

Chapter 2 : Flutter Basics

Objectives

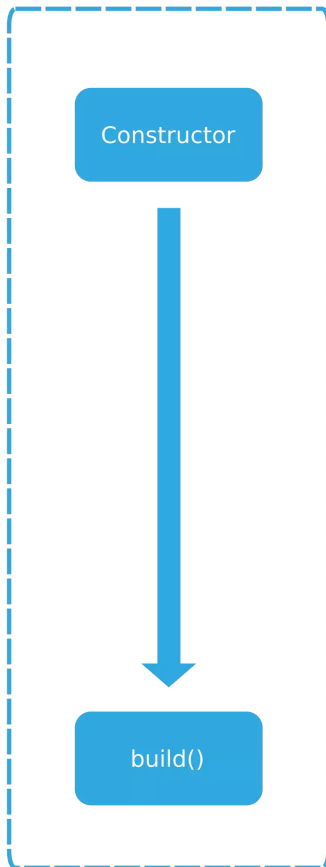
- Widgets
- Gestures
- Concept of State
- Layers
- Introduction to Dart Programming
- Variables and Data types
- Decision Making and Loops
- Functions
- Object Oriented Programming.
- Introduction to Widgets
- Widget Build Visualization

Materials

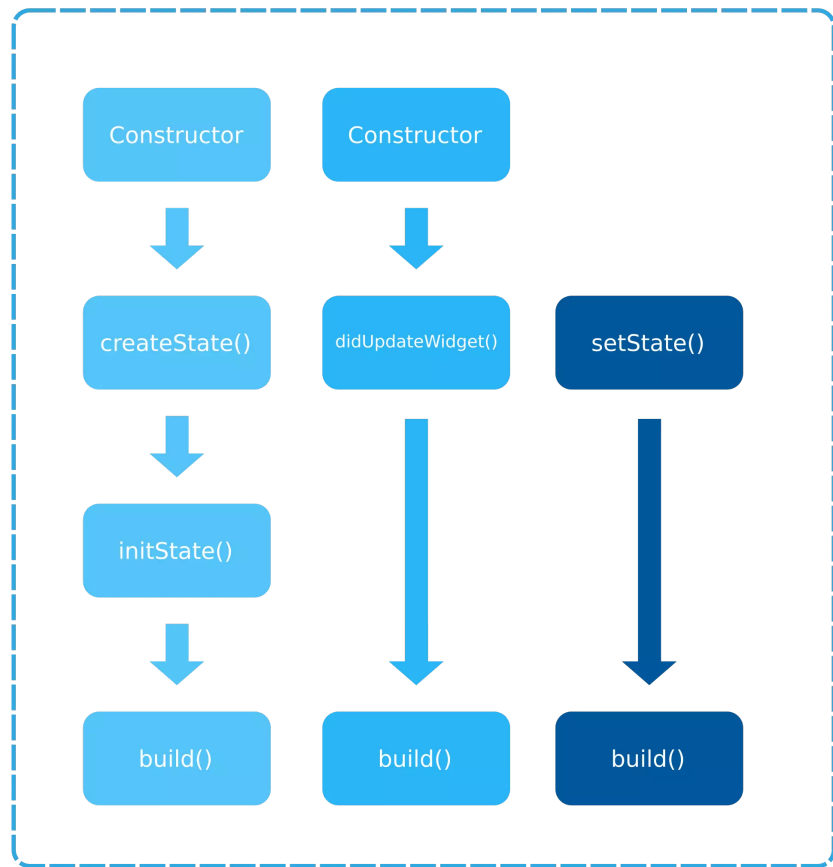
Widgets

In Flutter, widgets are the building blocks of a Flutter app. Everything you see on the screen of a Flutter app is a widget. They are like small pieces of a puzzle that you put together to make a complete picture.

Stateless



Stateful



Lifecycle of a StatefulWidget

In Flutter, a **StatefulWidget** is a widget that can change its state during its lifetime. This means it can rebuild itself multiple times, reflecting changes in its internal state or external data. Understanding the **lifecycle** of a StatefulWidget is crucial for managing resources, performing actions at the right time, and ensuring optimal performance.

The lifecycle of a StatefulWidget involves the following stages:

1. `createState()`
2. `initState()`
3. `didChangeDependencies()`
4. `build()`
5. `setState()`
6. `dispose()`

1. `createState()`

- **Purpose:** This method is the first step in the lifecycle. It is called when the `StatefulWidget` is created. It returns an instance of the `State` class associated with the `StatefulWidget`.
- **Key Point:** This method is called only once.

Example:

```
@override
_ExampleState createState() => _ExampleState();
```

2. `initState()`

- **Purpose:** This method is called once when the `State` object is inserted into the widget tree. It is used for initializing any data or state the widget needs. For example, initializing variables, setting up streams, or starting animations.
- **Key Point:** Always call `super.initState()` to ensure the parent class is also initialized properly.

Example:

```
@override
void initState() {
  super.initState();
  // Initialization code here
}
```

3. `didChangeDependencies()`

- **Purpose:** This method is called immediately after `initState()` and whenever the widget's dependencies change. Dependencies can change if the widget depends on an `InheritedWidget` that itself changes.
- **Key Point:** Useful for actions that depend on the context or inherited widgets.

Example:

```
@override
void didChangeDependencies() {
  super.didChangeDependencies();
  // Respond to dependency changes
}
```

4. `build()`

- **Purpose:** This method is called whenever the widget is to be rendered. It is responsible for constructing the widget tree, which represents the UI. The `build()` method can be called multiple times during the widget's lifecycle, especially after calling `setState()`.
- **Key Point:** The UI is built based on the current state, so changes in state will trigger a rebuild.

Example:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Stateful Widget Lifecycle'),
    ),
    body: Center(
      child: Text('Hello, World!'),
    ),
  );
}
```

5. `setState()`

- **Purpose:** This method is used to notify the framework that the internal state of this object has changed. When called, it triggers a rebuild of the widget, ensuring that the UI reflects the updated state.
- **Key Point:** Use this method responsibly to avoid unnecessary rebuilds.

Example:

```
void _incrementCounter() {
  setState(() {
    _counter++;
  });
}
```

6. `dispose()`

- **Purpose:** This method is called when the `State` object is permanently removed from the widget tree. It is used for final cleanup, such as canceling timers, closing streams, or disposing of controllers.
- **Key Point:** After `dispose()`, the widget is no longer active, and its `State` object is considered dead.

Example:

```
@override
void dispose() {
  // Clean up resources
  super.dispose();
}
```

Summary of Stateful Widget Lifecycle:

1. `createState()` : Called once when the widget is created.
2. `initState()` : Initializes the widget's state.
3. `didChangeDependencies()` : Called when dependencies change.
4. `build()` : Constructs the UI.
5. `setState()` : Triggers a rebuild of the widget.
6. `dispose()` : Final cleanup when the widget is permanently removed.

Understanding the Stateful Widget lifecycle allows you to manage resources efficiently, perform initialization and cleanup actions at the right time, and control how your widget responds to state changes.

Types of Design System

Material Design

Material Design is a design language developed by Google. It is used primarily for Android apps but has become popular across various platforms due to its clean, modern aesthetics and usability principles. Material Design provides a consistent look and feel with a set of guidelines and components.

Key Principles of Material Design:

1. **Material Metaphor:** Material Design is based on the metaphor of physical materials like paper and ink. It uses shadows, depth, and responsive animations to mimic how real-world materials behave.
2. **Bold, Graphic, Intentional:** Emphasis on bold colors, large typography, and meaningful use of space. It aims for clarity and impact.
3. **Motion Provides Meaning:** Motion in Material Design is used to provide context and meaning. For example, transitions and animations help users understand the relationship between different elements.

Button

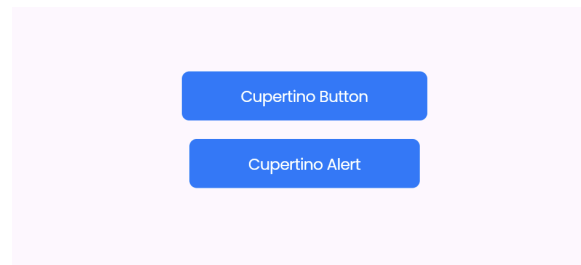
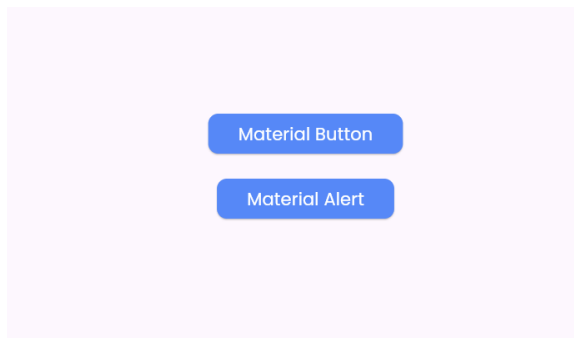
Cupertino Design

Cupertino Design is a design language developed by Apple for iOS. It aims to create a user experience that aligns with iOS conventions and aesthetics. Cupertino widgets provide a native iOS look and feel.

Key Principles of Cupertino Design:

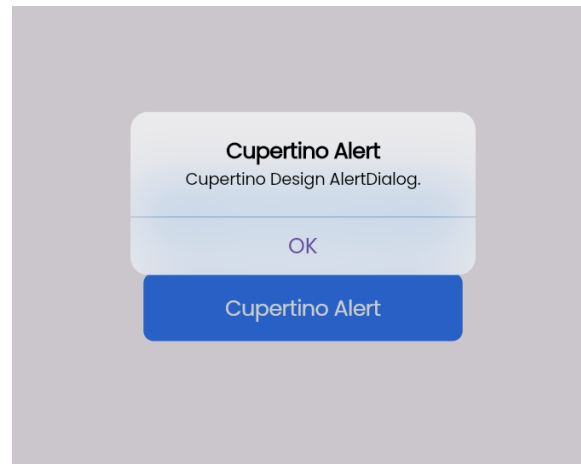
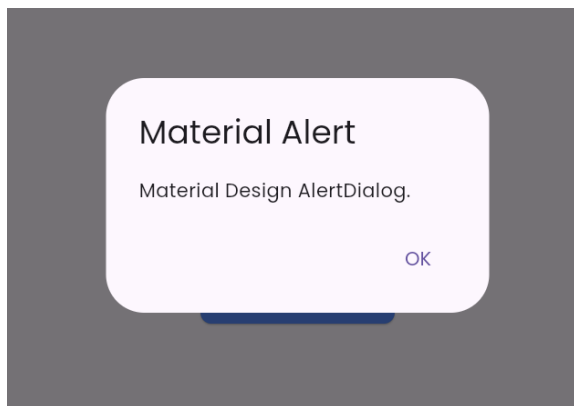
1. **Consistency with iOS:** Cupertino widgets adhere to the design guidelines set by Apple for iOS apps, ensuring that apps have a consistent look and feel with native iOS applications.
2. **Flat and Minimalistic:** Cupertino design tends to use flat, minimalistic elements without the elevated effects seen in Material Design.
3. **Native Feel:** Cupertino widgets are styled to feel native on iOS, using iOS-specific patterns, colors, and interactions.

Button



Alert Box

Alert Box



https://vanshshah1411.github.io/MaterialVSCupertino_flutter/



Material Design Vs Cupertino Design

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

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

class MyApp extends StatelessWidget {
  @override
```

```

Widget build(BuildContext context) {
  return MaterialApp(
    debugShowCheckedModeBanner: false,
    home: DefaultTabController(
      length: 2,
      child: Scaffold(
        appBar: AppBar(
          title: Text('Material vs Cupertino'),
          bottom: TabBar(
            tabs: [
              Tab(text: 'Material (Android)'),
              Tab(text: 'Cupertino (Apple / ios)'),
            ],
          ),
        ),
        body: TabBarView(
          children: [
            // Material Design UI
            MaterialUI(),
            // Cupertino Design UI
            CupertinoUI(),
          ],
        ),
      ),
    );
}

```

```

class MaterialUI extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.all(16.0),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          ElevatedButton(

```



```

        style: ElevatedButton.styleFrom(
          backgroundColor: Colors.blueAccent,
          foregroundColor: Colors.white,
          padding: EdgeInsets.symmetric(horizontal: 24, vertical: 12),
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(8),
          ),
        ),
      ),
      onPressed: () {
        print('Material Button Pressed');
      },
      child: Text('Material Button'),
    ),
  ),
  SizedBox(height: 20),
  ElevatedButton(
    style: ElevatedButton.styleFrom(
      backgroundColor: Colors.blueAccent, // Background color
      foregroundColor: Colors.white, // Text color
      padding: EdgeInsets.symmetric(horizontal: 24, vertical: 12),
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(8),
      ),
    ),
    onPressed: () {
      showDialog(
        context: context,
        builder: (BuildContext context) {
          return AlertDialog(
            title: Text('Material Alert'),
            content: Text('Material Design AlertDialog'),
            actions: <Widget>[
              TextButton(
                onPressed: () {
                  Navigator.of(context).pop();
                },
                child: Text('OK'),
              ),
            ],
          );
        },
      );
    },
  ),
),

```

```

        );
      },
    );
  },
  child: Text('Material Alert'),
),
],
),
);
}
}

```

```

class CupertinoUI extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.all(16.0),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          CupertinoButton(
            color: CupertinoColors.activeBlue,
            child: Text('Cupertino Button'),
            onPressed: () {
              print("Cupertino Button Pressed");
            },
          ),
          SizedBox(height: 20),
          CupertinoButton(
            color: CupertinoColors.activeBlue,
            child: Text('Cupertino Alert'),
            onPressed: () {
              showCupertinoDialog(
                context: context,
                builder: (BuildContext context) {
                  return CupertinoAlertDialog(
                    title: Text('Cupertino Alert'),
                    content: Text('Cupertino Design AlertDialog'),

