MS228
Advanced
Mobile Application Development

Dr. Jay Nanavati

Teaching Sessions

Type of Course	Compulsory
No. of theory sessions per week No. of practical sessions per week Total Contact Hours per week	03 (Three theory sessions of 1 hr. each) 03 (Single practical session of 3 hrs.) 06
Approx. total no. of theory sessions in semester Approx. total no. of practical sessions in semester Approx. total hours in semester	36 (12 weeks X 3 sessions) 36 (12 weeks X 3 sessions) 72
No. of credits	06

Course Outline

Examination Scheme	
Theory	Internal: 30 marks **(Unit tests: 06, Internal test: 09, Case study-Assignments: 10, Attendance: 05)** External: 70 marks (Written Examination: 70 marks) Total: 100 marks
Practical	Internal: 30 marks **(Tests: 10, Internal test: 10, Termwork/Labwork: 05, Attendance: 05)** External: 70 marks (Coding: 50 marks, Viva Voce: 20 marks) Total: 100 marks
Total marks	200

Course Contents

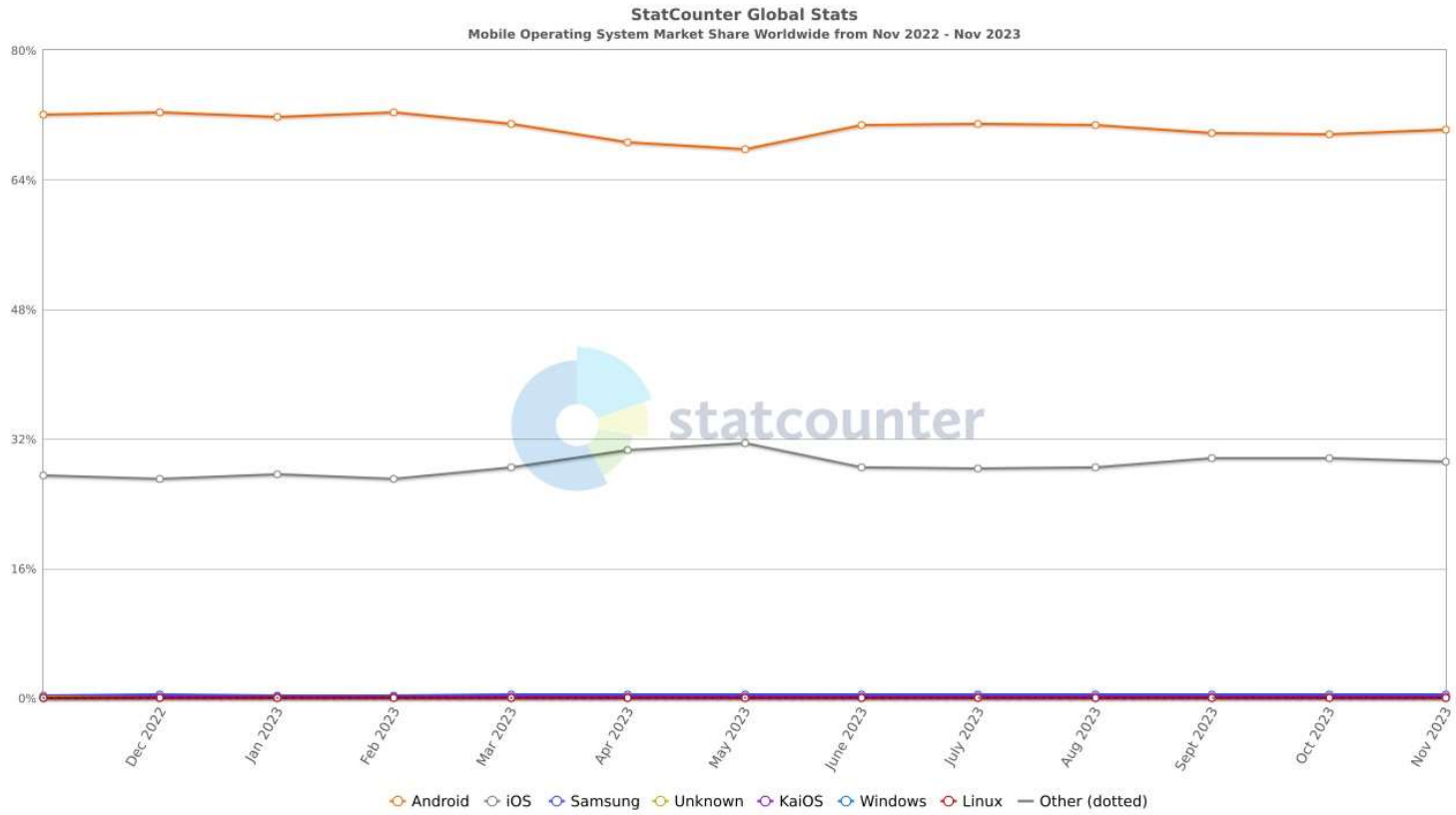
Unit Number	Title of the Unit	Minimum Number of Hours	
		Theory	Practical
1	Introduction to Cross-platform advance with Flutter & Dart	**05**	**36**
2	UI Designing with Flutter	**07**	
3	State management with App creation	**05**	
4	Architecting Flutter applications and its packages	**08**	
5	Introduction to Backend to Flutter	**05**	
6	Firebase and State Management with Flutter	**06**	

Mobile Application Development

Android vs iOS and others

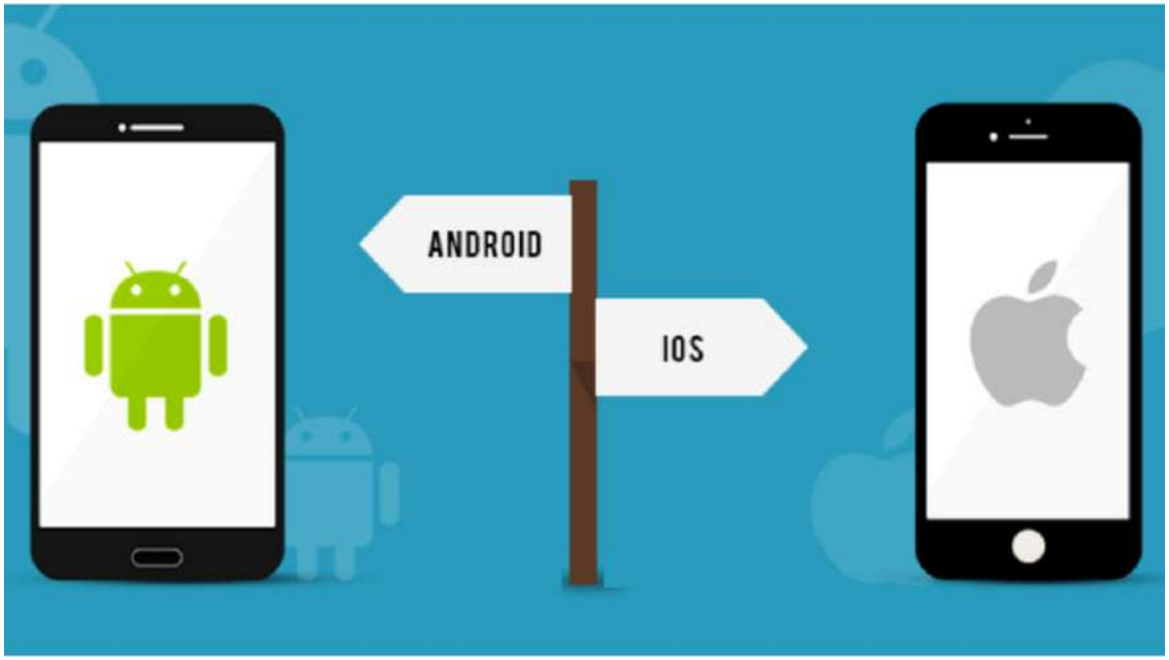
Source: https://gs.statcounter.com/os-market-share/mobile/worldwide

