latitude: ${locationData.latitude}, Longitude: ${locationData.longitude}");
```

📝 Summary (Final Ordered Steps):

Step	Description
1	Create a `Location` object
2	Check if location service is enabled

3	Request service if disabled
4	Check for location permission
5	Request permission if not granted
6	Call `getLocation()` to fetch coordinates
7	Use `latitude` and `longitude` from `LocationData`

✅ **Required Package:**
Make sure to add this to `pubspec.yaml`:

```
location: ^5.0.3
```

And import it:

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

Here is the **detailed solution for Question 40** from your question bank:

---

40. Rewrite the following code by adding exception handling mechanism.

```
 void main() {

 String str = "a23";

int age = int.parse(str);

 print("str = $str");

 print("age = $age");

 }
```

✅ 40. Rewrite the following code by adding exception handling mechanism

📘 Reference: Slide 224, Revised PPT CAUC513 AMP

🧩 Given Code (without exception handling):

```
void main() {

  String str = "a23";

  int age = int.parse(str);

  print("str = $str");

  print("age = $age");

}
```

🔺 This code will throw a `FormatException` because `"a23"` is not a valid integer.

🛠️ Updated Code with Exception Handling (`try-catch`):

```
void main() {
  String str = "a23";
  int age;

  try {
    age = int.parse(str); // This line may throw FormatException
    print("str = $str");
    print("age = $age");
  } catch (e) {
    print("Error occurred: $e");
  }
}
```

✅ Advanced Version with Specific Exception Handling:

```
void main() {
  String str = "a23";
  int age;

  try {
    age = int.parse(str);
    print("str = $str");

    print("age = $age");

  } on FormatException catch (e) {

    print("Format error: $e");

  } catch (e) {

    print("Unexpected error: $e");

  }

}
```

📝 **Key Concepts Used:**

- `try`: Wraps code that may throw an error.
- `catch (e)`: Catches any type of exception.
- `on FormatException`: Catches specific exception type (e.g., parsing errors).
- Prevents app crashes and allows graceful failure handling.

Here is the **detailed solution for Question 41** from your question bank:

✅ **41. Explain with example: How code can be monitored for probable exceptions and code to be executed in case exception occurs using `try`, `catch` and `finally`.**

📘 Reference: Slides 222–223, 229 — Revised PPT CAUC513 AMP

🧩 Exception Handling in Dart

Exception handling is used to manage runtime errors gracefully without crashing the app.
Flutter (Dart) provides three main keywords:

Keyword	Purpose
`try`	To wrap the code that might throw an exception.
`catch`	To catch and handle the exception.
`finally`	Executes a block of code regardless of whether an exception occurred or not.

💻 Example: Handling Division and Input Errors

```
void main() {
  int a = 10;
  int b = 0;
  int result;

  try {
    result = a ~/ b;  // Integer division may throw IntegerDivisionByZeroException

    print("Result: $result");

  } catch (e) {

    print("Exception caught: $e");

  } finally {

    print("This block always runs.");

  }

}
```

📝 Output:

```
Exception caught: IntegerDivisionByZeroException

This block always runs.
```

✅ Explanation of Each Part:

Part	Description
`try`	The risky code (`a ~/ b`) is placed here.
`catch`	Catches and prints the exception.

`finally`	Executes cleanup or important code, even if an exception was caught.

🎯 Another Example: Parsing User Input

```
void main() {
  String str = "abc";
  try {
    int number = int.parse(str);
    print("Number = $number");
  } catch (e) {
    print("Error: $e");
  } finally {
    print("Parsing attempt completed.");
  }
}
```

📝 Best Practices:

- Always use `try-catch` around risky code (like file I/O, parsing, HTTP, DB).
- Use `finally` for cleanup (e.g., closing a file/connection).

- Catch specific exceptions (`on FormatException catch(e)`) when possible.

Here is the **detailed solution for Question 42** from your question bank:

✅ 42. Explain real-life application of `await` with an example.

📘 Reference: Slide 242, Revised PPT CAUC513 AMP

🧩 What is `await` in Dart/Flutter?

- The `await` keyword is used to pause the execution of an async function **until a `Future` completes**.
- It allows writing **asynchronous code** in a **sequential, readable manner**.
- Commonly used when fetching data from a database, API, or reading files.

🎯 Real-Life Application Example: Fetching Weather Data from an API

In a weather app, you might fetch live temperature from an API using `await`.

💻 Example:

```
import 'package:http/http.dart' as http;

import 'dart:convert';
```

```
void main() async {
  print("Fetching weather data...");

  // Await is used to pause until the response is received
  var response = await http.get(Uri.parse("https://api.weatherapi.com/v1/current.json?key=YOUR_API_KEY&q=London"));

  if (response.statusCode == 200) {
    var data = jsonDecode(response.body);
    print("Current temperature in London: ${data['current']['temp_c']}°C");
  } else {
    print("Failed to fetch data");
  }