```

```

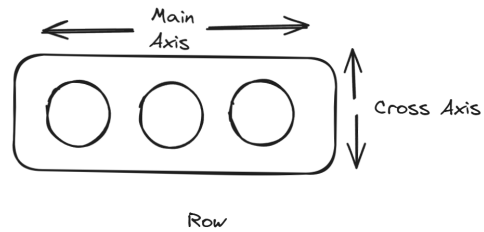
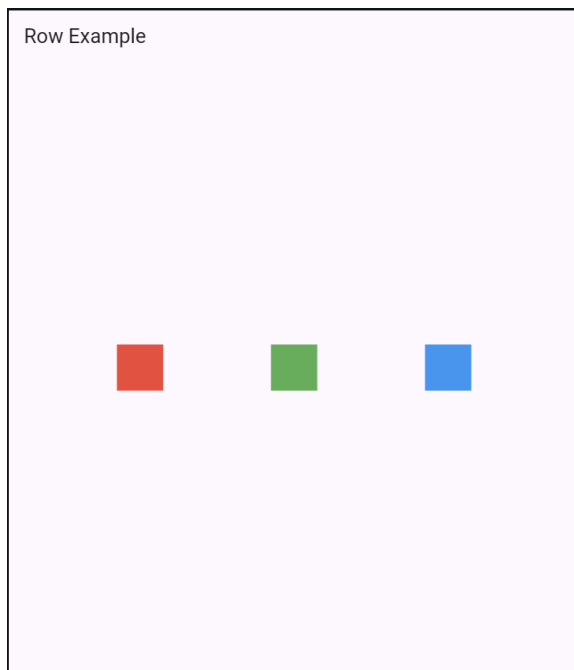
        actions: <Widget>[
          CupertinoDialogAction(
            child: Text('OK'),
            onPressed: () {
              Navigator.of(context).pop();
            },
          ),
        ],
      );
    },
  );
),
],
),
);
}
}

```

1. Row Widget

The `Row` widget in Flutter is used to display its children widgets in a horizontal array. It arranges its children in a single horizontal line, which can be useful for creating layouts where you want elements to be aligned side-by-side.

Here's a simple example:



Key Properties of the `Row` Widget

1. `children` :

- A list of widgets that are arranged horizontally. These can be any widget, such as `Text` , `Icon` , `Container` , etc.

2. `mainAxisAlignment` :

- Defines how the children are aligned along the horizontal axis (main axis).
- Common values include:
 - `MainAxisAlignment.start` : Aligns children to the start of the row.
 - `MainAxisAlignment.end` : Aligns children to the end of the row.
 - `MainAxisAlignment.center` : Centers children within the row.
 - `MainAxisAlignment.spaceBetween` : Distributes children evenly with space between them.
 - `MainAxisAlignment.spaceAround` : Distributes children evenly with space around them.
 - `MainAxisAlignment.spaceEvenly` : Distributes children evenly with equal space between them.

3. `crossAxisAlignment` :

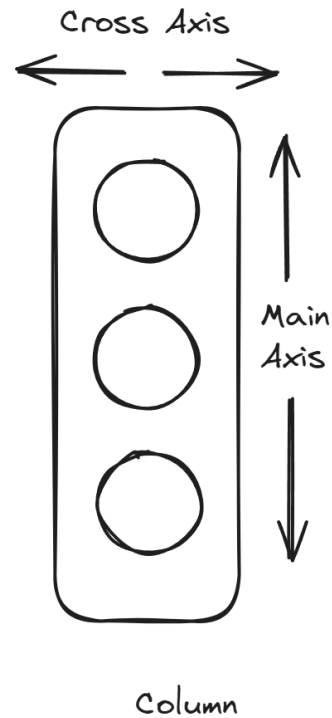
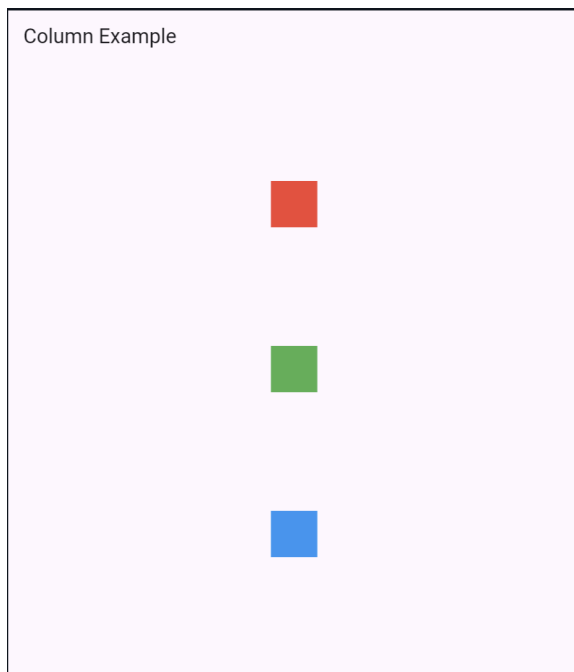
- Defines how the children are aligned along the vertical axis (cross axis).
- Common values include:
 - `CrossAxisAlignment.start` : Aligns children to the start of the row (top for horizontal rows).
 - `CrossAxisAlignment.end` : Aligns children to the end of the row (bottom for horizontal rows).
 - `CrossAxisAlignment.center` : Centers children within the row vertically.
 - `CrossAxisAlignment.stretch` : Stretches children to fill the vertical space.

4. `mainAxisSize` :

- Defines how much space the row should take along the main axis.
- Values include:
 - `MainAxisSize.max` : The row takes up as much horizontal space as possible.
 - `MainAxisSize.min` : The row only takes up as much horizontal space as needed by its children.

2. Column

The `Column` widget in Flutter is used to display its children widgets in a vertical array. It arranges its children in a single vertical line, which is useful for creating layouts where you want elements to be stacked one on top of the other.



Key Properties of the `Column` Widget

1. `children`:

- A list of widgets arranged vertically. These can be any widget, such as `Text`, `Icon`, `Container`, etc.

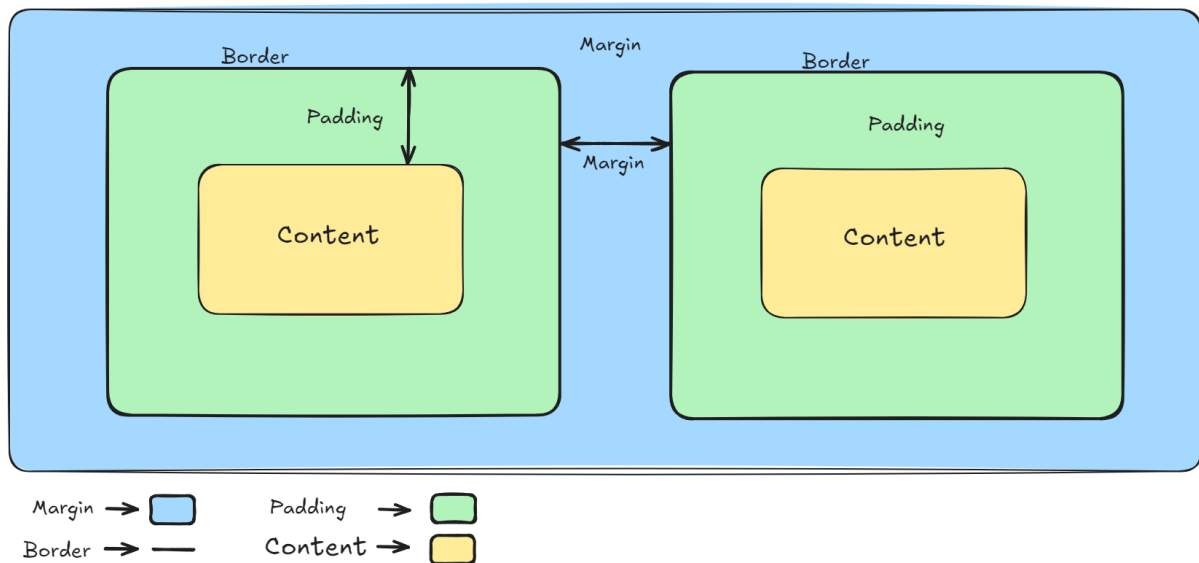
2. `mainAxisAlignment`:

- Defines how the children are aligned along the vertical axis (main axis).
- Common values include:
 - `MainAxisAlignment.start`: Aligns children to the top of the column.
 - `MainAxisAlignment.end`: Aligns children to the bottom of the column.
 - `MainAxisAlignment.center`: Centers children within the column.
 - `MainAxisAlignment.spaceBetween`: Distributes children evenly with space between them.
 - `MainAxisAlignment.spaceAround`: Distributes children evenly with space around them.

- `MainAxisAlignment.spaceEvenly` : Distributes children evenly with equal space between them.
3. `crossAxisAlignment` :
- Defines how the children are aligned along the horizontal axis (cross axis).
 - Common values include:
 - `CrossAxisAlignment.start` : Aligns children to the start of the column (left for vertical columns).
 - `CrossAxisAlignment.end` : Aligns children to the end of the column (right for vertical columns).
 - `CrossAxisAlignment.center` : Centers children within the column horizontally.
 - `CrossAxisAlignment.stretch` : Stretches children to fill the horizontal space.
4. `mainAxisSize` :
- Defines how much space the column should take along the main axis.
 - Values include:
 - `MainAxisSize.max` : The column takes up as much vertical space as possible.
 - `MainAxisSize.min` : The column only takes up as much vertical space as needed by its children.

3. Container

The `Container` widget in Flutter is a versatile and flexible widget used to create rectangular visual elements. It can be used for a variety of purposes such as adding padding, margins, borders, and background colors, or for positioning widgets in a specific layout.



Key Properties of the **Container** Widget

1. **alignment** :
 - Aligns the child widget within the container.
 - Common values include `Alignment.center` , `Alignment.topLeft` , `Alignment.bottomRight` , etc.
2. **padding** :
 - Adds space inside the container between its border and the child widget.
 - Defined using `EdgeInsets` (e.g., `EdgeInsets.all(16.0)`).
3. **margin** :
 - Adds space outside the container, separating it from other widgets.
 - Defined using `EdgeInsets` (e.g., `EdgeInsets.all(16.0)`).
4. **width and height** :
 - Defines the size of the container. If not provided, the container will size itself based on its child and constraints.
5. **color** :
 - Sets the background color of the container.
6. **decoration** :

- Allows more complex styling, such as adding borders, gradients, and shadows.
- Defined using `BoxDecoration` (e.g., `BoxDecoration(color: Colors.blue, borderRadius: BorderRadius.circular(10))`).

7. `child`:

- The widget inside the container. This is where you place the content of the container.

Example Usage

Here's an example demonstrating various properties of the `Container` widget:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      appBar: AppBar(title: Text('Container Widget Example')),
      body: ContainerExample(),
    ),
  ));
}

class ContainerExample extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Container(
            alignment: Alignment.center,
            padding: EdgeInsets.all(16.0),
            margin: EdgeInsets.all(16.0),
            width: 200,
            height: 100,
            color: Colors.blue,
```

```

        child: Text(
          'Blue Container',
          style: TextStyle(color: Colors.white),
        ),
      ),
      SizedBox(height: 20),
      Container(
        alignment: Alignment.center,
        padding: EdgeInsets.all(16.0),
        margin: EdgeInsets.all(16.0),
        decoration: BoxDecoration(
          color: Colors.green,
          borderRadius: BorderRadius.circular(12),
          boxShadow: [
            BoxShadow(
              color: Colors.black.withOpacity(0.5),
              spreadRadius: 2,
              blurRadius: 5,
              offset: Offset(0, 3),
            ),
          ],
        ),
        child: Text(
          'Green Container with Shadow',
          style: TextStyle(color: Colors.white),
        ),
      ),
      SizedBox(height: 20),
      Container(
        alignment: Alignment.center,
        padding: EdgeInsets.all(16.0),
        margin: EdgeInsets.all(16.0),
        width: 200,
        height: 100,
        decoration: BoxDecoration(
          gradient: LinearGradient(
            colors: [Colors.purple, Colors.orange],
          ),
        ),
      ),
    ),
  ),
),

```

```

        borderRadius: BorderRadius.circular(10),
      ),
      child: Text(
        'Gradient Container',
        style: TextStyle(color: Colors.white),
      ),
    ],
  ),
);
}
}

```

Explanation

1. Basic Container:

- **color** : Sets a solid background color (`Colors.blue`).
- **padding** : Adds space inside the container.
- **margin** : Adds space outside the container.
- **width** and **height** : Define the size of the container.
- **child** : Displays a `Text` widget inside the container.

2. Container with Decoration:

- **decoration** : Uses `BoxDecoration` to set a background color, border radius, and shadow.
- **BoxShadow** : Adds a shadow effect to the container.

3. Container with Gradient:

- **decoration** : Uses `BoxDecoration` to set a gradient background.
- **gradient** : Applies a linear gradient from `Colors.purple` to `Colors.orange` .

Key Points

- **Container** is highly customizable and can be used for many layout and styling needs.
- **padding** adds space inside the container.

- `margin` adds space outside the container.
- `decoration` allows for complex styling including borders, gradients, and shadows.
- `alignment` and `child` control the placement and content within the container.

This flexibility makes the `Container` widget a powerful tool for creating various UI elements and layouts in Flutter.

container

column

row

Text

- **Text Widget:** This is like writing something on the wall.