Android vs iOS and others



Android vs iOS

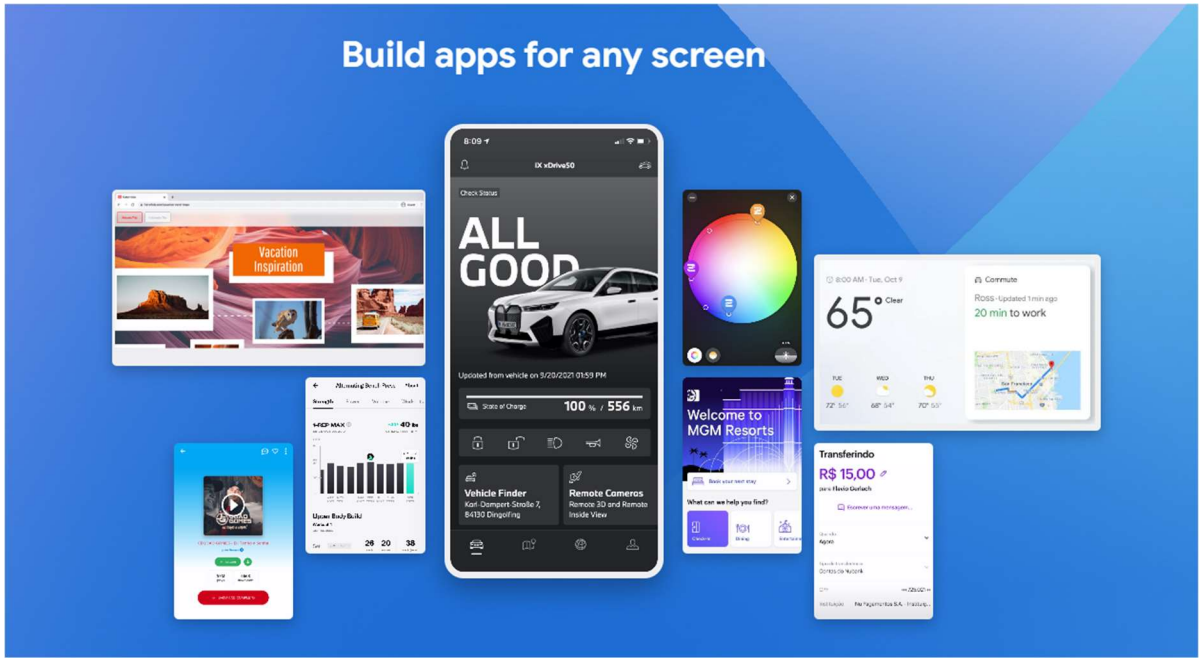
Android vs iOS



Android + iOS

Mobile Application Development using Flutter

Mobile Application Development using Flutter



Introduction to Flutter

- Flutter is Google's portable UI framework for building modern, native, and reactive applications for iOS and Android.
- It is used to develop **cross-platform (also known as multi-platform)** applications for Android, iOS, Linux, macOS, Windows and the web from a single codebase.
- Flutter was released in May 2017.
- The first version of Flutter was known as "Sky" and ran on the Android operating system.
- Official website: flutter.dev
- Latest version: 3.19.5 (as on 11th April, 2024)

Flutter for many platforms

- Mobile [Android, iOS]
- Web [Chrome, Edge, Firefox etc.]
- Desktop [Windows, macOS, Linux]
- Embedded [IoT]

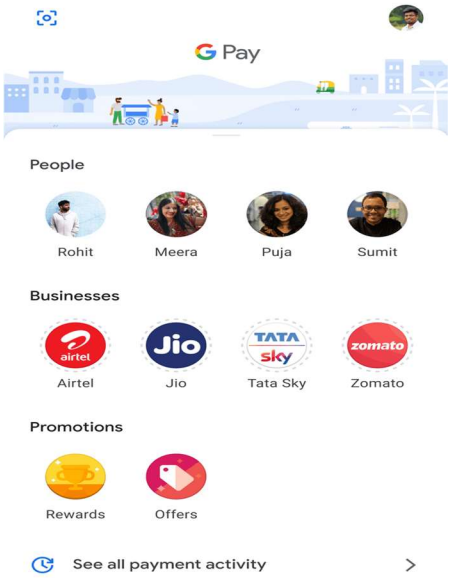
More about Flutter

- Flutter is a framework and **not a programming language**.
- **Dart** is the programming language which is used to develop Flutter applications.
- Flutter is cross-platform because it is coded in Dart, and Dart is cross-platform.
- Flutter uses Dart to create your user interface, removing the need to use separate languages like XML Markup or visual designers.
- Flutter uses the Skia 2D rendering engine that works with different types of hardware and software platforms and is also used by Google Chrome, Chrome OS, Android, Mozilla Firefox, Firefox OS, and others.

Flutter in Real-life Applications

?

Flutter in Real-life Applications



GPay (Google Pay)

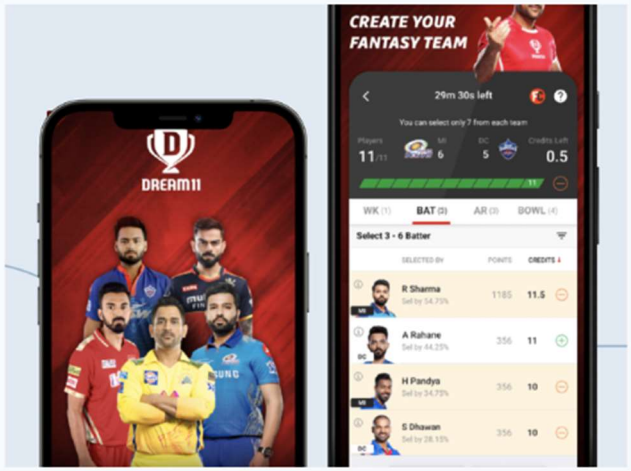
Going global at Google Pay with Flutter

Success metrics

70%
reduction in engineering effort

35%
reduction in lines of code

Flutter in Real-life Applications



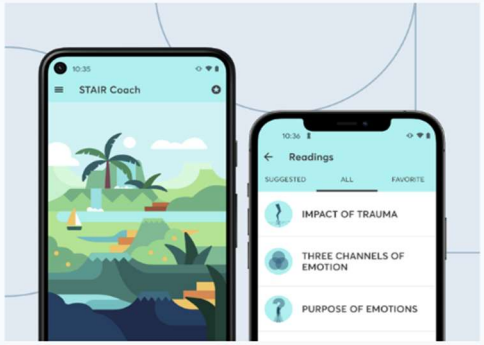
Dream11

Supporting 50 million fantasy sports users in India

Flutter in Real-life Applications

Toyota

Improving infotainment systems at Toyota with Flutter



US Department of Veterans Affairs

STAIR: Helping veterans acclimate back into civilian life with Flutter

ByteDance

Increasing productivity by 33% at ByteDance with Flutter

Flutter in Real-life Applications

The My BMW App

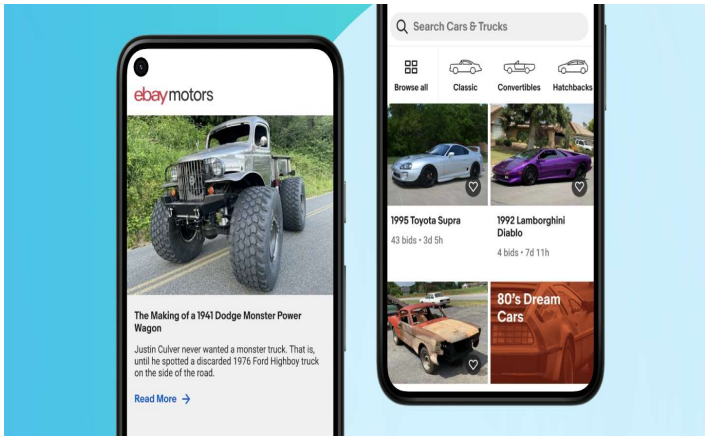
Flutter in Real-life Applications

The Xianyu app

Alibaba, the world's biggest online commerce company, used Flutter to create Xianyu app, which has 50M+ downloads!

This app is a platform for buying and selling used goods.