  print("Done.");
}
```

📝 Explanation:

Step	Description

`await http.get(...)`	Sends an HTTP request and **waits for response** without blocking UI.
`jsonDecode`	Parses the response body once it's received.
`async`	Declares that `main()` is asynchronous and can use `await`.

✅ Why is `await` important in real apps?

- Ensures that app **does not freeze** while waiting for data.
- Prevents **callback hell** and makes code **easy to read and maintain**.
- Used with: Firebase, REST APIs, file system access, animations, etc.

🧪 Common Use Cases in Flutter:

- `await Firebase.initializeApp()`
- `await SharedPreferences.getInstance()`
- `await Navigator.push(...)`
- `await showDialog(...)`

Here is the **detailed solution for Question 43** from your question bank:

✅ 43. Each of the following statements is associated with either Stateless or Stateful widget. Put them on the correct side to create the difference between Stateless and Stateful widget.

📘 Reference: Slide 79, Revised PPT CAUC513 AMP

🧩 Statements to Classify:

1. Examples: `CheckboxListTile`, `RadioListTile`, `TextField`
2. Subclass of `StatefulWidget` class
3. Overrides `createState()` method
4. `setState()` is used.
5. Examples: `Text`, `Row`, `Column`
6. Subclass of `StatelessWidget` class
7. Subclass of `State` is not used.
8. Overrides the `build()` method
9. `setState()` is not used.
10. Subclass of `State` is used.

🧾 Correct Classification:

📦 **Stateful Widget:**

No.	Statement
1	Examples: `CheckboxListTile`, `RadioListTile`, `TextField`
2	Subclass of `StatefulWidget` class
3	Overrides `createState()` method
4	`setState()` is used
10	Subclass of `State` is used

📦 **Stateless Widget:**

No.	Statement
5	Examples: `Text`, `Row`, `Column`
6	Subclass of `StatelessWidget` class

7	Subclass of `State` is not used
8	Overrides the `build()` method
9	`setState()` is not used

📝 Key Differences Recap:

Feature	Stateless Widget	Stateful Widget
UI Change	Not possible after build	Possible using `setState()`
Lifecycle Classes	One class (extends `StatelessWidget`)	Two classes (`StatefulWidget` and `State`)
Examples	`Text`, `Icon`, `Row`, `Column`	`Checkbox`, `Slider`, `TextField`
Performance	Better (no state management)	Slightly slower (manages state)

Here is the **detailed solution for Question 44** from your question bank:

✅ 44. Explain custom exception with an example.

📘 Reference: Slides 233–235, Revised PPT CAUC513 AMP

🧩 What is a Custom Exception in Dart?

- A **custom exception** is a user-defined class used to throw meaningful and specific error messages during exceptional situations in a program.
- It helps improve **error handling** and **code readability**, especially in large apps or libraries.

🛠️ How to Create a Custom Exception

You can create a custom exception by defining a class that implements the `Exception` interface.

💻 Example: Creating and Using a Custom Exception

```
// Step 1: Define a custom exception

class NegativeAmountException implements Exception {

  String errorMessage() {

    return "Amount cannot be negative!";

  }

}
```

```
// Step 2: Use the custom exception in a function
void deposit(double amount) {
  if (amount < 0) {
    throw NegativeAmountException();
  } else {
    print("Deposited: ₹$amount");
  }
}

// Step 3: Handle the exception using try-catch
void main() {
  try {
    deposit(-500);
  } catch (e) {
    print("Exception caught: ${e is NegativeAmountException ? e.errorMessage() : e}");
  }
}
```

📝 **Output:**

```
Exception caught: Amount cannot be negative!
```

✅ Key Concepts:

Concept	Description
`implements Exception`	Declares that your class is a custom exception
`throw`	Used to manually raise the exception
`try-catch`	Catches and handles the exception
`errorMessage()`	Returns a custom message from the exception

🔐 Why Use Custom Exceptions?

- Helps clearly identify specific errors (e.g., `LoginFailedException`, `InvalidInputException`).
- Makes debugging and logging more meaningful.
- Supports scalable error-handling in enterprise apps.

Here is the **detailed solution for Question 45** from your question bank:

✅ 45. Discuss how images can be used in a Flutter app.

📘 Reference: Slides 117–120, Revised PPT CAUC513 AMP

🧩 Overview:

Flutter allows displaying images in various ways, such as:

- From **assets** (local project folder)
- From **network** (online URLs)
- From **device file system**

All these use the `Image` widget with different constructors like `Image.asset`, `Image.network`, or `Image.file`.

🪜 A. Steps to Use Images from Assets

✅ 1. Create an Assets Folder

Inside your Flutter project directory:

```
/assets/images/
```

- Add your image files (e.g., `logo.png`) inside this folder.

✅ 2. Update `pubspec.yaml`

Uncomment and configure the `assets` section:

```
flutter:

  assets:

    - assets/images/logo.png

    # or to include all images in the folder

    - assets/images/
```

✅ 3. Use `Image.asset` Widget

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

🧩 B. Display Image from the Internet

Use `Image.network` for online images:

```
Image.network('https://example.com/image.png')
```

🧩 C. Display Image from File System

For accessing files from the device:

```
import 'dart:io';


Image.file(File('/storage/emulated/0/Pictures/sample.jpg'))
```

> Note: This requires permissions and may need the `image_picker` or `file_picker` plugin.

💡 Image Widget Properties

Property	Description
`width` / `height`	Sets the size of the image
`fit`	Adjusts how the image should be inscribed (e.g., `BoxFit.cover`, `BoxFit.fill`)
`alignment`	Aligns the image within its container
`color`	Applies a color filter to the image

✅ Example – Using Asset Image in a Flutter App:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text("Image Demo")),
    body: Center(
      child: Image.asset(
        'assets/images/logo.png',
        width: 200,
        height: 200,
        fit: BoxFit.cover,
      ),
    ),
  );
}
```

📌 Summary

Source	Widget Used
Asset	`Image.asset()`
Network	`Image.network()`
File System	`Image.file()`

Always ensure the `pubspec.yaml` is configured correctly when using asset images.