```
Text('Hello, world!')
```

This widget displays the text "Hello, world!" on the screen.

- **Button Widget:** This is like a button you press to ring a doorbell.

```
ElevatedButton(
  onPressed: () {
    // Action when button is pressed
  },
  child: Text('Press Me'),
)
```

This widget creates a button that says "Press Me". When you press it, something happens (like ringing the doorbell).

- **Image Widget:** This is like hanging a picture on the wall.

```
Image.network('<https://example.com/myimage.jpg>')
```

This widget shows an image from the internet.

Widgets Example

Just like you combine different materials to build a house, you can combine different widgets to build your app.

Here's a simple example of combining widgets to create a basic screen with some text and a button:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: DefaultTabController(
        length: 4,
        child: Scaffold(
          appBar: AppBar(
            title: Text('Widget Examples'),
            bottom: TabBar(
              tabs: [
                Tab(text: 'Row'),
                Tab(text: 'Column'),
                Tab(text: 'Text'),
                Tab(text: 'Container'),
              ],
            ),
          ),
          body: TabBarView(
            children: [
              RowWidget(),
              ColumnWidget(),
              TextWidget(),
              ContainerWidget(),
            ],
          ),
        ),
      ),
    );
  }
}
```

```

    ),
  ),
);
}
}

class RowWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: [
        ElevatedButton(
          style: ElevatedButton.styleFrom(
            backgroundColor: Colors.blueAccent, // Backgrou
            nd color
            foregroundColor: Colors.white, // Text color
            padding:
              EdgeInsets.symmetric(horizontal: 24, vertic
            al: 12), // Padding
            shape: RoundedRectangleBorder(
              borderRadius: BorderRadius.circular(8), // Ro
            unded corners
            ),
          ),
          onPressed: () {
            print("Row Button 1 Pressed");
          },
          child: Text('Button 1'),
        ),
        ElevatedButton(
          style: ElevatedButton.styleFrom(
            backgroundColor: Colors.blueAccent, // Backgrou
            nd color
            foregroundColor: Colors.white, // Text color
            padding:
              EdgeInsets.symmetric(horizontal: 24, vertic
            al: 12), // Padding

```

```

        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(8), // Ro
unded corners
        ),
      ),
      onPressed: () {
        print("Row Button 2 Pressed");
      },
      child: Text('Button 2'),
    ),
    ElevatedButton(
      style: ElevatedButton.styleFrom(
        backgroundColor: Colors.blueAccent, // Backgrou
nd color
        foregroundColor: Colors.white, // Text color
        padding:
          EdgeInsets.symmetric(horizontal: 24, vertic
al: 12), // Padding
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(8), // Ro
unded corners
        ),
      ),
      onPressed: () {
        print("Row Button 3 Pressed");
      },
      child: Text('Button 3'),
    ),
  ],
);
}
}

```

```

class ColumnWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,

```

```

        children: [
          ElevatedButton(
            style: ElevatedButton.styleFrom(
              backgroundColor: Colors.blueAccent, // Backgrou
nd color
              foregroundColor: Colors.white, // Text color
              padding:
                EdgeInsets.symmetric(horizontal: 24, vertic
al: 12), // Padding
              shape: RoundedRectangleBorder(
                borderRadius: BorderRadius.circular(8), // Ro
unded corners
              ),
            ),
            onPressed: () {
              print("Column Button 1 Pressed");
            },
            child: Text('Button 1'),
          ),
          ElevatedButton(
            style: ElevatedButton.styleFrom(
              backgroundColor: Colors.blueAccent, // Backgrou
nd color
              foregroundColor: Colors.white, // Text color
              padding:
                EdgeInsets.symmetric(horizontal: 24, vertic
al: 12), // Padding
              shape: RoundedRectangleBorder(
                borderRadius: BorderRadius.circular(8), // Ro
unded corners
              ),
            ),
            onPressed: () {
              print("Column Button 2 Pressed");
            },
            child: Text('Button 2'),
          ),
          ElevatedButton(

```



```

        style: ElevatedButton.styleFrom(
            backgroundColor: Colors.blueAccent, // Backgrou
nd color
            foregroundColor: Colors.white, // Text color
            padding:
                EdgeInsets.symmetric(horizontal: 24, vertic
al: 12), // Padding
            shape: RoundedRectangleBorder(
                borderRadius: BorderRadius.circular(8), // Ro
unded corners
            ),
        ),
        onPressed: () {
            print("Column Button 3 Pressed");
        },
        child: Text('Button 3'),
    ),
],
);
}
}

```

```

class TextWidget extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Center(
            child: Text(
                'Material Design Text Widget',
                style: TextStyle(
                    fontSize: 20, // Font size
                    fontWeight: FontWeight.w500, // Font weight
                    color: Colors.black, // Text color
                    letterSpacing: 1.0, // Letter spacing
                    height: 1.5, // Line height
                ),
            ),
        );
    }
}

```

```

}

class ContainerWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        decoration: BoxDecoration(color: Colors.deepPurple[90
0]),
        padding: EdgeInsets.all(32),
        child: Container(
          padding: EdgeInsets.all(32), // Padding inside the
container
          decoration: BoxDecoration(
            color: Colors.deepPurple[400], // Highlight paddi
ng area
            border: Border.all(color: Colors.black, width: 1
0),
          ),
          child: Text(
            'Content Area',
            style: TextStyle(
              fontSize: 32,
              fontWeight: FontWeight.w500,
              color: Colors.black,
              backgroundColor: Colors.deepPurple[100],
            ),
          ),
        ),
      ),
    );
  }
}

```

In this example:

- **MaterialApp:** The main widget that starts your app.
- **Scaffold:** Provides a structure for the app (like a skeleton for the house).

- **AppBar**: A widget for the app's title bar.
- **Center**: Centers its child widgets.
- **Column**: Arranges its children vertically.
- **Text**: Displays text.
- **ElevatedButton**: A button you can press.

Summary

Widgets in Flutter are like the pieces you use to build your app. Each widget has a specific role, and by combining them, you create the full user interface of your app.

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

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

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData(
        useMaterial3: true,
      ),
      debugShowCheckedModeBanner: false,
      home: DefaultTabController(
        length: 10,
        child: Scaffold(
          appBar: AppBar(
            title: const Text('Widget Examples'),
            bottom: const TabBar(
              isScrollable: true,
              tabs: [
```

```

        Tab(text: 'Row'),
        Tab(text: 'Column'),
        Tab(text: 'Text'),
        Tab(text: 'Container'),
        Tab(text: 'Text Field'),
        Tab(text: 'Image'),
        Tab(text: 'List View'),
        Tab(text: 'Stack'),
        Tab(text: 'Icon'),
        Tab(text: 'Scaffold'),
    ],
  ),
),
body: const TabBarView(
  children: [
    RowWidget(),
    ColumnWidget(),
    TextWidget(),
    ContainerWidget(),
    TextFieldWidget(),
    ImageWidget(),
    ListViewWidget(),
    StackWidget(),
    IconWidget(),
    ScaffoldWidget(),
  ],
),
),
),
);
}
}

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

  @override
  Widget build(BuildContext context) {

```

```

return Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    ElevatedButton(
      style: ElevatedButton.styleFrom(
        backgroundColor: Colors.blueAccent, // Background
        foregroundColor: Colors.white, // Text color
        padding:
          const EdgeInsets.symmetric(horizontal: 24, vertical: 12),
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(8), // Rounded corners
        ),
      ),
      onPressed: () {
        print("Row Button 1 Pressed");
      },
      child: const Text('Button 1'),
    ),
    ElevatedButton(
      style: ElevatedButton.styleFrom(
        backgroundColor: Colors.blueAccent, // Background
        foregroundColor: Colors.white, // Text color
        padding:
          const EdgeInsets.symmetric(horizontal: 24, vertical: 12),
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(8), // Rounded corners
        ),
      ),
      onPressed: () {
        print("Row Button 2 Pressed");
      },
      child: const Text('Button 2'),
    ),
    ElevatedButton(
      style: ElevatedButton.styleFrom(
        backgroundColor: Colors.blueAccent, // Background
        foregroundColor: Colors.white, // Text color
        padding:

```

```

        const EdgeInsets.symmetric(horizontal: 24, vertical: 24),
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(8), // Round corners
        ),
      ),
      onPressed: () {
        print("Row Button 3 Pressed");
      },
      child: const Text('Button 3'),
    ),
  ],
);
}
}

```

```

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

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: [
        ElevatedButton(
          style: ElevatedButton.styleFrom(
            backgroundColor: Colors.blueAccent, // Background color
            foregroundColor: Colors.white, // Text color
            padding:
              const EdgeInsets.symmetric(horizontal: 24, vertical: 24),
            shape: RoundedRectangleBorder(
              borderRadius: BorderRadius.circular(8), // Round corners
            ),
          ),
          onPressed: () {
            print("Column Button 1 Pressed");
          },
          child: const Text('Button 1'),
        ),
      ],
    );
  }
}

```

```

ElevatedButton(
  style: ElevatedButton.styleFrom(
    backgroundColor: Colors.blueAccent, // Background
    foregroundColor: Colors.white, // Text color
    padding:
      const EdgeInsets.symmetric(horizontal: 24, vertical: 12),
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(8), // Rounded corners
    ),
  ),
  onPressed: () {
    print("Column Button 2 Pressed");
  },
  child: const Text('Button 2'),
),
ElevatedButton(
  style: ElevatedButton.styleFrom(
    backgroundColor: Colors.blueAccent, // Background
    foregroundColor: Colors.white, // Text color
    padding:
      const EdgeInsets.symmetric(horizontal: 24, vertical: 12),
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(8), // Rounded corners
    ),
  ),
  onPressed: () {
    print("Column Button 3 Pressed");
  },
  child: const Text('Button 3'),
),
],
);
}
}

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

```

```

@override
Widget build(BuildContext context) {
  return const Center(
    child: Text(
      'Material Design Text Widget',
      style: TextStyle(
        fontSize: 20, // Font size
        fontWeight: FontWeight.w500, // Font weight
        color: Colors.black, // Text color
        letterSpacing: 1.0, // Letter spacing
        height: 1.5, // Line height
      ),
    ),
  );
}

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

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        decoration: BoxDecoration(color: Colors.deepPurple[900])
        padding: const EdgeInsets.all(32),
        child: Container(
          padding: const EdgeInsets.all(32), // Padding inside
          decoration: BoxDecoration(
            color: Colors.deepPurple[400], // Highlight padding
            border: Border.all(color: Colors.black, width: 10),
          ),

          child: Text(
            'Content Area',
            style: TextStyle(
              fontSize: 32,
              fontWeight: FontWeight.w500,

```



```

        color: Colors.black,
        backgroundColor: Colors.deepPurple[100],
      ),
    ),
  ),
));
}
}

```

```

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

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Padding(
        padding: const EdgeInsets.all(32.0),
        child: TextField(
          decoration: InputDecoration(
            labelText: 'Enter your name',
            border: const OutlineInputBorder(),
            prefixIcon: const Icon(Icons.person),
            filled: true,
            fillColor: Colors.grey[200],
          ),
        ),
      ),
    );
  }
}

```

```

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

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Padding(

```

```

padding: const EdgeInsets.all(32.0),
child: Center(
  child: Container(
    width: 200,
    height: 200,
    decoration: BoxDecoration(
      borderRadius: BorderRadius.circular(16),
      boxShadow: [
        BoxShadow(
          color: Colors.black.withOpacity(0.2),
          spreadRadius: 5,
          blurRadius: 7,
          offset: const Offset(0, 3),
        ),
      ],
    ),
    child: ClipRRect(
      borderRadius: BorderRadius.circular(16),
      child: Image.network(
        'https://images.unsplash.com/photo-1721990570',
        fit: BoxFit.cover,
      ),
    ),
  ),
);
}

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

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        padding: const EdgeInsets.all(32.0),

```

```

        child: ListView.builder(
          itemCount: 20,
          itemBuilder: (context, index) {
            return Card(
              child: ListTile(
                leading: const CircleAvatar(
                  backgroundColor: Colors.deepPurpleAccent,
                  child: Icon(Icons.star, color: Colors.white),
                ),
                title: Text('Item $index'),
                subtitle: Text('Subtitle $index'),
                trailing: const Icon(Icons.chevron_right),
                contentPadding:
                  const EdgeInsets.symmetric(horizontal: 16),
              ),
            );
          },
        ),
      ),
    );
  }
}

```

```

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

```

```

  @override

```

```

  Widget build(BuildContext context) {

```

```

    return Center(

```

```

      child: Stack(

```

```

        alignment: Alignment.center,

```

```

        children: [

```

```

          // First Card (bottom-most)

```

```

          Positioned(

```

```

            child: Card(

```

```

              elevation: 8,

```

```

              shape: RoundedRectangleBorder(

```

```

                borderRadius: BorderRadius.circular(16),

```

```

    ),
    child: const SizedBox(
      width: 200,
      height: 200,
      child: Center(child: Text('Card 1')),
    ),
  ),
),
// Second Card (middle)
Positioned(
  left: 10,
  top: 10,
  child: Card(
    elevation: 12,
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(16),
    ),
    child: const SizedBox(
      width: 180,
      height: 180,
      child: Center(child: Text('Card 2')),
    ),
  ),
),
// Third Card (top-most)
Positioned(
  left: 20,
  top: 20,
  child: Card(
    elevation: 16,
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(16),
    ),
    child: const SizedBox(
      width: 160,
      height: 160,
      child: Center(child: Text('Card 3')),
    ),
  ),
),

```

```

        ),
      ),
    ],
  ),
);
}
}

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

  @override
  Widget build(BuildContext context) {
    return const Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Icon(Icons.favorite, size: 100, color: Colors.redAccent),
          SizedBox(height: 20),
          Icon(Icons.thumb_up, size: 100, color: Colors.blueAccent),
          SizedBox(height: 20),
          Icon(Icons.access_alarm, size: 100, color: Colors.deepOrange),
        ],
      ),
    );
  }
}

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Scaffold Example'),
      ),
      drawer: Drawer(

```

```

    child: ListView(
      padding: EdgeInsets.zero,
      children: <Widget>[
        const DrawerHeader(
          decoration: BoxDecoration(
            color: Colors.deepPurpleAccent,
          ),
          child: Text(
            'Drawer Header',
            style: TextStyle(
              color: Colors.white,
              fontSize: 24,
            ),
          ),
        ),
        ListTile(
          title: const Text('Item 1'),
          onTap: () {},
        ),
        ListTile(
          title: const Text('Item 2'),
          onTap: () {},
        ),
      ],
    ),
    body: const Center(
      child: Text('Hello, Scaffold!'),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: () {},
      child: const Icon(Icons.add),
    ),
  );
}
}

```

Gestures

In Flutter, gestures are actions that users can perform with their fingers, like tapping, dragging, or swiping, to interact with the app. Think of gestures as the ways you can communicate with your app using touch.

Example to Understand Gestures

Imagine you have a smartphone. When you touch the screen in different ways, different things happen:

- **Tap:** Like pressing a button on a remote control.
- **Double Tap:** Like double-clicking with a mouse.
- **Long Press:** Like holding down a button.
- **Swipe:** Like flipping through pages in a book.

Common Gestures in Flutter

Flutter provides several widgets to detect and respond to these gestures. Here are a few examples:

Tap Gesture

A simple tap gesture can be detected using the `GestureDetector` widget.