Flutter in Real-life Applications



ebay motors

Delighting engineers at eBay with Flutter

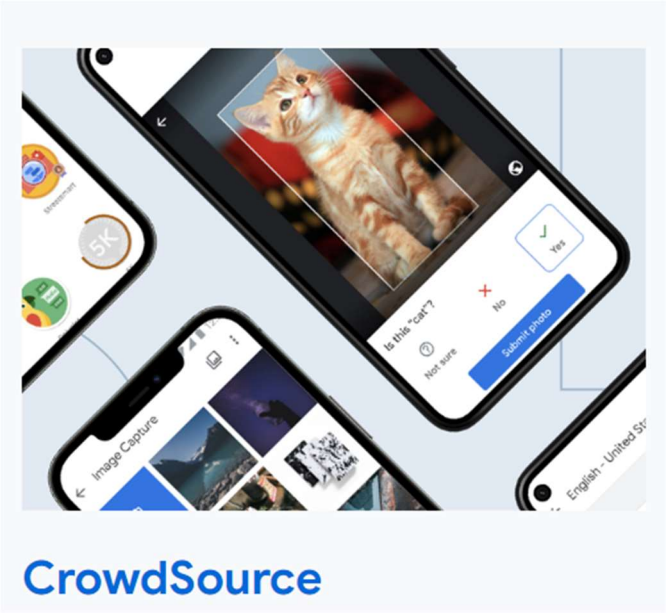
Success metrics

98.3%
shared code

70%
developers believe development is 2x faster than native solutions

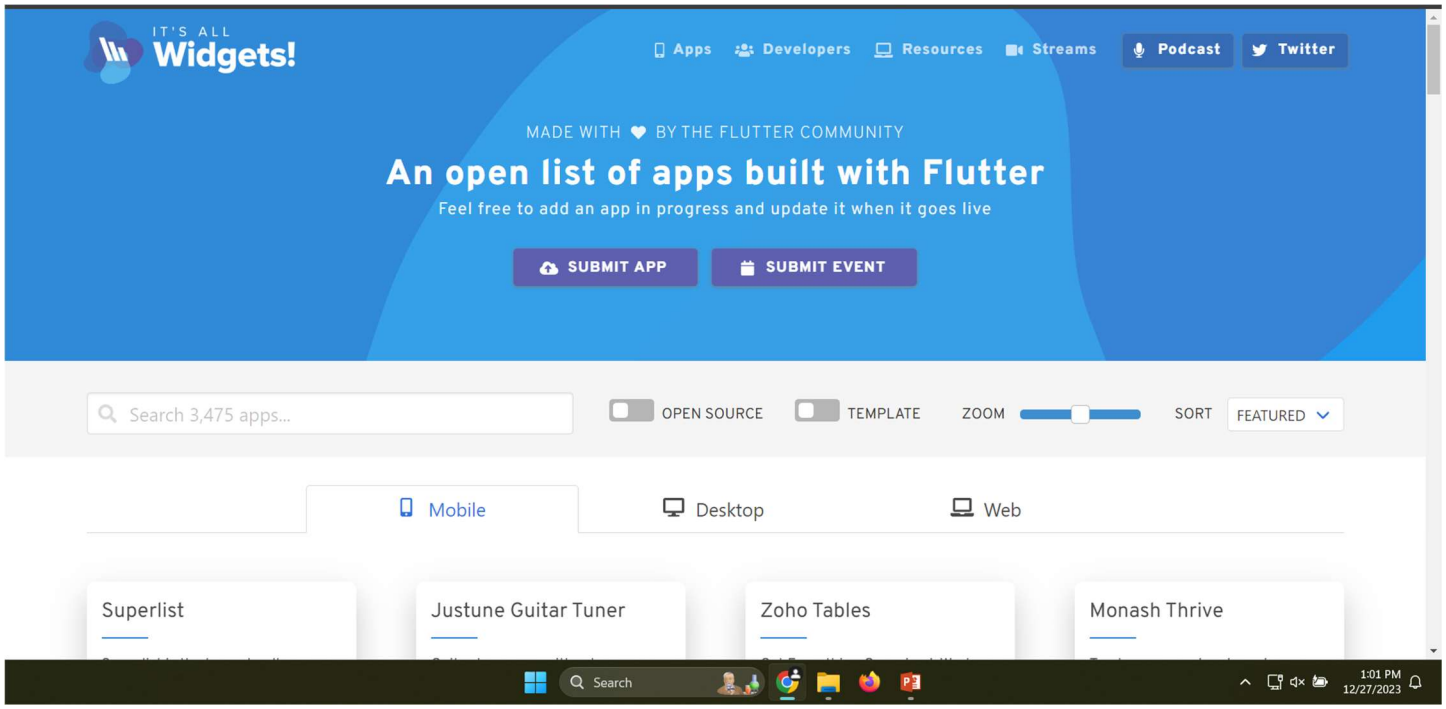
100%
developers say they prefer over iOS or Android

Flutter in Real-life Applications



Crowdsource by Google is a fun, gamified way for millions of people to help train Google's AI and machine learning models, helping products like Google Translate, Maps, and Photos better serve users across regions and cultures.
Currently, over three million users from 190 countries contribute to Crowdsource

Flutter in total



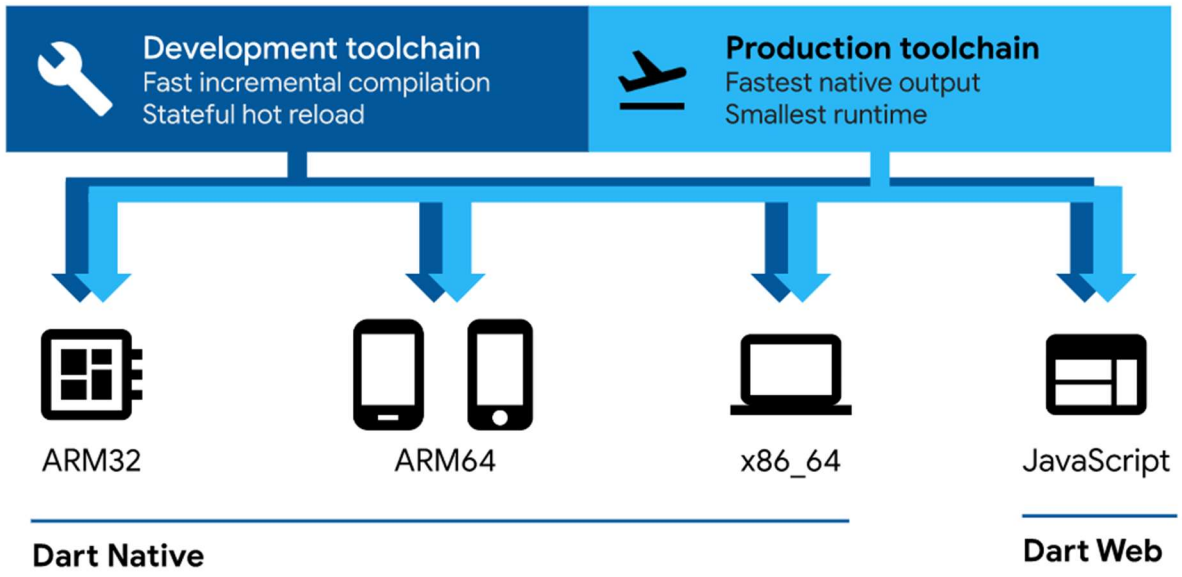
https://itsallwidgets.com/

Cross-platform ability of Dart

- Dart's compiler technology lets you run code in different ways:

1. Native platform: For apps targeting mobile and desktop devices, Dart includes both a **Dart VM with just-in-time (JIT) compilation and an ahead-of-time (AOT) compiler** for producing ***machine code***.
2. Web platform: For apps targeting the web, Dart can compile for development or production purposes. Its web compiler translates Dart into JavaScript.

Cross-platform ability of Dart



Installing Flutter and Configuring Android Studio for Flutter Development

- Download and extract Flutter.
- Download and install Android Studio.
- Configure Android Studio.

Online Flutter IDEs

https://dartpad.dev/

https://flutterstudio.app/

https://zapp.run/

Widgets

- Widgets are building blocks which will be assembled together to make a UI of a Flutter application.
- Every object in your Flutter application's UI is a widget. Structure is defined with widgets, styles are defined with widgets, even animations and routing is handled by widgets.
- Widgets are Dart classes that know how to describe their view.
- Flutter offers several widgets out of the box, which the Flutter developers will combine to make custom widgets and robust UI.
- Each widget has many properties. These properties are assigned values and the values control appearance and behavior of the widgets.

Top-level Widgets in a Flutter application

MaterialApp

MaterialApp Widget is the **starting point** of your app, it tells Flutter that you are going to use Material components and follow the material design in your app.

MaterialApp is a widget that introduces a number of widgets Navigator, Theme that are required to build a material design app.

Scaffold

Scaffold Widget is used under MaterialApp, it gives you many basic functionalities, like AppBar, BottomNavigationBar, Drawer, FloatingActionButton, etc.

The Scaffold is designed to be the single top-level container for a MaterialApp although it is not necessary to nest a Scaffold.

The runApp() method

The runApp() method is used to execute the app and display the root widget on the screen.

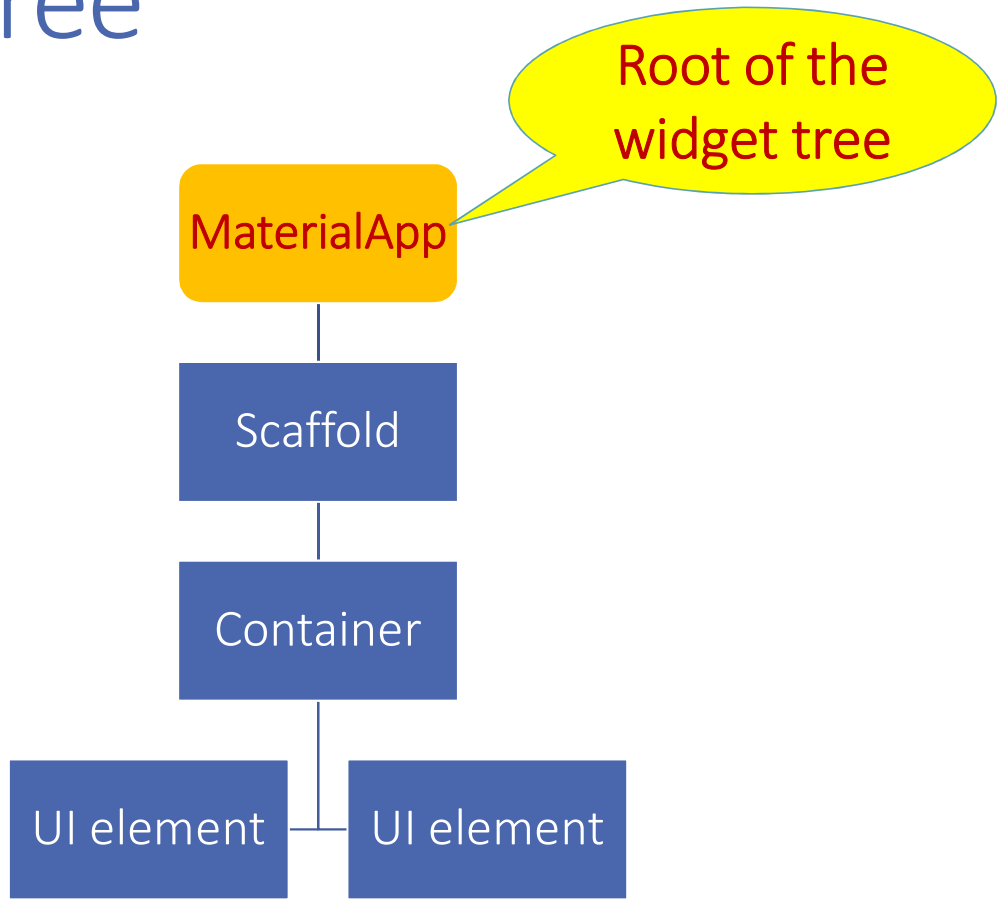
It takes in a Widget as its argument, which is typically the **root** of the app's widget hierarchy. This widget is then passed to the Flutter Engine to be rendered on the screen.

The runApp() method triggers the build process.

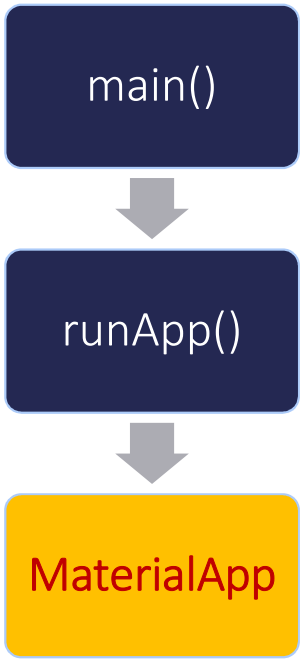
Difference between main() and runApp():

The main() function contains the logic to initialize the app, such as setting up any dependencies or configurations that are required.