```
GestureDetector(  
  onTap: () {  
    print('Tapped!');  
  },  
  child: Container(  
    color: Colors.blue,  
    width: 100,  
    height: 100,  
    child: Center(  
      child: Text('Tap Me'),  
    ),  
  ),  
)
```

In this example:

- **GestureDetector**: This widget detects various gestures. Here, it detects a tap.
- **onTap**: This callback is triggered when the container is tapped.
- **Container**: A widget that contains a blue box with text "Tap Me".

Long Press Gesture

Detecting a long press gesture is just as easy.

```
GestureDetector(
  onLongPress: () {
    print('Long Pressed!');
  },
  child: Container(
    color: Colors.green,
    width: 100,
    height: 100,
    child: Center(
      child: Text('Long Press Me'),
    ),
  ),
)
```

Here:

- **onLongPress**: This callback is triggered when the container is long-pressed.

Swipe Gesture

You can detect a swipe gesture (or drag) using the `GestureDetector` as well.

```
GestureDetector(
  onPanUpdate: (details) {
    print('Swiped: ${details.delta}');
  },
  child: Container(
    color: Colors.red,
    width: 100,
    height: 100,
```



```

        child: Center(
          child: Text('Swipe Me'),
        ),
      ),
    )
  )
)

```

In this example:

- **onPanUpdate:** This callback is triggered when the container is swiped or dragged.
- **details.delta:** This provides information about the swipe, like the direction and speed.

Combining Gestures

You can detect multiple gestures on the same widget. Here's an example:

```

import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      home: Scaffold(
        appBar: AppBar(
          title: Text('Gesture Examples'),
        ),
        body: GestureExamples(),
      ),
    );
  }
}

class GestureExamples extends StatefulWidget {

```

```

    @override
    _GestureExamplesState createState() => _GestureExamplesState();
  }

class _GestureExamplesState extends State<GestureExamples> {
  String _gestureMessage = 'Perform a gesture';

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          GestureDetector(
            onTap: () => _updateMessage('Tap detected'),
            onDoubleTap: () => _updateMessage('Double Tap detected'),
            onLongPress: () => _updateMessage('Long Press detected'),
            onPanUpdate: (details) => _updateMessage(
              'Pan detected: ${details.localPosition.toString()}'
            ),
            onVerticalDragUpdate: (details) => _updateMessage(
              'Vertical Drag detected: ${details.localPosition.toString()}'
            ),
            child: Container(
              color: Colors.blueAccent,
              width: 500,
              height: 500,
              child: Center(
                child: Text(
                  'Gesture Area',
                  style: TextStyle(color: Colors.white, fontSize: 20),
                ),
              ),
            ),
          ),
        ],
      ),
    );
  }
}

```

```

        ),
      ),
    ),
    SizedBox(height: 20),
    Text(
      _gestureMessage,
      style: TextStyle(fontSize: 16),
    ),
  ],
),
);
}

void _updateMessage(String message) {
  setState(() {
    _gestureMessage = message;
  });
}
}

```

In this example:

- **onTap**: Detects a single tap.
- **onDoubleTap**: Detects a double tap.
- **onLongPress**: Detects a long press.

Summary

Gestures in Flutter allow you to make your app interactive. By using widgets like `GestureDetector`, you can respond to various touch actions such as taps, swipes, and long presses. This makes your app more engaging and user-friendly.

Concept of State

In Flutter, the concept of "state" refers to any data or information that can change over time within your app. The state of a widget is what allows it to

remember information and change its appearance or behavior in response to user actions or other events.

Example to Understand State

Think of a simple counter app where you press a button to increase a number displayed on the screen. The number is the state, and pressing the button changes the state.

StatelessWidget vs StatefulWidget

- **StatelessWidget:** A widget that doesn't change over time. It has no state.
- **StatefulWidget:** A widget that can change over time. It has a state that can change.

Simple StatefulWidget Example: Counter App

Let's create a basic counter app where the number increases each time you press a button.

1. **Create a New Flutter Project:** You can use your IDE or the command line to create a new Flutter project.
2. **Create the Counter Widget:**

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: CounterScreen(),
    );
  }
}

class CounterScreen extends StatefulWidget {
  @override
```

```

    _CounterScreenState createState() => _CounterScreenState
    ();
  }

class _CounterScreenState extends State<CounterScreen> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Counter App'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pressed the button this many time
s:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),

```

```

    ),
  );
}
}

```

Explanation

1. Main Function:

```

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

```

This is the entry point of the app. `runApp` starts the app and takes a widget as an argument.

2. MyApp Widget:

```

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: CounterScreen(),
    );
  }
}

```

This is the main widget of the app, which sets `CounterScreen` as the home screen.

3. CounterScreen StatefulWidget:

```

class CounterScreen extends StatefulWidget {
  @override
  _CounterScreenState createState() => _CounterScreenSta
te();
}

```

`CounterScreen` is a stateful widget that can change over time.

4. `_CounterScreenState`:

```
class _CounterScreenState extends State<CounterScreen> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Counter App'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pressed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

```
    );  
  }  
}
```

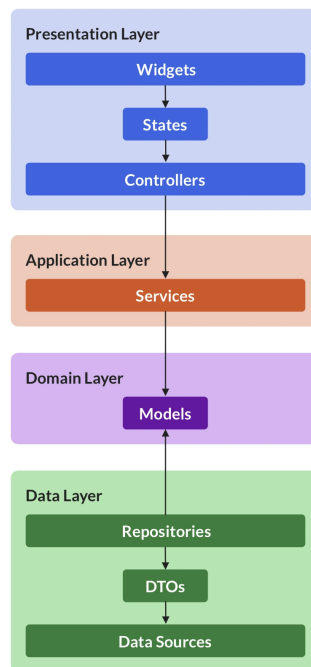
- **_counter**: This variable holds the state, which is the number of times the button has been pressed.
- **_incrementCounter**: This method increases the `_counter` by 1 and calls `setState` to update the UI.
- **build Method**: This method builds the UI. It includes a `Scaffold` with an `AppBar`, a `Center` widget containing a `Column`, and a `FloatingActionButton`.

Summary

In this simple example, the state (`_counter`) is stored in the `_CounterScreenState` class. When the button is pressed, the state is updated, and the UI is rebuilt to reflect the new state. This demonstrates how state in Flutter allows your app to react to user input and update its interface accordingly.

Layers

In modern software development, maintaining a well-organized and scalable codebase is crucial for building robust applications. The layered architecture, often referred to as clean architecture, is a design pattern that divides an application into distinct layers, each responsible for a specific aspect of the system.



1. Presentation Layer

- **Widgets:**
 - In Flutter, widgets are the basic building blocks of the user interface (UI). They define the visual components and structure of the UI, such as buttons, text fields, and layouts. Widgets are used to create the visual representation of the app.
- **States:**
 - State refers to the dynamic data in your application that can change over time. In Flutter, state management is crucial for building interactive and responsive UIs. State holds the current values of the UI components and allows for updating them when needed.
- **Controllers:**
 - Controllers manage the logic and interaction between the UI and the data. They handle user inputs, update the state, and manage the flow of data between the UI (widgets) and the underlying layers. Controllers are responsible for ensuring that the correct data is displayed in the UI and that user actions trigger the appropriate responses.

2. Application Layer

- **Services:**
 - Services contain the business logic of the application. They perform operations that are not directly tied to the UI but are essential for the functioning of the app, such as fetching data from an API, processing user inputs, or executing business rules. Services act as intermediaries between the presentation layer and the domain layer.

3. Domain Layer

- **Models:**
 - Models represent the core data structures and business logic of the application. They define the entities and their relationships, encapsulating the rules and behaviors associated with the data. Models are usually independent of the UI and can be reused across different parts of the application.

4. Data Layer

- **Repositories:**
 - Repositories are responsible for handling data operations such as fetching, storing, and updating data. They abstract the data sources from the rest of the application, providing a consistent interface for accessing data. Repositories ensure that the application does not need to know where the data comes from (e.g., a database, an API, or a local file).
- **DTOs (Data Transfer Objects):**
 - DTOs are simple objects used to transfer data between different layers of the application. They are often used to encapsulate data that needs to be passed from the data layer to the domain or presentation layers. DTOs help in keeping the application layers loosely coupled by providing a common format for data exchange.
- **Data Sources:**
 - Data sources refer to the origin of the data, such as databases, web APIs, or local storage. They provide the raw data that the repositories access and manage. The data layer interacts directly with these sources to retrieve or store information needed by the application.

Summary of Layer Interaction

- The **Presentation Layer** interacts with the **Application Layer** to handle user inputs, display data, and manage UI state.
- The **Application Layer** uses **Services** to execute business logic and interact with the **Domain Layer**.
- The **Domain Layer** contains the core models that represent the business entities and logic.
- The **Data Layer** manages the actual data, with **Repositories** accessing **Data Sources** and using **DTOs** to transfer data across layers.

This architecture promotes separation of concerns, making it easier to manage, test, and maintain the application. Each layer has a distinct responsibility, ensuring that changes in one part of the application do not negatively impact other parts.

Introduction to Dart Programming

Objective: Understand what Dart is and why it's used with Flutter.

What is Dart?

Dart is a free and open-source programming language developed by Google. It's used to create both the backend (server side) and frontend (user side) of applications.

Key Components

1. Dart SDK:

- The Software Development Kit (SDK) includes tools to write and run Dart programs.

2. Dart VM:

- The Dart Virtual Machine is a program that runs Dart code directly.

3. dart2js:

- A tool that converts Dart code into JavaScript. This is useful because not all websites support Dart, but they do support JavaScript.

Simple Example

Here's a basic example to show how Dart works:

```
void main() {  
  print('Hello, world!');  
}
```

This program prints " `Hello, world!` " on the screen.

Key Features

- **Server Side:** Dart can be used to write server code.
- **User Side:** Dart can be used to create user interfaces, especially with Flutter for mobile apps.
- **JavaScript Compatibility:** Dart code can be converted to JavaScript, making it versatile for web development.

Example Breakdown

1. Printing a Message:

```
void main() {  
  print('Hello, world!');  
}
```

- This code is the starting point of a Dart program. It uses the `print` function to display a message.

2. Variables:

```
void main() {  
  var name = 'John Doe';  
  print('Hello, $name!');  
}
```

- Here, `var name = 'John Doe';` creates a variable called `name` and assigns it the value 'John Doe'. The `print` function then uses this variable to display a personalized message.

Features of Dart Programming Language

Dart has several important features:

1. Easy to Understand:

- Dart's syntax is similar to C# and Java, making it familiar to many developers.
- It supports code reuse, keeping programs clean and easy to understand.

2. Object-Oriented Programming (OOP):

- Like Java and C++, Dart follows OOP principles, making it powerful for building complex applications.

3. Open Source:

- Dart is open source, making it popular among individual developers and organizations.

4. Browser Support:

- Dart programs can be converted to JavaScript using the dart2js compiler, ensuring compatibility with all modern web browsers.

5. Type Safe:

- Dart combines static and runtime checks to ensure variables match their defined types, reducing errors.

6. Flexible Compilation and Execution:

- Supports both Just-in-Time (JIT) and Ahead-of-Time (AOT) compilation, offering flexibility in how code is compiled and executed.
- The dart2js tool adds further value by converting Dart code to JavaScript.

7. Asynchronous Programming:

- Dart supports asynchronous programming, allowing it to handle multiple tasks simultaneously and efficiently.

▼ Example of Asynchronous Programming in Dart

```
void main() async {  
  print('Start');  
  
  await Future.delayed(Duration(seconds: 2), () {  
    print('Hello after 2 seconds');  
  });  
  
  print('End');  
}
```

- **Explanation:**

- `print('Start');` is executed first.
- `await Future.delayed(...)` waits for 2 seconds before printing 'Hello after 2 seconds'.
- `print('End');` is executed last, but before the delayed message.

This example demonstrates how Dart can handle tasks asynchronously, allowing other parts of the code to run while waiting.

Variables and Data Types

Variables in Dart

What is a Variable?

A variable is a name assigned to a memory location where data is stored. The type of variable depends on the type of data it stores.

Declaring Variables

- **Single Variable:**

```
type variable_name;
```

- **Multiple Variables:**

```
type variable1, variable2, variable3;
```

Types of Variables

1. **Static Variables**
2. **Dynamic Variables**
3. **Final or Const Variables**

Rules for Variable Names

- Cannot be a **keyword**.
- Can contain alphabets and numbers.
- No spaces or special characters (except `_` and `$`).
- Cannot start with a number.

Example of Variable Declaration

```
void main() {  
  int age = 30;  
  double height = 5.9;  
  bool isStudent = false;  
  String name = "Alice", city = "Wonderland";  
  
  print(age);      // Output: 30  
  print(height);   // Output: 5.9  
  print(isStudent); // Output: false  
  print(name);     // Output: Alice  
  print(city);     // Output: Wonderland  
}
```

Keywords in Dart

- Keywords in Dart are reserved words with predefined functions.
- They cannot be used as variable names or identifiers.
- These keywords help define the structure and syntax of the Dart programming language.

abstract	continue	new	this	as
false	true	final	null	default
throw	finally	do	for	try
catch	get	dynamic	rethrow	typedef
if	else	return	var	break
enum	void	int	String	double
bool	list	map	implements	set
switch	case	while	static	import
export	in	external	this	super
with	class	extends	is	const
yield	factory			

Common Keywords in Dart

All Keywords refer

<https://dart.dev/language/keywords>

Dynamic Variables

- Declared with `dynamic` keyword.
- Can store any type of value.
- Example:

```
void main() {
  dynamic item = "Book";
  print(item); // Output: Book

  item = 10;
  print(item); // Output: 10
}
```

Final and Const Variables

- **Final:** Value assigned once, can be set at runtime.


```
final name = "Alice";  
final String city = "Wonderland";
```

- **Const:** Compile-time constant, value known before runtime.

```
const pi = 3.14;  
const String greeting = "Hello";
```

Null Safety in Dart

Variables can be declared to hold null values by appending `?`.

```
void main() {  
  int? age;  
  age = null;  
  print(age); // Output: null  
}
```

Introduction to Dart Data Types

In Dart, just like in other programming languages such as C, C++, or Java, every variable has a specific type. This type determines what kind of data the variable can hold. Understanding data types is crucial because it helps you handle data correctly in your programs.

1. Numbers in Dart

In Dart, numbers are essential for representing and manipulating numeric values. Dart provides a few different types for handling numbers, and each type has its own characteristics and uses.

Types of Numbers in Dart

1. `int`: Used to represent whole numbers (integers).

- **Syntax:** `int var_name;`
- Example:

```
void main() {
  // Declare an integer
  int num1 = 10;
  print(num1); // Output: 10
}
```

2. **double**: Used to represent numbers with decimal points (floating-point numbers).

- **Syntax:** `double var_name;`
- Example:

```
void main() {
  // Declare a double value
  double num2 = 10.5;
  print(num2); // Output: 10.5
}
```

3. **num**: This is a general type that can hold either `int` or `double`. It is useful when you need a variable that can be either an integer or a floating-point number.

- Example:

```
void main() {
  // Use num to hold different types of numbers
  num num1 = 10; // an integer
  num num2 = 10.5; // a double
  print(num1); // Output: 10
  print(num2); // Output: 10.5
}
```

Parsing Strings to Numbers

Dart allows you to convert strings containing numeric values into actual numbers using the `parse()` function.

Example:

```
void main() {
  // Parsing strings to numbers
  var num1 = num.parse("5");
  var num2 = num.parse("3.14");

  // Adding parsed numbers
  var sum = num1 + num2;
  print("Sum = $sum"); // Output: Sum = 8.14
}
```

Properties of Numbers

- `hashCode`: Gets the hash code of the number.
- `isFinite`: Checks if the number is finite (not infinite).
- `isInfinite`: Checks if the number is infinite.
- `isNaN`: Checks if the number is 'Not a Number' (NaN).
- `isNegative`: Checks if the number is negative.
- `sign`: Returns -1 for negative numbers, 0 for zero, and 1 for positive numbers.
- `isEven`: Checks if the number is even.
- `isOdd`: Checks if the number is odd.

Methods for Numbers

- `abs()`: Returns the absolute value.

```
void main() {
  int number = -5;
  print(number.abs()); // Output: 5
}
```

- `ceil()`: Rounds the number up to the nearest integer.

```
void main() {
  double number = 3.14;
```

```
print(number.ceil()); // Output: 4
}
```

- `floor()` : Rounds the number down to the nearest integer.

```
void main() {
  double number = 3.14;
  print(number.floor()); // Output: 3
}
```

- `compareTo()` : Compares the number with another number.

```
void main() {
  int number1 = 10;
  int number2 = 20;
  print(number1.compareTo(number2)); // Output: -1
}
```

- `remainder()` : Returns the remainder after division.

```
void main() {
  int dividend = 10;
  int divisor = 3;
  print(dividend.remainder(divisor)); // Output: 1
}
```

- `round()` : Rounds the number to the nearest integer.

```
void main() {
  double number = 3.6;
  print(number.round()); // Output: 4
}
```

- `toDouble()` : Converts the number to a double.

```
void main() {
  int number = 10;
```

```
print(number.toDouble()); // Output: 10.0
}
```

- `toInt()` : Converts the number to an integer.

```
void main() {
  double number = 10.5;
  print(number.toInt()); // Output: 10
}
```

- `toString()` : Converts the number to a string.

```
void main() {
  int number = 10;
  print(number.toString()); // Output: "10"
}
```

- `truncate()` : Removes the fractional part of the number.

```
void main() {
  double number = 10.9;
  print(number.truncate()); // Output: 10
}
```

Example: Handling Different Number Types

Example of Type Conversion and Errors:

```
void main() {
  // Correct usage of num
  num variable = 10;
  variable = 10.5; // No error, num can be int or double

  // Error: Cannot assign double to int
  int integerVar = 10;
  // integerVar = 10.5; // This line will cause an error

  // Correct usage of double
```

```
double doubleVar = 10; // Can store an int value
doubleVar = 10.5; // Can also store a double value
```

2. Strings in Dart

Strings in Dart are used to represent sequences of characters. Dart provides various features and methods to handle and manipulate strings efficiently.

Declaring Strings

In Dart, you can declare strings using either single quotes (`' '`) or double quotes (`" "`). Both types of quotes work the same way.

Syntax:

```
String var_name = 'value';
```

Example:

```
void main() {
  String greeting = 'Hello, Dart!';
  print(greeting); // Output: Hello, Dart!
}
```

String Concatenation

You can combine strings using the `+` operator.

Example:

```
void main() {
  String firstName = 'John';
  String lastName = 'Doe';
  String fullName = firstName + ' ' + lastName;
  print(fullName); // Output: John Doe
}
```

String Interpolation

Dart allows you to embed variables inside strings using `${}` within double-quoted strings.

Example:

```
void main() {
  String name = 'Alice';
  int age = 30;
  String introduction = 'Hello, my name is $name and I am
  $age years old.';
  print(introduction); // Output: Hello, my name is Alice a
  nd I am 30 years old.
}
```

For simple variables, you can directly use the variable name without curly braces:

Example:

```
void main() {
  String name = 'Bob';
  String greeting = 'Hi, $name!';
  print(greeting); // Output: Hi, Bob!
}
```

Multi-line Strings

You can create multi-line strings using triple quotes (`'''` or `"""`).

Example:

```
void main() {
  String multiLine = '''This is a string
  that spans across multiple
  lines.''';
  print(multiLine);
  // Output:
  // This is a string
  // that spans across multiple
```

```
// lines.  
}
```

String Methods

Dart provides various methods to manipulate strings. Here are some commonly used methods:

Example: Working with Strings

Example:

```
void main() {  
  String name = 'Alice';  
  String greeting = 'Hello, $name!';  
  
  // Print length  
  print('Length of greeting: ${greeting.length}'); // Output: Length of greeting: 13  
  
  // Convert to uppercase  
  print(greeting.toUpperCase()); // Output: HELLO, ALICE!  
  
  // Trim spaces  
  String padded = '  Dart  ';  
  print(padded.trim()); // Output: Dart  
  
  // Substring  
  print(greeting.substring(7, 12)); // Output: Alice  
  
  // Check if contains  
  print(greeting.contains('Alice')); // Output: true  
  
  // Replace text  
  print(greeting.replaceAll('Alice', 'Bob')); // Output: Hello, Bob!  
  
  // Split text  
  String sentence = 'I love Dart programming.';
```



```
List<String> words = sentence.split(' ');  
print(words); // Output: [I, love, Dart, programming.]  
}
```

This guide should help you understand the basics of working with strings in Dart, including declaring, manipulating, and using various string methods.

Basic String Methods

1. Creating Strings

- `String.fromCharCodeCodes(Iterable<int> charCodes)` : Creates a string from a list of character codes.
- `String.fromCharCode(int charCode)` : Creates a string from a single character code.
- `String.fromEnvironment(String name, {String defaultValue})` : Gets a string from environment variables.

2. Accessing Characters

- `codeUnitAt(int index)` : Returns the UTF-16 code unit at the specified index.
- `runes` : Returns an iterable of the Unicode code points of the string.
- `characters` : Returns an iterable of the string's characters (requires the `characters` package).

3. Querying Strings

- `contains(Pattern pattern, [int start = 0])` : Checks if the string contains the specified pattern.
- `startsWith(Pattern pattern, [int start = 0])` : Checks if the string starts with the specified pattern.
- `endsWith(String other)` : Checks if the string ends with the specified string.
- `isEmpty` : Checks if the string is empty.
- `isNotEmpty` : Checks if the string is not empty.
- `length` : Returns the number of characters in the string.

4. Manipulating Strings

- `toLowerCase()` : Converts all characters in the string to lowercase.

- `toUpperCase()` : Converts all characters in the string to uppercase.
- `trim()` : Removes leading and trailing whitespace from the string.
- `trimLeft()` : Removes leading whitespace from the string.
- `trimRight()` : Removes trailing whitespace from the string.
- `replaceAll(Pattern from, String replace)` : Replaces all occurrences of a pattern with a new string.
- `replaceFirst(Pattern from, String to, [int startIndex = 0])` : Replaces the first occurrence of a pattern with a new string.
- `replaceRange(int start, int end, String replacement)` : Replaces the substring from start to end with a new string.
- `padLeft(int width, [String padding = ' '])` : Pads the string on the left to the specified width.
- `padRight(int width, [String padding = ' '])` : Pads the string on the right to the specified width.
- `substring(int start, [int end])` : Returns the substring from start to end.
- `split(Pattern pattern)` : Splits the string at each occurrence of the pattern and returns a list of substrings.
- `join(Iterable<String> strings)` : Joins a list of strings into a single string.

5. Comparing Strings

- `compareTo(String other)` : Compares this string to another string.
- `==` (equality operator): Checks if two strings are equal.
- `hashCode` : Returns the hash code for the string.

6. Interpolating and Formatting Strings

- String interpolation: `String name = 'World'; print('Hello, $name!');`
- `replaceAllMapped(Pattern from, String replace(Match match))` : Replaces all occurrences of a pattern with a new string computed from a function.

Example Usage

Here's an example demonstrating some of these string methods:

```

void main() {
  String str = 'Hello, World!';

  // Accessing Characters
  print(str.codeUnitAt(0)); // Output: 72
  print(str.runes.toList()); // Output: [72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]

  // Querying Strings
  print(str.contains('world')); // Output: true
  print(str.startsWith('Hello')); // Output: true
  print(str.endsWith('!')); // Output: true
  print(str.isEmpty); // Output: false
  print(str.isNotEmpty); // Output: true
  print(str.length); // Output: 13

  // Manipulating Strings
  print(str.toLowerCase()); // Output: hello, world!
  print(str.toUpperCase()); // Output: HELLO, WORLD!
  print(str.trim()); // Output: Hello, World!
  print(str.replaceAll('World', 'Dart')); // Output: Hello, Dart!
  print(str.substring(0, 5)); // Output: Hello
  print(str.split(', ')); // Output: [Hello, world!]
  print(str.padLeft(15, '*')); // Output: **Hello, World!
  print(str.padRight(15, '*')); // Output: Hello, World!**

  // Interpolating and Formatting Strings
  String name = 'Dart';
  print('Hello, $name!'); // Output: Hello, Dart!

  // Advanced Replacement
  String text = 'The quick brown fox';
  String newText = text.replaceAllMapped(RegExp(r'\\b\\w'),
    (match) => match.group(0)!.toUpperCase());
  print(newText); // Output: The Quick Brown Fox
}

```

3. Booleans

Booleans represent truth values: either `true` or `false`.

Example:

```
void main() {  
  // Declare boolean variables  
  bool isDay = true;  
  bool isNight = false;  
  
  // Print the values  
  print("Is it day? $isDay");  
  print("Is it night? $isNight");  
}
```

Output:

```
Is it day? true  
Is it night? false
```

4. List in Dart

In Dart, a **List** is a collection of items that are ordered and can be accessed by their index. It is similar to arrays in other programming languages. Lists are versatile and can hold various types of data, such as numbers, strings, or even other lists.

Creating Lists

Lists in Dart can be created in several ways. Here's a basic guide to creating and using lists.

1. Declaring an Empty List:

You can declare an empty list and then add elements to it.