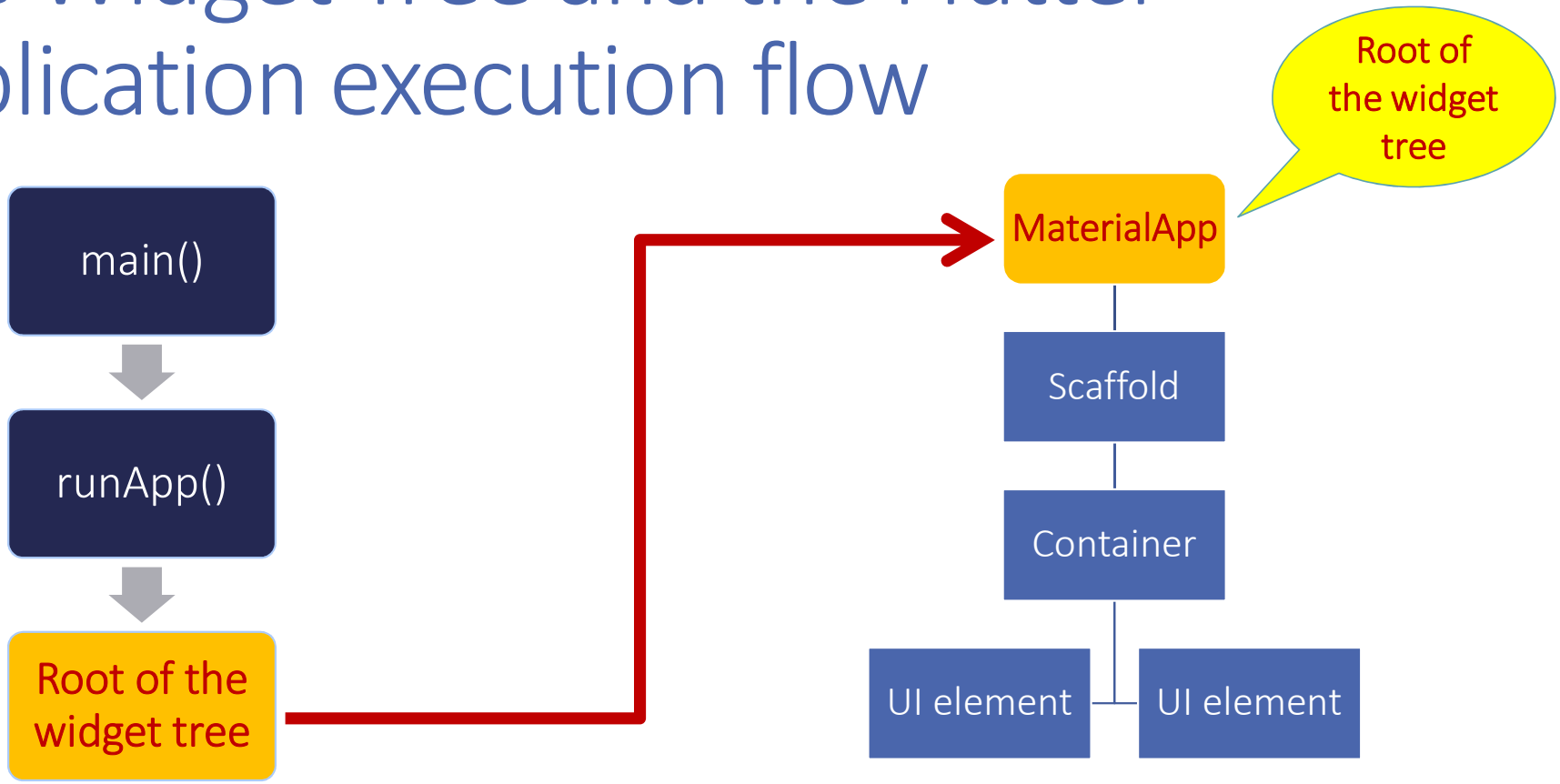
The Widget Tree



Flutter application execution flow

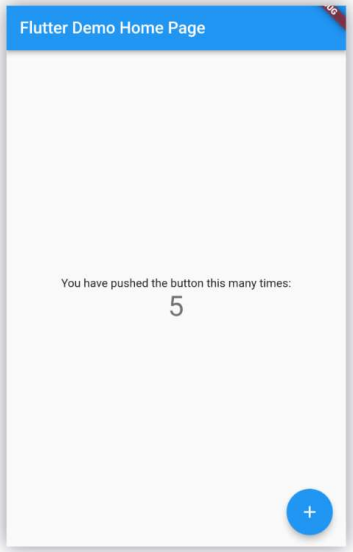


The Widget Tree and the Flutter application execution flow

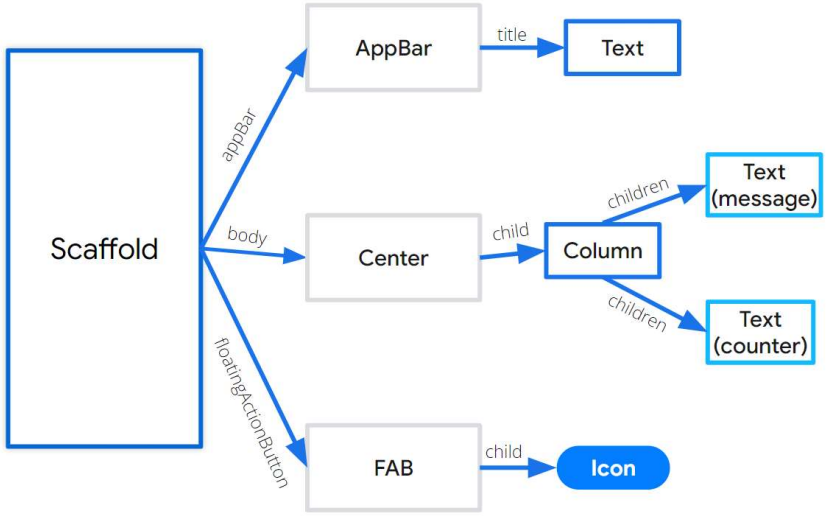


Widget Tree Example

UI



Breakdown



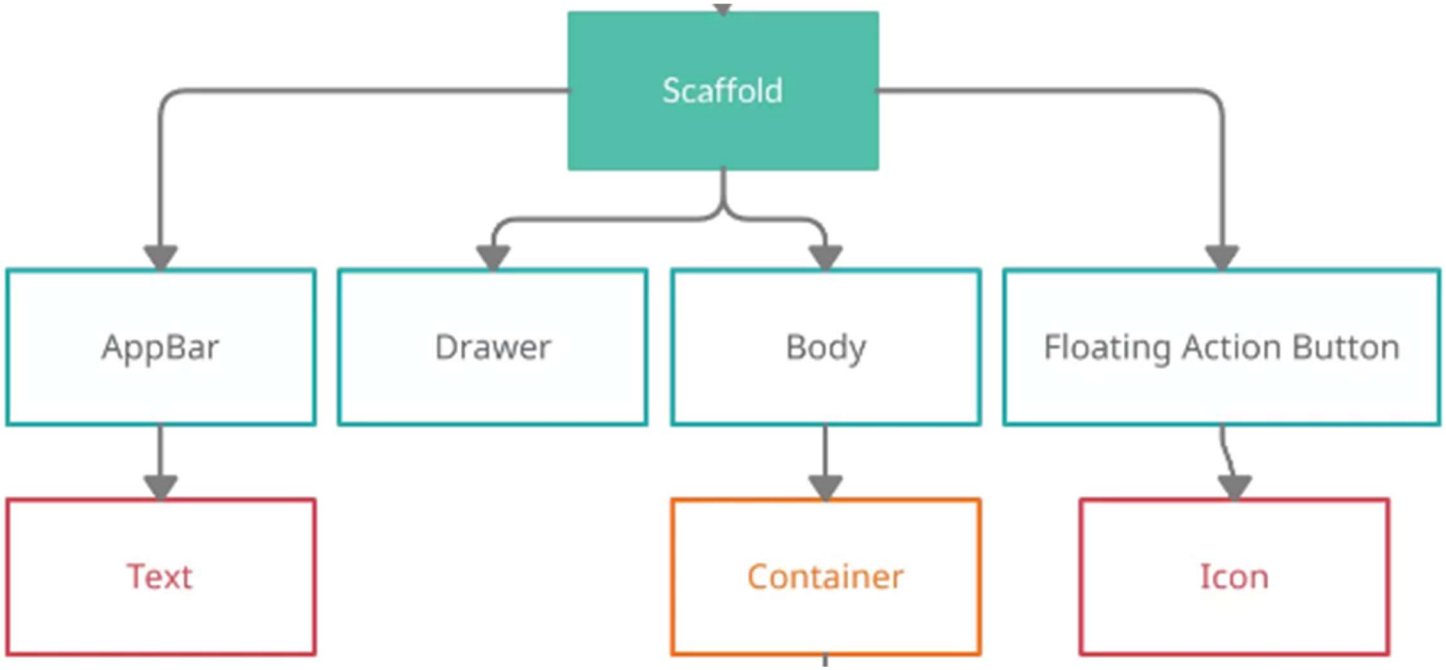
Widgets

- The runApp() function takes the given Widget and makes it the **root** of the widget tree.
- A widget's main job is to implement a **build()** function, which describes the widget in terms of other, lower-level widgets.

The MaterialApp widget

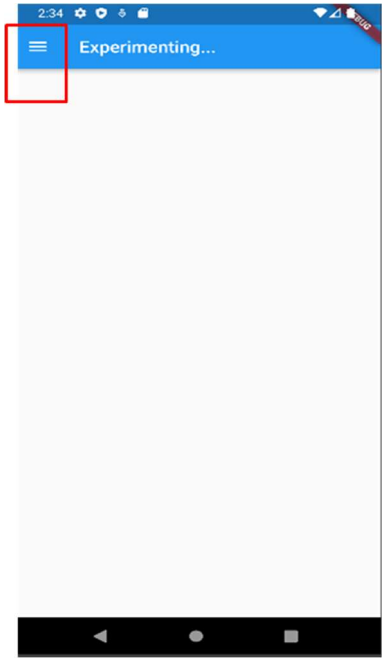
- color: It controls the primary color used in the application.
- darkTheme: It provides theme data for the dark theme for the application.
- debugShowCheckedModeBanner: This property takes a boolean as the object to decide whether to show the debug banner or not.
- home: This property takes a Scaffold widget as the object to show on the default route of the app.
- theme: This property takes a ThemeData class as the object to describe the theme for the MaterialApp.
- title: The title property takes a string as the object to decide the one-line description of the app for the device. When the user press the recent apps button on mobile the text proceeded in title is displayed.

The Scaffold widget



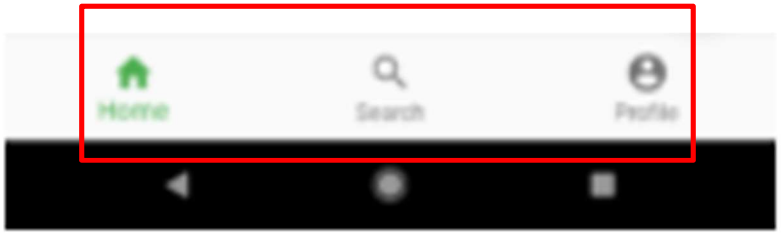
Scaffold widget

AppBar



Drawer

FloatingActionButon



BottomNavigationBar

The Scaffold widget

Property	Value	Remark
appBar	AppBar object	
body	Container object	Can be set to a single widget such as Text too.
drawer	Drawer object	
floatingActionButton	**FloatingActionButtonWidget** object	

The Appbar widget

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

The IconButton widget

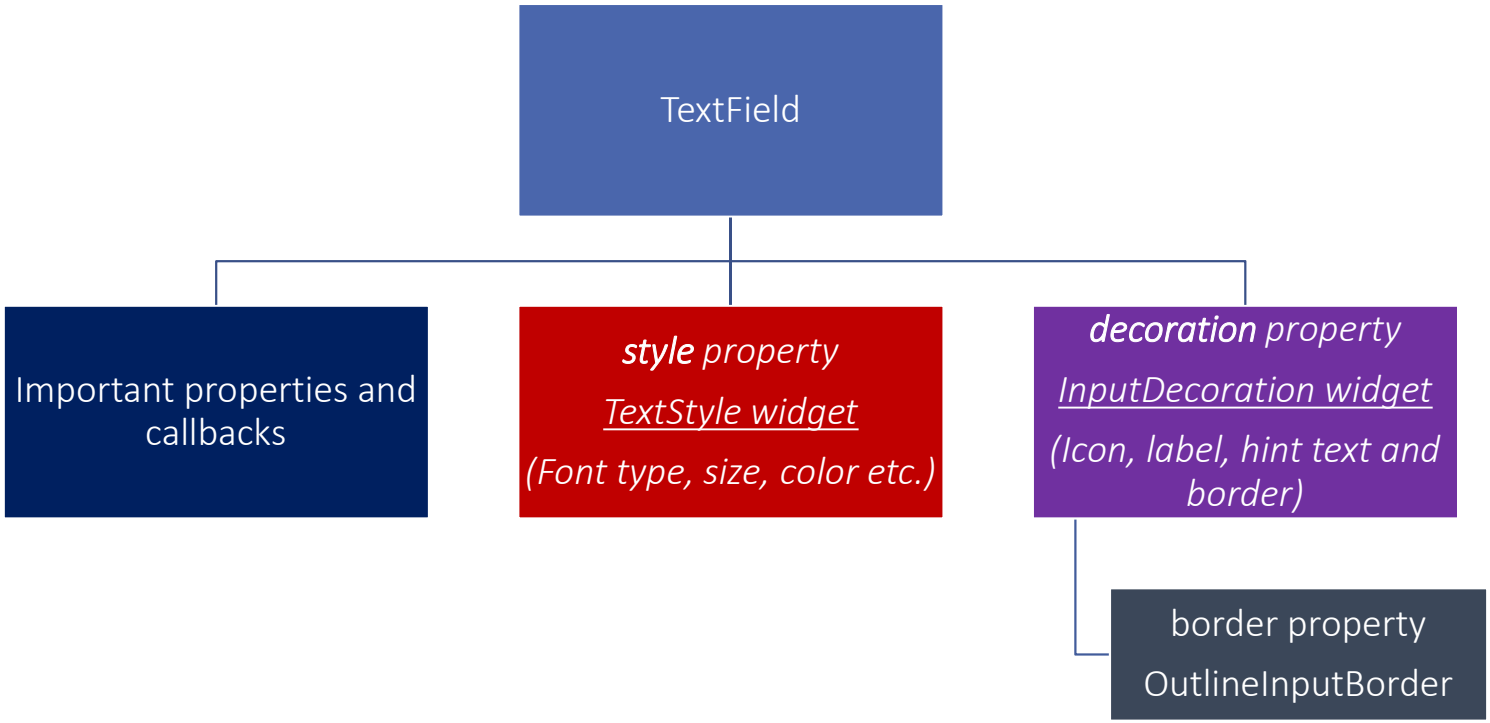
Property	Value	Remark/Usage/Example
icon	Preferably an Icon widget.	icon:const Icon(Icons.exit_to_app)
onPressed	(){ code to execute on tap… }	onPressed: (){ **exit(0)**; } Add the following import statement at the top for exit(0) : **import 'dart:io';**