```
void main() {  
  List<int> numbers = []; // Empty list of integers
```

```

    numbers.add(10); // Add elements to the list
    numbers.add(20);
    numbers.add(30);

    print(numbers); // Output: [10, 20, 30]
}

```

2. Declaring a List with Initial Elements:

You can declare a list with some initial values.

```

void main() {
    List<String> fruits = ['Apple', 'Banana', 'Cherry'];

    print(fruits); // Output: [Apple, Banana, Cherry]
}

```

3. Fixed-Size List:

You can create a list with a fixed size, where the size is defined at the time of creation. You can then modify the elements, but the size remains fixed.

```

void main() {
    List<String> fixedList = List<String>.filled(3, 'default');

    fixedList[0] = 'Hello';
    fixedList[1] = 'World';
    fixedList[2] = 'Dart';

    print(fixedList); // Output: [Hello, World, Dart]
}

```

4. List with Generated Values:

You can also create a list by generating values using a function.

```

dartCopy code
void main() {
    List<int> generatedList = List<int>.generate(5, (index) =

```

```
> index * 2);

    print(generatedList); // Output: [0, 2, 4, 6, 8]
}
```

Accessing List Elements

You can access elements in a list using their index, which starts at 0.

```
dartCopy code
void main() {
    List<String> colors = ['Red', 'Green', 'Blue'];

    print(colors[0]); // Output: Red
    print(colors[1]); // Output: Green
    print(colors[2]); // Output: Blue
}
```

Modifying List Elements

You can change the value of a specific element by accessing it via its index.

```
dartCopy code
void main() {
    List<String> animals = ['Cat', 'Dog', 'Bird'];

    animals[1] = 'Fish'; // Change 'Dog' to 'Fish'

    print(animals); // Output: [Cat, Fish, Bird]
}
```

Basic List Methods

1. Adding Elements

- `add(E value)` : Adds a single element to the end of the list.
- `addAll(Iterable<E> iterable)` : Adds all elements of the given iterable to the end of the list.

2. Removing Elements

- `remove(Object value)` : Removes the first occurrence of the value from the list.
- `removeAt(int index)` : Removes the element at the specified index.
- `removeLast()` : Removes the last element of the list.
- `removeRange(int start, int end)` : Removes elements from the specified range.

3. Accessing Elements

- `elementAt(int index)` : Returns the element at the specified index.
- `first` : Gets the first element of the list.
- `last` : Gets the last element of the list.
- `single` : Gets the single element of the list, assuming the list has only one element.
- `indexOf(E element)` : Returns the first index of the element, or -1 if not found.
- `lastIndexOf(E element)` : Returns the last index of the element, or -1 if not found.

4. Updating Elements

- `[]` (index operator): Access or update the element at the specified index.
- `insert(int index, E element)` : Inserts an element at the specified index.
- `insertAll(int index, Iterable<E> iterable)` : Inserts all elements of the given iterable at the specified index.

5. Querying Lists

- `contains(Object element)` : Checks if the list contains the specified element.
- `isEmpty` : Checks if the list is empty.
- `isNotEmpty` : Checks if the list is not empty.
- `length` : Returns the number of elements in the list.

6. Sorting and Reversing

- `sort([int compare(E a, E b)])` : Sorts the list in place according to the provided comparison function.

- `reversed` : Returns an iterable with the elements of the list in reverse order.

7. Sublist and Ranges

- `sublist(int start, [int end])` : Returns a new list containing the elements in the specified range.
- `fillRange(int start, int end, [E fillValue])` : Replaces the elements in the specified range with the given value.
- `replaceRange(int start, int end, Iterable<E> newContents)` : Replaces the elements in the specified range with the given iterable.

8. Transformation

- `map<T>(T f(E e))` : Returns a new iterable with the results of applying the function to each element.
- `where(bool test(E element))` : Returns a new iterable with all elements that satisfy the test.
- `forEach(void f(E element))` : Applies the function to each element of the list.
- `reduce(T combine(T value, E element))` : Reduces the list to a single value by repeatedly applying the combine function.
- `fold<T>(T initialValue, T combine(T previousValue, E element))` : Similar to `reduce`, but allows specifying an initial value.
- `join([String separator = ""])` : Returns a string representation of the list, joined by the separator.

9. Copy and Clear

- `toList({bool growable = true})` : Returns a new list containing the elements of the iterable.
- `clear()` : Removes all elements from the list.

Example Usage

Here's an example demonstrating some of these methods:

```
void main() {
  List<int> numbers = [1, 2, 3, 4, 5];
```



```

// Add elements
numbers.add(6);
numbers.addAll([7, 8, 9]);

// Remove elements
numbers.remove(2);
numbers.removeAt(0);
numbers.removeLast();

// Access elements
print(numbers[0]); // Output: 3
print(numbers.first); // Output: 3
print(numbers.last); // Output: 8

// Update elements
numbers[0] = 10;
numbers.insert(1, 15);

// Query list
print(numbers.contains(15)); // Output: true
print(numbers.isEmpty); // Output: false
print(numbers.length); // Output: 6

// Sort and reverse
numbers.sort();
print(numbers); // Output: [3, 4, 5, 6, 7, 10, 15]
print(numbers.reversed.toList()); // Output: [15, 10, 7,
6, 5, 4, 3]

// Sublist and ranges
print(numbers.sublist(1, 3)); // Output: [4, 5]
numbers.fillRange(0, 2, 99);
print(numbers); // Output: [99, 99, 5, 6, 7, 10, 15]

// Transformation
var mapped = numbers.map((e) => e * 2).toList();
print(mapped); // Output: [198, 198, 10, 12, 14, 20, 30]

```

```
// Copy and clear
var copy = numbers.toList();
numbers.clear();
print(numbers); // Output: []
print(copy); // Output: [99, 99, 5, 6, 7, 10, 15]
}
```

5. Maps in Dart

A **Map** in Dart is a collection of key-value pairs, where each key is associated with a value. It is similar to dictionaries in Python or hash maps in other programming languages. Maps are useful when you need to store and retrieve data based on a unique key.

Creating Maps

Maps in Dart can be created in different ways. Here's a basic guide to creating and using maps.

1. Declaring an Empty Map:

You can declare an empty map and then add key-value pairs to it.

```
dartCopy code
void main() {
  // Declaring an empty map
  Map<String, String> emptyMap = {};

  // Adding key-value pairs
  emptyMap['name'] = 'John';
  emptyMap['city'] = 'New York';

  print(emptyMap); // Output: {name: John, city: New York}
}
```

2. Declaring a Map with Initial Key-Value Pairs:

You can declare a map with some initial key-value pairs.

```
dartCopy code
void main() {
  // Declaring a map with initial values
  Map<String, int> ages = {
    'Alice': 30,
    'Bob': 25,
    'Charlie': 35
  };

  print(ages); // Output: {Alice: 30, Bob: 25, Charlie: 35}
}
```

Accessing and Modifying Map Elements

You can access values in a map using their keys and modify them as needed.

```
dartCopy code
void main() {
  Map<String, String> capitals = {
    'USA': 'Washington, D.C.',
    'France': 'Paris',
    'Japan': 'Tokyo'
  };

  // Accessing a value
  print(capitals['France']); // Output: Paris

  // Modifying a value
  capitals['Japan'] = 'Kyoto'; // Change Tokyo to Kyoto

  print(capitals); // Output: {USA: Washington, D.C., France: Paris, Japan: Kyoto}
}
```

Basic Map Methods

1. Adding and Updating Elements

- `putIfAbsent(K key, V ifAbsent())` : Adds a key-value pair to the map if the key is not already present.
- `addAll(Map<K, V> other)` : Adds all key-value pairs from another map to the current map.
- `update(K key, V update(V value), {V ifAbsent()})` : Updates the value for the provided key, or adds it if it does not exist.
- `updateAll(V update(K key, V value))` : Updates all values in the map by applying the provided function.

2. Removing Elements

- `remove(Object key)` : Removes the value for the specified key from the map.
- `clear()` : Removes all key-value pairs from the map.

3. Accessing Elements

- `[]` (index operator): Accesses the value associated with the specified key.
- `containsKey(Object key)` : Checks if the map contains the specified key.
- `containsValue(Object value)` : Checks if the map contains the specified value.
- `entries` : Returns an iterable of the key-value pairs in the map.
- `keys` : Returns an iterable of the keys in the map.
- `values` : Returns an iterable of the values in the map.

4. Querying Maps

- `isEmpty` : Checks if the map is empty.
- `isNotEmpty` : Checks if the map is not empty.
- `length` : Returns the number of key-value pairs in the map.

5. Transforming Maps

- `map<K2, V2>(MapEntry<K2, V2> transform(K key, V value))` : Returns a new map with the results of applying the transform function to each key-value pair.

- `forEach(void action(K key, V value))`: Applies the function to each key-value pair in the map.

6. Getting Default Values

- `putIfAbsent(K key, V ifAbsent())`: Returns the value for the specified key, or adds the key with a value computed by `ifAbsent()` and returns that value.

Example Usage

Here's an example demonstrating some of these methods:

```
void main() {
  Map<String, int> map = {'a': 1, 'b': 2, 'c': 3};

  // Adding and updating elements
  map['d'] = 4; // Add new key-value pair
  map.putIfAbsent('e', () => 5); // Add if key is absent
  map.update('a', (value) => value + 1); // Update existing value
  map.updateAll((key, value) => value * 2); // Update all values

  // Removing elements
  map.remove('b'); // Remove key-value pair by key
  map.clear(); // Clear all key-value pairs

  // Accessing elements
  map = {'a': 1, 'b': 2, 'c': 3};
  print(map['a']); // Access value by key, Output: 1
  print(map.containsKey('a')); // Check if key exists, Output: true
  print(map.containsValue(2)); // Check if value exists, Output: true

  // Querying map
  print(map.isEmpty); // Check if map is empty, Output: false
  print(map.isNotEmpty); // Check if map is not empty, Output: true
}
```

```

put: true
  print(map.length); // Get the number of key-value pairs,
Output: 3

// Transforming maps
map.forEach((key, value) {
  print('Key: $key, Value: $value');
});
// Output:
// Key: a, Value: 1
// Key: b, Value: 2
// Key: c, Value: 3

var newMap = map.map((key, value) => MapEntry(key, value
* 10));
print(newMap); // Output: {a: 10, b: 20, c: 30}

// Getting default values
var value = map.putIfAbsent('d', () => 4);
print(value); // Output: 4
print(map); // Output: {a: 1, b: 2, c: 3, d: 4}

// Accessing keys and values
print(map.keys); // Output: (a, b, c, d)
print(map.values); // Output: (1, 2, 3, 4)

// Entries
print(map.entries); // Output: (MapEntry(a: 1), MapEntry
(b: 2), MapEntry(c: 3), MapEntry(d: 4))
}

```

Explanation:

- **Adding and Updating Elements:** Methods like `putIfAbsent`, `addAll`, and `update` are used to add or update key-value pairs in the map.
- **Removing Elements:** Methods like `remove` and `clear` are used to remove elements from the map.

- **Accessing Elements:** Methods and properties like `containsKey`, `containsValue`, `keys`, `values`, and `entries` provide access to various parts of the map.
- **Querying Maps:** Properties like `isEmpty`, `isNotEmpty`, and `length` help to query the map.
- **Transforming Maps:** Methods like `map` and `forEach` are used to transform or apply functions to the map.
- **Getting Default Values:** The `putIfAbsent` method is used to get default values or add new key-value pairs if the key is absent.

These methods provide powerful ways to manage and manipulate maps in Dart, making them essential for Flutter development when dealing with key-value data structures.

Comments in Dart

Comments help explain code for better understanding. They are not executed by the compiler and serve as documentation.

Types of Comments

1. Single-line Comment:

- Uses `//` to comment a single line.
- Example:

```
void main() {
  double area = 3.14 * 4 * 4;
  // This prints the area of a circle with radius 4
  print(area);
}
```

- Output: `50.24`

2. Multi-line Comment:

- Uses `/* */` to comment multiple lines.
- Example:

```
void main() {
  var lst = [1, 2, 3];
  /* This prints the whole list at once */
  print(lst);
}
```

- Output: `[1, 2, 3]`

3. Documentation Comment:

- Uses `///` for single lines or `/** */` for multiple lines.
- Provide detailed explanations, often used for public APIs and generating documentation.
- Example:

```
/// This function prints a greeting message
void greet() {
  print('Hello, world!');
}

/**
 * This function prints a farewell message.
 * It is used to say goodbye to users.
 */
void farewell() {
  print('Goodbye, world!');
}
```

Summary

- **Single-line Comments:** For brief notes or explanations on a single line.
- **Multi-line Comments:** For commenting out blocks of code or providing longer explanations.
- **Documentation Comments:** For detailed descriptions and generating documentation, typically for functions, classes, or complex logic.

Operators in Dart

- **Operators** are special symbols used to perform operations on values or variables.
- Think of them like tools in a toolbox, each designed for a specific task.

Different types of operators in Dart

The following are the various types of operators in Dart:

- Arithmetic Operators
- Relational Operators
- Type Test Operators
- Bitwise Operators
- Assignment Operators
- Logical Operators
- Conditional Operators
- Cascade Notation Operators

1. Arithmetic Operators

Arithmetic operators are used for basic math operations. They work with numbers to perform calculations.

- **+** (Addition): Adds two numbers.
Example: `5 + 3` results in `8`.
- **-** (Subtraction): Subtracts one number from another.
Example: `5 - 3` results in `2`.
- ***** (Multiplication): Multiplies two numbers.
Example: `5 * 3` results in `15`.
- **/** (Division): Divides one number by another and gives a decimal result.
Example: `6 / 2` results in `3.0`.
- **~/** (Integer Division): Divides and gives the whole number part of the result.
Example: `7 ~/ 2` results in `3`.

- `%` (Modulus): Gives the remainder of a division.

Example: `7 % 2` results in `1`.

Example Program:

```
void main() {
    int a = 2;
    int b = 3;

    print("Sum (a + b) = ${a + b}");
    print("Difference (a - b) = ${a - b}");
    print("Negation -(a - b) = ${-(a - b)}");
    print("Product (a * b) = ${a * b}");
    print("Division (b / a) = ${b / a}");
    print("Quotient (b ~/ a) = ${b ~/ a}");
    print("Remainder (b % a) = ${b % a}");
}
```

2. Relational Operators

Relational operators compare values and give a result that tells whether a condition is true or false.

- `>` (Greater than): Checks if one value is larger than another.

Example: `5 > 3` is `true`.

- `<` (Less than): Checks if one value is smaller than another.

Example: `5 < 3` is `false`.

- `>=` (Greater than or equal to): Checks if one value is larger or equal to another.

Example: `5 >= 3` is `true`.

- `<=` (Less than or equal to): Checks if one value is smaller or equal to another.

Example: `5 <= 3` is `false`.

- `==` (Equal to): Checks if two values are the same.

Example: `5 == 5` is `true`.

- `!=` (Not equal to): Checks if two values are different.

Example: `5 != 3` is `true`.

Example Program:

```
void main() {  
    int a = 2;  
    int b = 3;  
  
    print("a > b: ${a > b}");  
    print("a < b: ${a < b}");  
    print("a >= b: ${a >= b}");  
    print("a <= b: ${a <= b}");  
    print("a == b: ${a == b}");  
    print("a != b: ${a != b}");  
}
```

3. Type Test Operators

These operators are used to check the type of a value.

- `is`: Checks if a value is of a specific type.

Example: `"Hello" is String` is `true`.

- `is!`: Checks if a value is not of a specific type.

Example: `3.3 is! int` is `true`.

Example Program:

```
void main() {  
    String a = 'Hello';  
    double b = 3.3;  
  
    print(a is String); // true  
    print(b is! int);   // true  
}
```

5. Assignment Operators

Assignment operators are used to set or update the value of a variable.

- `=` (Equal to): Assigns a value to a variable.

Example: `int a = 5;`

- `??=` (Null-aware assignment): Assigns a value only if the variable is `null`.

Example: `int? a; a ??= 10;` sets `a` to `10` if `a` is `null`.

Example Program:

```
void main() {  
    int a = 5;  
    int b = 7;  
  
    var c = a * b;  
    print("c = $c");  
  
    var d;  
    d ??= a + b;  
    print("d = $d");  
  
    d ??= a - b;  
    print("d = $d"); // d remains 12  
}
```

6. Logical Operators

Logical operators are used to combine multiple conditions.

- `&&` (AND): Returns `true` if both conditions are `true`.

Example: `(5 > 3) && (7 > 4)` is `true`.

- `||` (OR): Returns `true` if at least one condition is `true`.

Example: `(5 > 3) || (7 < 4)` is `true`.

- `!` (NOT): Reverses the result of a condition.

Example: `!(5 > 3)` is `false`.

Example Program:

```

void main() {
  bool a = true;
  bool b = false;

  print("a && b: ${a && b}");
  print("a || b: ${a || b}");
  print("!a: ${!a}");
}

```

7. Conditional Operators

Conditional operators choose between two values based on a condition.

- `condition ? expression1 : expression2`: Executes `expression1` if `condition` is `true`, otherwise `expression2`.

Example: `(5 > 3) ? "Yes" : "No"` results in `"Yes"`.

- `expression1 ?? expression2`: Returns `expression1` if it's not `null`, otherwise `expression2`.

Example: `null ?? "Default"` results in `"Default"`.

Example Program:

```

void main() {
  int a = 5;

  var result = (a < 10) ? "Correct" : "Wrong";
  print(result);


  int? n;
  var value = n ?? "n is null";
  print(value);

  n = 10;
  value = n;
  print(value);
}

```

8. Cascade Notation Operators

Cascade notation allows performing multiple operations on the same object.

-  (Cascade): Allows you to call multiple methods on the same object.**Example:**

Example Program:

```
class Calc {
  int a = 0;
  int b = 0;

  void set(x, y) {
    this.a = x;
    this.b = y;
  }

  void add() {
    var z = this.a + this.b;
    print(z);
  }
}

void main() {
  Calc calculator = new Calc();
  Calc calculator2 = new Calc();

  calculator.set(1, 2);
  calculator.add();

  calculator2
    ..set(3, 4)
    ..add();
}
```

Standard Input and Output

In Dart, you can interact with the user through the console using standard input and output. Here's how you can do it:

Standard Input

To read input from the user, you need to use the `dart:io` library. This library allows you to access the standard input (keyboard) and output (console). The `stdin.readLineSync()` function is commonly used to read user input as a string.

1. Taking a String Input

Here's how to take a string input from the user:

```
import 'dart:io';

void main() {
  print("Enter your name:");

  // Reading the user's name
  String? name = stdin.readLineSync();

  // Printing a greeting message
  print("Hello, $name! \nWelcome to Dart!!");
}
```

Example:

- **Input:** `John`
- **Output:** `Hello, John! Welcome to Dart!!`

2. Taking an Integer Input

To take an integer input, you first read the input as a string and then convert it to an integer using `int.parse()`:

```
import 'dart:io';

void main() {
  print("Enter your favourite number:");

  // Reading and converting input to integer
  int? number = int.parse(stdin.readLineSync()!);
}
```

```
// Printing the number
print("Your favourite number is $number");
}
```

Example:

- **Input:** 42
- **Output:** Your favourite number is 42

Standard Output

Dart provides two ways to display output:

1. Using `print`

The `print()` function adds a newline after the output:

```
import 'dart:io';

void main() {
  print("Welcome to Dart!");
}
```

2. Using `stdout.write`

The `stdout.write()` function prints text without adding a newline:

```
import 'dart:io';

void main() {
  stdout.write("Welcome to Dart!");
}
```

Comparison:

- **Using `print`**: Moves to the next line after printing.
- **Using `stdout.write`**: Stays on the same line.

Example with both methods:


```
import 'dart:io';

void main() {
  print("Welcome to Dart! // from print");
  stdout.write("Welcome to Dart! // from stdout.write()");
}
```

Output:

```
Welcome to Dart! // from print
Welcome to Dart! // from stdout.write()
```

Example: Simple Addition Program

Here's a complete example that reads two numbers from the user, adds them, and prints the result:

```
import 'dart:io';

void main() {
  print("-----Addition Program-----");
  print("Enter the first number:");
  int? num1 = int.parse(stdin.readLineSync());

  print("Enter the second number:");
  int? num2 = int.parse(stdin.readLineSync());

  // Adding the numbers
  int sum = num1 + num2;

  // Printing the result
  print("Sum is $sum");
}
```

Example Execution:

- **Input:** 5, 7
- **Output:** Sum is 12

Conditional Statements in Dart

Conditional statements in Dart are used to make decisions in your code. They allow you to execute different blocks of code based on certain conditions. Here's an overview of the main types of conditional statements in Dart:

1. `if` Statement

The `if` statement is used to execute a block of code if a specified condition is true.

Syntax:

```
if (condition) {  
    // Code to execute if condition is true  
}
```

Example:

```
void main() {  
    int number = 10;  
  
    if (number > 5) {  
        print('The number is greater than 5');  
    }  
}
```

2. `if-else` Statement

The `if-else` statement allows you to execute one block of code if a condition is true and another block if the condition is false.

Syntax:

```
if (condition) {  
    // Code to execute if condition is true  
} else {  
    // Code to execute if condition is false  
}
```

Example:

```

void main() {
    int number = 3;

    if (number > 5) {
        print('The number is greater than 5');
    } else {
        print('The number is 5 or less');
    }
}

```

3. **if-else if-else** Statement

This structure allows you to check multiple conditions sequentially.

Syntax:

```

if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition1 is false and condition2
    is true
} else {
    // Code to execute if none of the above conditions are tr
    ue
}

```

Example:

```

void main() {
    int number = 7;

    if (number > 10) {
        print('The number is greater than 10');
    } else if (number > 5) {
        print('The number is greater than 5 but 10 or less');
    } else {
        print('The number is 5 or less');
    }
}

```

```
}  
}
```

4. `switch` Statement

The `switch` statement is used to select one of many code blocks to execute. It compares the value of a variable to a series of cases and executes the code block associated with the matching case.

Syntax:

```
switch (expression) {  
  case value1:  
    // Code to execute if expression equals value1  
    break;  
  case value2:  
    // Code to execute if expression equals value2  
    break;  
  default:  
    // Code to execute if expression does not match any cas  
e  
}
```

Example:

```
void main() {  
  String day = 'Monday';  
  
  switch (day) {  
    case 'Monday':  
      print('Start of the work week');  
      break;  
    case 'Friday':  
      print('End of the work week');  
      break;  
    case 'Saturday':  
    case 'Sunday':  
      print('Weekend');  
      break;  
  }
```

```
    default:
      print('Invalid day');
  }
}
```

5. Conditional Expressions (Ternary Operator)

The ternary operator is a shorthand for `if-else` statements. It allows you to write a simple conditional statement in a single line.

Syntax:

```
condition ? expressionIfTrue : expressionIfFalse;
```

Example:

```
void main() {
  int number = 10;
  String result = number > 5 ? 'The number is greater than
5' : 'The number is 5 or less';
  print(result);
}
```

Summary

- **if Statement:** Executes code if a condition is true.
- **if-else Statement:** Executes one block if the condition is true, another if it's false.
- **if-else if-else Statement:** Handles multiple conditions sequentially.
- **switch Statement:** Matches a variable against multiple values and executes corresponding code.
- **Conditional Expressions:** Provides a concise way to perform conditional operations.

Understanding and using these conditional statements effectively will help you control the flow of your Dart programs and handle different scenarios based on your program's requirements.

Looping and Control Flow in Dart

Looping and control flow statements are essential for executing repetitive tasks and controlling the flow of a program based on various conditions.

1. For Loop

The `for` loop is used to execute a block of code a specific number of times. It is useful when you know beforehand how many times you want to iterate.

Syntax:

```
for (initialization; condition; increment) {  
    // Code to execute in each iteration  
}
```

Example:

```
void main() {  
    for (int i = 0; i < 5; i++) {  
        print(i);  
    }  
}
```

Explanation:

- **Initialization:** `int i = 0` sets up the loop variable.
- **Condition:** `i < 5` is checked before each iteration; the loop continues as long as it is true.
- **Increment:** `i++` increments the loop variable after each iteration.

2. While Loop

The `while` loop repeatedly executes a block of code as long as a specified condition is true. It is used when you don't know in advance how many times the loop will run.

Syntax:

```
while (condition) {  
    // Code to execute as long as condition is true  
}
```

```
}
```

Example:

```
void main() {  
    int i = 0;  
    while (i < 5) {  
        print(i);  
        i++;  
    }  
}
```

Explanation:

- The loop continues to execute as long as `i < 5` is true.
- The loop variable `i` is incremented inside the loop to eventually break the loop when the condition becomes false.

3. Do-While Loop

The `do-while` loop is similar to the `while` loop but ensures that the code block is executed at least once before checking the condition.

Syntax:

```
do {  
    // Code to execute  
} while (condition);
```

Example:

```
void main() {  
    int i = 0;  
    do {  
        print(i);  
        i++;  
    } while (i < 5);  
}
```

Explanation:

- The code inside the `do` block executes first, then the condition is checked.
- The loop continues as long as `i < 5` is true.

4. For-in Loop

The `for-in` loop is used to iterate over elements in a collection such as a list or a map. It is useful for looping through items in a collection without needing an index.

Syntax:

```
for (var item in collection) {  
    // Code to execute for each item in the collection  
}
```

Example:

```
void main() {  
    List<String> fruits = ['Apple', 'Banana', 'Cherry'];  
    for (var fruit in fruits) {  
        print(fruit);  
    }  
}
```

Explanation:

- The loop iterates over each element in the `fruits` list, assigning each element to `fruit`.

5. Break Statement

The `break` statement is used to exit a loop before its condition becomes false. It immediately terminates the innermost loop.

Example:

```
void main() {  
    for (int i = 0; i < 10; i++) {  
        if (i == 5) {  
            break;  
        }  
        print(i);  
    }  
}
```



```
}  
}
```

Explanation:

- The loop will terminate when `i` equals 5 due to the `break` statement.

6. Continue Statement

The `continue` statement is used to skip the rest of the code inside the current iteration of a loop and proceed with the next iteration.

Example:

```
void main() {  
    for (int i = 0; i < 10; i++) {  
        if (i % 2 == 0) {  
            continue;  
        }  
        print(i);  
    }  
}
```

Explanation:

- The `continue` statement skips the `print(i)` statement for even numbers and only prints odd numbers.

7. Labels

Labels can be used with `break` and `continue` statements to specify which loop to break out of or continue. This is useful in nested loops.

Syntax:

```
outerLoop: for (int i = 0; i < 5; i++) {  
    for (int j = 0; j < 5; j++) {  
        if (j == 3) {  
            break outerLoop;  
        }  
        print('$i, $j');
```