The FloatingActionButton widget

Property	Value	Remark/Usage/Example
child	You can put Icons or text to represent the purpose of this button.	child: Icon(Icons.add)
onPressed	(){ code to execute on tap… }	onPressed: (){ **exit(0)**; } Add the following import statement at the top for exit(0) : **import 'dart:io';**

To change the location of the floating action button on the screen, use floatingActionButtonLocation property of Scaffold() widget:
Scaffold(floatingActionButtonLocation: FloatingActionButtonLocation.centerFloat)

The TextField widget



Creating a TextField with style and decoration

1. Create a **TextStyle** object.
2. Create an **OutlineInputBorder** object.
3. Create an **InputDecoration** object and set its border property to border created in step-2.
4. Create TextField and set its style and decoration properties to object created in step-1 and step-3 respectively.

Changing font color, size and weight using TextStyle class

Property/Method	Value	Remark/Usage/Example
color	Colors.____	Affects font color
fontSize	A positive integer	Affects font size
fontWeight	FontWeight.bold FontWeight.w100 (Thinnest) to FontWeight.900 (Dark black) w400=normal, w700=bold	Affects font weight
fontStyle	FontStyle.italic	Makes font italic
decoration	TextDecoration.underline = underline TextDecoration.lineThrough = strike-through	Draws underline or strikethrough

Creating TextStyle

```
TextStyle style1 = const TextStyle(
color:Colors.deepOrange,
fontSize:18,
fontWeight: FontWeight.bold,
fontStyle:FontStyle.italic,
decoration:TextDecoration.underline );
```

Creating a border

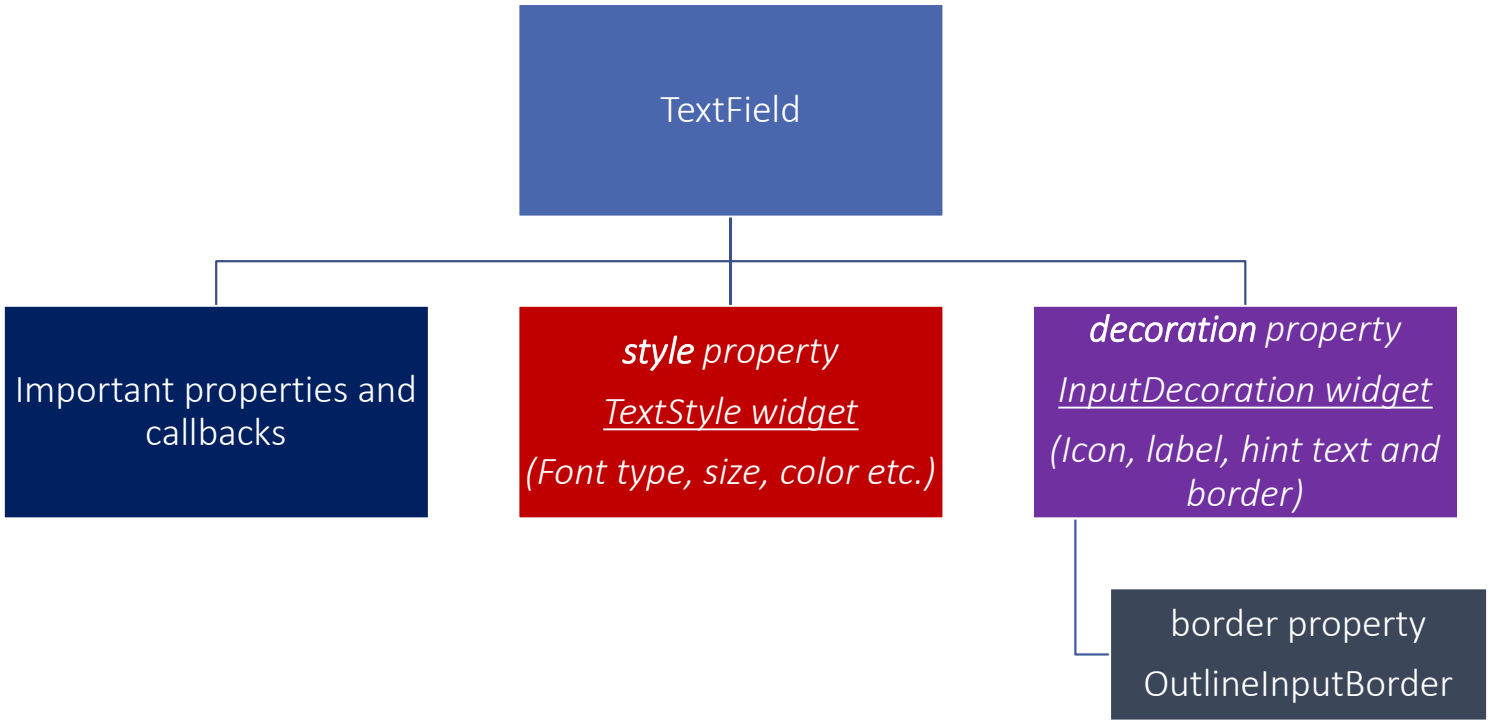
Create a circular border:

```
OutlineInputBorder brd =
OutlineInputBorder(borderRadius:BorderRadius.circular(100));
```

Create a regular rectangular border:

```
OutlineInputBorder brd =  OutlineInputBorder(borderRadius:BorderRadius.zero);
```

The TextField widget



Creating a TextField with style and decoration

1. Create style object.
2. Create a border object.
3. Create an input decoration object and set its border property to border created in step-2.
4. Create TextField and set its style and decoration properties to object created in step-1 and step-3 respectively.

Changing font color, size and weight using TextStyle class

Property/Method	Value	Remark/Usage/Example
color	Colors.____	Affects font color
fontSize	A positive integer	Affects font size
fontWeight	FontWeight.bold FontWeight.w100 (Thinnest) to FontWeight.900 (Dark black) w400=normal, w700=bold	Affects font weight
fontStyle	FontStyle.italic	Makes font italic
decoration	TextDecoration.underline = underline TextDecoration.lineThrough = strike-through	Draws underline or strikethrough

Creating TextStyle

```
TextStyle style1 = const TextStyle(
color:Colors.deepOrange,
fontSize:18,
fontWeight: FontWeight.bold,
fontStyle:FontStyle.italic,
decoration:TextDecoration.underline );
```

Creating a border

Create a circular border:

```
OutlineInputBorder brd =
OutlineInputBorder(borderRadius:BorderRadius.circular(100));
```

Changing icon, label text, hint text and border using InputDecoration class

Property	Value	Remark/Usage/Example
icon	Icon(Icons._____)	Icon which is displayed **before** the TextField
label	Widget	Label for the TextField. Can use a Text or an Icon widget.
labelText	String	Label for the TextField
hintText	String	Hint text for the input. It appears only after the cursor is placed inside the TextField and disappears as soon as the input is typed-in.
border	OutlineInputBorder() - Border around the TextField UnderlineInputBorder() – Default	

You can use any one of label or labelText property. Attempt to use both of them results into runtime error.