```
}  
}
```

Explanation:

- The `break outerLoop;` statement exits the outer loop when `j` equals 3.

Summary

- **for Loop:** Executes code a specific number of times.
- **while Loop:** Executes code as long as a condition is true.
- **do-while Loop:** Executes code at least once before checking the condition.
- **for-in Loop:** Iterates over elements in a collection.
- **break Statement:** Exits the innermost loop.
- **continue Statement:** Skips to the next iteration of the loop.
- **Labels:** Used to control nested loops with `break` and `continue`.

Functions in Dart

Functions are blocks of code designed to perform a specific task. They help in organizing and reusing code, making programs more modular and easier to manage. Here's a guide to understanding and using functions in Dart:

1. Built-in Functions

Built-in functions are predefined functions provided by the programming language. They are readily available and do not require you to define them. In Dart (the language used with Flutter), these functions cover a wide range of tasks, such as working with strings, numbers, collections, and more.

Examples of Built-in Functions in Dart:

- `print()`: Prints a message to the console.
 - Example:

```
void main() {  
    print('Hello, World!');
```

```
}
```

- `int.parse()` : Converts a string to an integer.

- Example:

```
void main() {  
  String numberString = '42';  
  int number = int.parse(numberString);  
  print(number); // Output: 42  
}
```

- `List.add()` : Adds an element to a list.

- Example:

```
void main() {  
  List<String> fruits = ['Apple', 'Banana'];  
  fruits.add('Orange');  
  print(fruits); // Output: [Apple, Banana, Orange]  
}
```

- `sqrt()` : Computes the square root of a number. (Requires importing the `dart:math` library).

- Example:

```
import 'dart:math';  
  
void main() {  
  double result = sqrt(16);  
  print(result); // Output: 4.0  
}
```

2. User-Defined Functions

User-defined functions are those that you create yourself to perform specific tasks that are not covered by built-in functions. They allow you to encapsulate reusable logic in your code.

1. Defining a Function

In Dart, you define a function using the `returnType` followed by the `functionName`, a list of parameters enclosed in parentheses, and a block of code enclosed in curly braces.

Syntax:

```
returnType functionName(parameters) {  
    // Code to execute  
    return value; // Return a value if returnType is not v  
oid  
}
```

Example:

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

Explanation:

- `int` is the return type, indicating the function returns an integer.
- `add` is the function name.
- `(int a, int b)` are the parameters.
- The function returns the sum of `a` and `b`.

2. Calling a Function

To use a function, you call it by its name and provide the necessary arguments.

Syntax:

```
functionName(arguments);
```

Example:

```
void main() {  
    int result = add(5, 3);  
}
```

```
    print(result); // Output: 8
}
```

Explanation:

- `add(5, 3)` calls the `add` function with arguments `5` and `3`.
- The result of `add(5, 3)` is stored in the `result` variable and printed.

3. Functions with No Return Value

If a function does not return a value, you use `void` as the return type.

Example:

```
void greet(String name) {
    print('Hello, $name!');
}
```

Explanation:

- The `greet` function takes a single `String` parameter and prints a greeting message.

4. Optional Parameters

Dart functions can have optional parameters. You can use either named or positional optional parameters.

- **Positional Optional Parameters:** Enclosed in square brackets.

Example:

```
void printDetails(String name, [int age = 0]) {
    print('Name: $name, Age: $age');
}
```

Explanation:

- `age` is an optional parameter with a default value of `0`.

Usage:

```
void main() {
    printDetails('Alice'); // Output: Name: Alice, Age: 0
}
```

```
printDetails('Bob', 25); // Output: Name: Bob, Age: 25
}
```

- **Named Optional Parameters:** Enclosed in curly braces.

Example:

```
void printDetails({required String name, int age = 0}) {
  print('Name: $name, Age: $age');
}
```

Explanation:

- `name` is a required named parameter.
- `age` is an optional named parameter with a default value of `0`.

Usage:

```
void main() {
  printDetails(name: 'Alice'); // Output: Name: Alice, Age: 0
  printDetails(name: 'Bob', age: 25); // Output: Name: Bob, Age: 25
}
```

5. Arrow Functions

For simple functions with a single expression, you can use the arrow syntax for conciseness.

Syntax:

```
returnType functionName(parameters) => expression;
```

Example:

```
int add(int a, int b) => a + b;
```

Explanation:

- This is a shorthand for the function that returns `a + b`.

6. Anonymous Functions

Anonymous functions (or lambda functions) are functions without a name. They are often used as arguments to other functions.

Syntax:

```
(parameters) => expression;
```

Example:

```
void main() {  
    var numbers = [1, 2, 3, 4];  
    var doubled = numbers.map((number) => number * 2);  
    print(doubled); // Output: (2, 4, 6, 8)  
}
```

Explanation:

- The anonymous function `(number) => number * 2` is used to double each number in the list.

7. Higher-Order Functions

Functions that take other functions as parameters or return functions are called higher-order functions.

Example:

```
void performOperation(int a, int b, int Function(int, in  
t) operation) {  
    print(operation(a, b));  
}  
  
int add(int x, int y) => x + y;  
int multiply(int x, int y) => x * y;  
  
void main() {  
    performOperation(5, 3, add); // Output: 8  
    performOperation(5, 3, multiply); // Output: 15  
}
```

Explanation:

- `performOperation` takes two integers and a function `operation` that takes two integers and returns an integer.

Summary

- **Defining a Function:** Use `returnType functionName(parameters) { ... }`.
- **Calling a Function:** Use `functionName(arguments);`.
- **No Return Value:** Use `void` as the return type.
- **Optional Parameters:** Use square brackets `[]` for positional and curly braces `{ }` for named parameters.
- **Arrow Functions:** Shorter syntax for functions with a single expression.
- **Anonymous Functions:** Functions without names, used as arguments.
- **Higher-Order Functions:** Functions that take other functions as arguments or return functions.

Differences Between Built-in and User-Defined Functions:

- **Origin:**
 - Built-in functions are provided by the programming language.
 - User-defined functions are created by the programmer.
- **Availability:**
 - Built-in functions are always available and can be used without any prior definition.
 - User-defined functions must be defined before they can be used.
- **Flexibility:**
 - Built-in functions have fixed functionality as defined by the language.
 - User-defined functions can be customized to perform any task you need.

When to Use Each:

- **Built-in Functions:** Use these whenever possible, as they are optimized and tested for performance.
- **User-Defined Functions:** Use these when you need custom logic that is not provided by built-in functions or when you want to encapsulate repeated code for better reusability and readability.

In summary, both built-in and user-defined functions are essential in programming, allowing for efficient and modular code development. Built-in functions handle common tasks, while user-defined functions enable you to implement specific logic as needed.

Understanding and using functions effectively allows you to create well-organized, reusable, and maintainable Dart code.

Object-Oriented Programming (OOP) in Dart

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design and structure software. Dart, being an object-oriented language, allows you to use OOP principles to create reusable, modular, and maintainable code. Here's a breakdown of the core concepts of OOP in Dart:

1. Classes and Objects

- **Class:** A blueprint for creating objects. It defines the properties (attributes) and methods (functions) that the objects created from the class will have.
- **Object:** An instance of a class. It is created based on the class blueprint and can use the properties and methods defined in the class.

Example:

```
// Define a class
class Person {
  // Properties
  String name;
  int age;

  // Constructor
  Person(this.name, this.age);
}
```

```

// Method
void greet() {
    print('Hello, my name is $name and I am $age years old.');
```

```
}
```

```
}
```

```

void main() {
    // Create an object of the class
    Person person = Person('Alice', 30);
    person.greet(); // Output: Hello, my name is Alice and I
    am 30 years old.
}
```

this Keyword

In Dart, the **this** keyword refers to the current instance of the class in which it is used. It is primarily used to distinguish between class properties and parameters with the same name, and to access the current object's members.

Key Uses of **this** Keyword

1. **Accessing Instance Variables:** Use **this** to access instance variables and methods of the current object.
2. **Differentiating Between Instance Variables and Parameters:** Use **this** to differentiate between instance variables and parameters or local variables when they have the same name.
3. **Passing the Current Object:** Use **this** to pass the current object as a parameter to other methods or constructors.

Syntax and Examples

1. Accessing Instance Variables

When you want to refer to instance variables within a class method, you can use **this** to clarify that you are referring to the current object's variables.

```

class Person {
    String name;
    int age;

    Person(this.name, this.age); // Constructor with parameter names same as instance variables

    void printInfo() {
        print('Name: ${this.name}');
        print('Age: ${this.age}');
    }
}

void main() {
    Person person = Person('Alice', 30);
    person.printInfo();
}

```

Output:

```

Name: Alice
Age: 30

```

In this example, `this.name` and `this.age` are used to access the instance variables `name` and `age`.

2. Differentiating Between Instance Variables and Parameters

When constructor parameters or local variables have the same names as instance variables, `this` helps to distinguish them.

```

class Rectangle {
    int width;
    int height;

    Rectangle(int width, int height) {
        this.width = width; // `this.width` refers to the instance variable
        this.height = height; // `this.height` refers to the instance variable
    }
}

```

```

    instance variable
  }

  void display() {
    print('Width: $width, Height: $height');
  }
}

void main() {
  Rectangle rect = Rectangle(10, 20);
  rect.display();
}

```

Output:

```
Width: 10, Height: 20
```

In this case, `this.width` and `this.height` are used in the constructor to assign values to the instance variables.

3. Passing the Current Object

You can use `this` to pass the current object as a parameter to other methods or constructors.

```

class Box {
  int length;

  Box(this.length);

  void describe(Box other) {
    print('Comparing this box with another box of length
    ${other.length}');
  }

  void compare() {
    describe(this); // Passing the current object to the me
    thod
  }
}

```

```

}

void main() {
  Box box1 = Box(15);
  Box box2 = Box(10);
  box1.compare(); // Output: Comparing this box with another box of length 15
}

```

Output:

```

Comparing this box with another box of length 15

```

In this example, `this` is used in the `compare` method to pass the current object `box1` to the `describe` method.

Summary

- `this` refers to the current instance of a class.
- It is useful for accessing instance variables and methods.
- It helps in distinguishing between instance variables and parameters with the same name.
- It can be used to pass the current object to other methods or constructors.

Using `this` appropriately helps in writing clear and concise object-oriented code in Dart.

2. Encapsulation

Encapsulation is the concept of hiding the internal state of an object and requiring all interaction to be performed through an object's methods. This is done to protect the object's integrity by preventing outside interference and misuse.

Example:

```

class BankAccount {
  // Private property
  double _balance;
}

```

```

// Constructor
BankAccount(this._balance);

// Method to deposit money
void deposit(double amount) {
  if (amount > 0) {
    _balance += amount;
  }
}

// Method to withdraw money
void withdraw(double amount) {
  if (amount > 0 && amount <= _balance) {
    _balance -= amount;
  }
}

// Method to get the balance
double getBalance() {
  return _balance;
}
}

void main() {
  BankAccount account = BankAccount(1000);
  account.deposit(500);
  account.withdraw(200);
  print('Balance: ${account.getBalance()}'); // Output: Balance: 1300.0
}

```

3. Inheritance

Inheritance allows a class to inherit properties and methods from another class. The class that inherits is called the "subclass" or "derived class," and the class being inherited from is called the "superclass" or "base class."

Example:

```
// Base class
class Animal {
  void eat() {
    print('Animal is eating.');
```

```
}
}
```

```
// Subclass
class Dog extends Animal {
  void bark() {
    print('Dog is barking.');
```

```
}
}
```

```
void main() {
  Dog dog = Dog();
  dog.eat(); // Output: Animal is eating.
  dog.bark(); // Output: Dog is barking.
}
```

4. Polymorphism

Polymorphism allows methods to do different things based on the object it is acting upon. It enables a single interface to represent different underlying forms (data types).

Example:

```
// Base class
class Shape {
  void draw() {
    print('Drawing a shape.');
```

```
}
}
```

```
// Subclass
class Circle extends Shape {
  @override
  void draw() {
```

```

        print('Drawing a circle.');
```

5. Abstraction

Abstraction involves hiding complex implementation details and showing only the necessary features of an object. This is typically achieved using abstract classes.

Example:

```

// Abstract class
abstract class Shape {
    void draw(); // Abstract method
}

// Concrete class
class Triangle extends Shape {
    @override
    void draw() {
        print('Drawing a triangle.');
```



```

}

void main() {
  Shape shape = Triangle();
  shape.draw(); // Output: Drawing a triangle.
}

```

6. Constructors and Initialization

Constructors are special methods used to initialize objects. Dart allows you to define named and default constructors.

Example:

```

class Car {
  String make;
  String model;

  // Default constructor
  Car(this.make, this.model);

  // Named constructor
  Car.withDefaults() : make = 'Unknown', model = 'Unknown';
}

void main() {
  Car car1 = Car('Toyota', 'Corolla');
  print('Car 1: ${car1.make} ${car1.model}'); // Output: Car
  r 1: Toyota Corolla

  Car car2 = Car.withDefaults();
  print('Car 2: ${car2.make} ${car2.model}'); // Output: Ca
  r 2: Unknown Unknown
}

```

Summary

- **Classes and Objects:** Define classes to create objects with specific properties and methods.

- **Encapsulation:** Hide the internal state of an object and provide methods to interact with it.
 - **Inheritance:** Create subclasses that inherit properties and methods from a superclass.
 - **Polymorphism:** Use a single method or interface to represent different underlying forms.
 - **Abstraction:** Hide complex implementation details and expose only the essential features.
 - **Constructors and Initialization:** Use constructors to initialize objects, with options for default and named constructors.
-