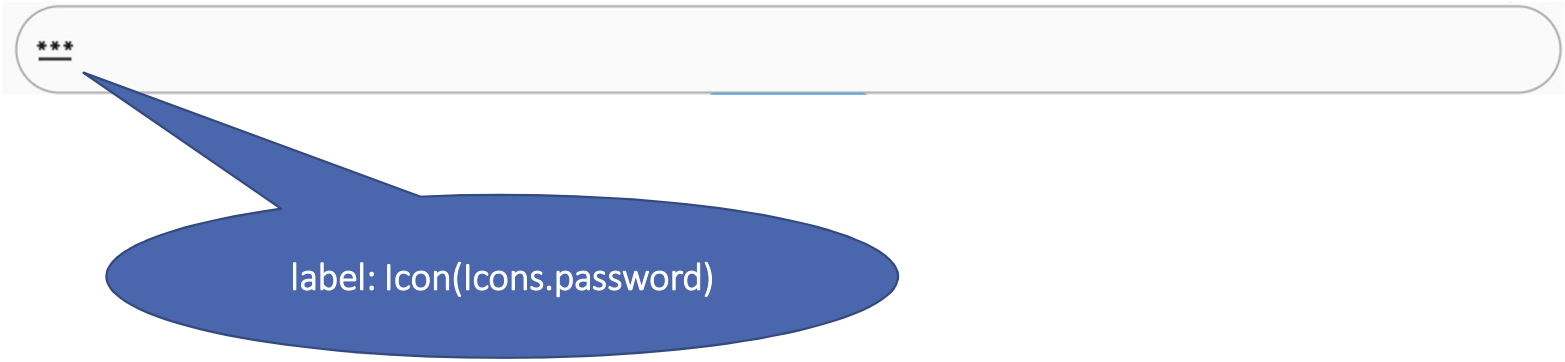
Changing icon, label text, hint text and border using InputDecoration class

InputDecoration with only label:

InputDecoration with **both** icon and label:

Changing icon, label text, hint text and border using InputDecoration class

InputDecoration with label set to an icon:



Creating InputDecoration

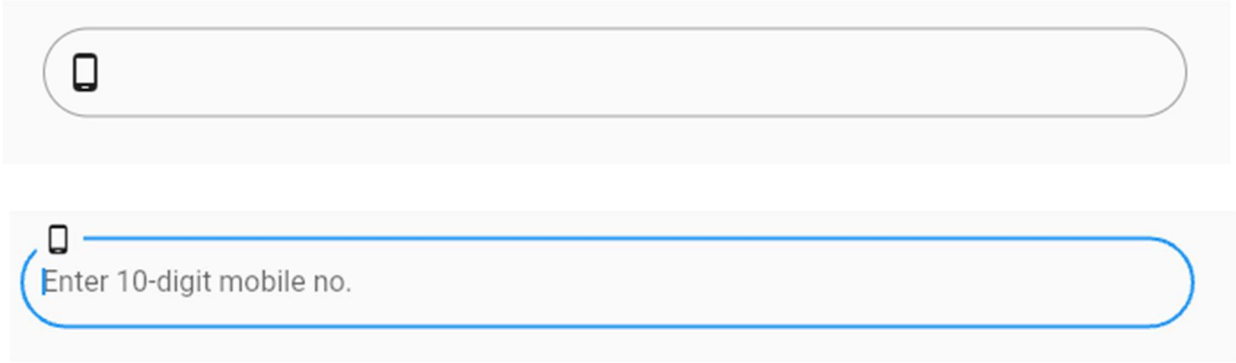
Create InputDecoration object and use border and *String label*:

InputDecoration idc1 = InputDecoration(border:brd,labelText:"Email",hintText:"Enter valid email id");

Creating InputDecoration

Create InputDecoration object and use border and *icon* instead of String label:

InputDecoration idc2 = InputDecoration (border:brd, **icon:const Icon(Icons.phone_android_rounded)**, hintText: “Enter 10-digit mobile no.”);



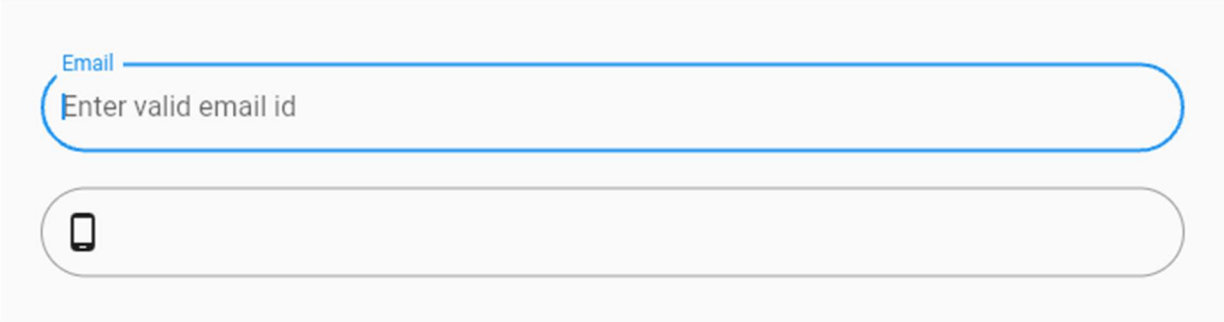
Applying style and decoration to a TextField

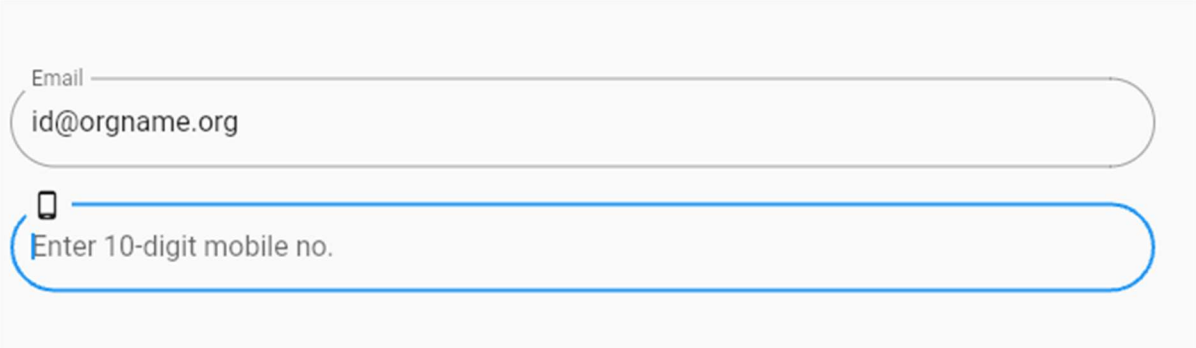
✓ To apply the font-related settings set style property of a TextField.

✓To apply border and label/icon, set **decoration** property of a TextField.

```
TextField t1 = TextField(style:style1,decoration:idc1 );
TextField t2 = TextField(style:style1,decoration:idc2 );
```

TextFields with style and decoration





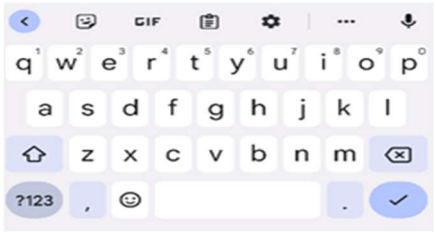
Other important properties of the TextField widget

Property/Method	Value	Remark/Usage/Example
controller	TextEditingController widget	Used to retrieve the text from the TextField.
obscureText	true or false(default)	Masks the text of the TextField, used for password.
maxLength	A positive integer	Maximum no. of characters allowed. It will also **stop** taking input when maxLength characters are entered. It shows a counter at the lower-right corner of the TextField.
maxLines	Default value: 1	Maximum no. of lines if input. maxLength considers total no. of characters in all lines together.

Other important properties of the TextField widget

Property/Method	Value	Remark/Usage/Example
keyboardType	TextInputType.number TextInputType.text TextInputType.datetime	
textAlign	TextAlign.center	causes the cursor and text to align
readOnly	true or false(default)	Makes the textfield read-only.

TextInputType.number



TextInputType.text

Retrieving value from a TextField

Using onChanged() callback:

```
String username="";
TextField t1 = TextField( onChanged:(value ){ username=value;} …
```

Using a TextEditingContoller:

```
TextEditingController tec = TextEditingController();
TextField t2 = TextField(controller:tec);
String password = tec.text;
```

Widget Categories

- The flutter widgets are grouped into 14 categories based on their functionality in a flutter app.

1. Accessibility: This group of widgets makes a flutter app more user-friendly.

2. Animation and motion: Widgets that bring animation and motion to other widgets are known as animation and motion widgets.

3. Assets, Images, and Icons: These widgets manage assets like displaying images and icons.

4. Async: These are used in the flutter application to give async capabilities.

5. Basics: This is a collection of widgets that are required for the building of any flutter application.

6. Cupertino: These widgets are created for iOS.

Widget Categories

7. Input: In a flutter application, this collection of widgets provides input capability.

8. Interaction Models: Such widgets manage touch events and direct users to various views inside the app.

9. Layout: This collection of flutter widgets aids in placing other widgets on the screen.

10. Material Components: This is a collection of flutter widgets based on Google's Material Design.

11. Painting and effects: These are a group of flutter widgets that change the appearance of their children without altering their design or shape.

12. Scrolling: This adds scroll ability to several other flutter widgets that aren't ordinarily scrollable.

13. Styling refers to the app's theme, responsiveness, and sizing.

14. Text: These are used to display text.

Structure of a Flutter app

import 'package:flutter/material.dart';

void main() => runApp(const object of MaterialApp);

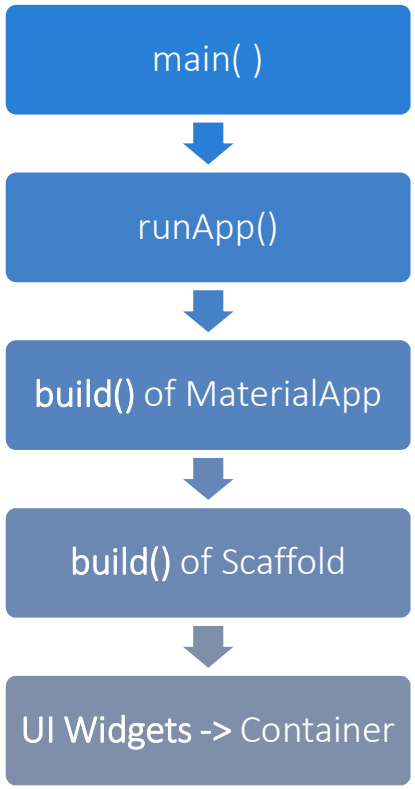
Definition of MaterialApp class

-Override build() method and return a MaterialApp object

Definition of Scaffold class

-Override build() method and return a Scaffold object

Hierarchy of widgets



More about main() and runApp()

- main() is defined as: void main() => runApp(const MyApp());
- Arrow syntax i.e => is used to while defining main().
- The void runApp(Widget widget) takes a widget as an argument and sets it on a screen. i.e. It attached the given widget to the screen.
- It makes the given widget the root widget of the app and other widgets as the child of it.

MaterialApp widget

- A MaterialApp widget acts as the root of the widget tree of a Flutter application.
- runapp() method takes an object of type MaterialApp as an argument and covers the screen with the MaterialApp object.
- MaterialApp is implemented as a subclass of StatelessWidget class.
- build() method of the StatelessWidget class returns a MaterialApp object.
- Syntax:

MaterialApp m = MaterialApp(title:t , home:h)

where t is an object of **String** class and h is an object of **Scaffold** class.

Properties of MaterialApp widget

- color: It controls the primary color used in the application.
- darkTheme: It provides theme data for the dark theme for the application.
- debugShowCheckedModeBanner: This property takes a boolean as the object to decide whether to show the debug banner or not.
- home: This property takes a widget as the object to show on the default route of the app.
- theme: This property takes a ThemeData class as the object to describe the theme for the MaterialApp.
- title: The title property takes a string as the object to decide the one-line description of the app for the device. When the user press the recent apps button on mobile the text proceeded in title is displayed.

Scaffold widget

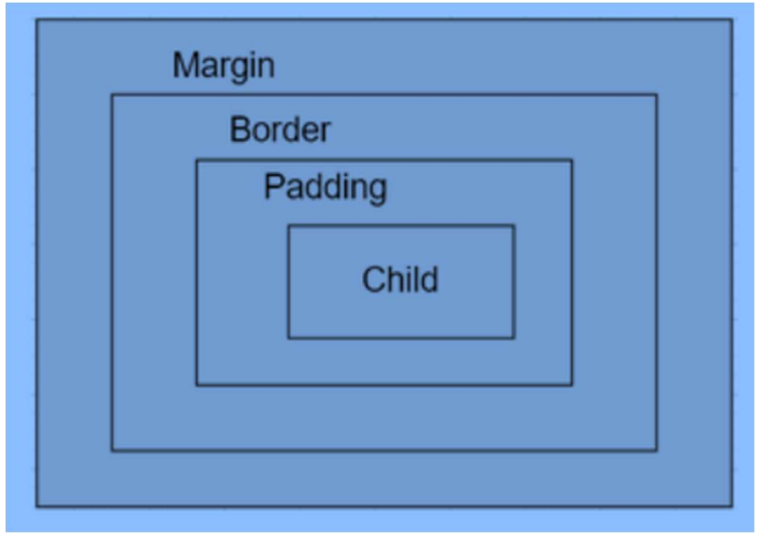
- A Scaffold implements basic material design visual layout.
- This widget provides APIs for showing drawers, appbars and the body of the app.
- The **body** property of the Scaffold will be used here to display the contents i.e. GUI elements of the app.
- Syntax:

Scaffold s = Scaffold(**appBar:a , body:b**);

where a is an object of **AppBar** class and b is an object of **Container** class.

Container widget

- A Container class can be used to store one or more widgets and position them on the screen according to our convenience. Basically, a container is like a box to store contents.



Properties of Container widget

- child: Container widget has a property ‘child:’ which stores its children. The child class can be any widget.
- color: The color property sets the background color of the entire container. Now we can visualize the position of the container using a background color.
- height and width: By default, a container class takes the space that is required by the child. We can also specify the height and width of the container based on our requirements. (height: n, width: n OR double.infinity Note: n is no. of pixels)
- alignment: The alignment is used to position the child within the container. We can align in different ways: bottom, bottom center, left, right, etc. (alignment: Alignment.___)

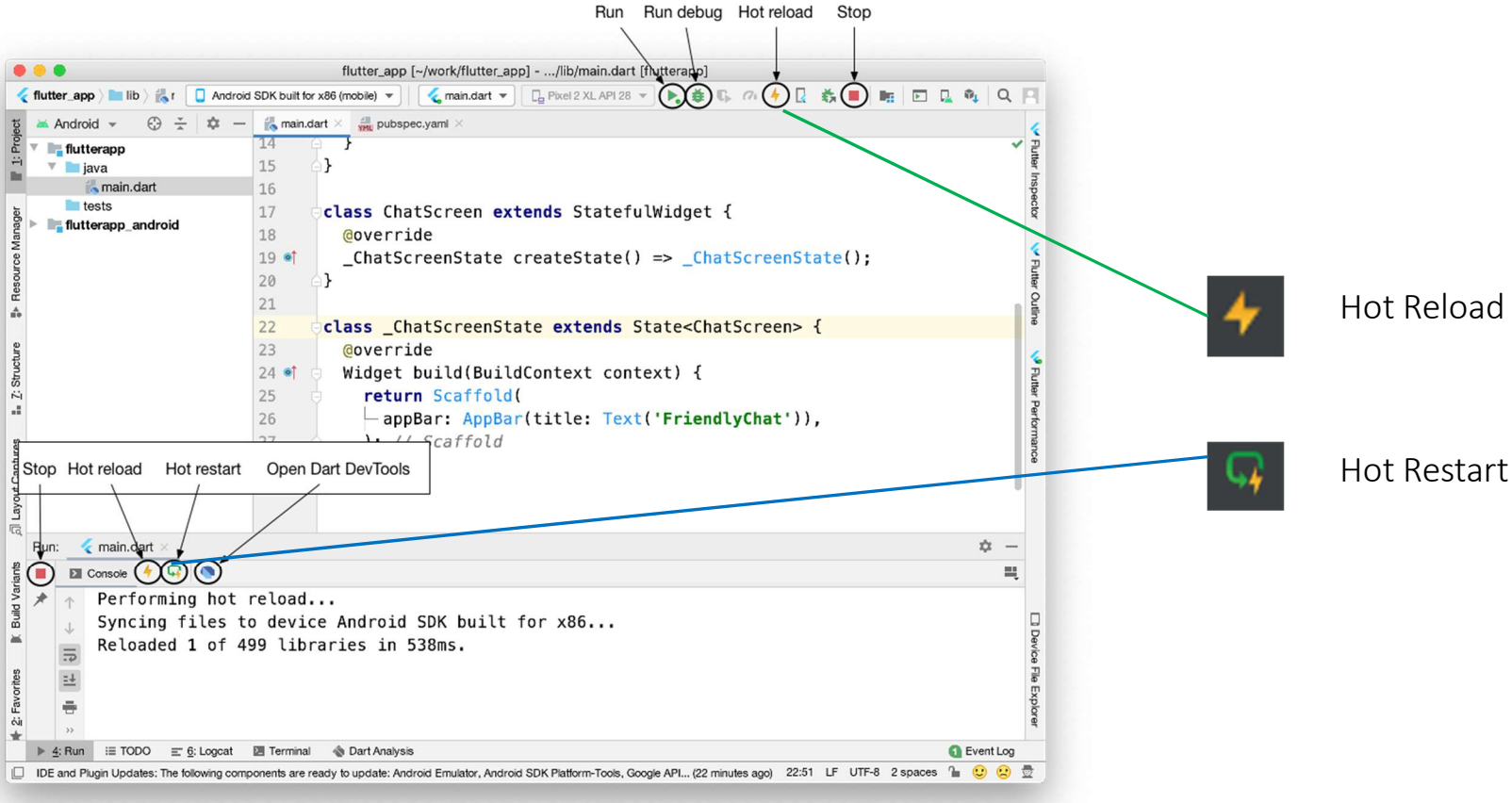
Features of Flutter

- Open-source and Free
- Cross-platform (Single Codebase)
- Fast and simple development (Hot Reload)
- Widget library
- Native Performance

Hot Reload

- Hot reload works by injecting updated source code files into the running Dart Virtual Machine (VM).
- After the VM updates classes with the new versions of fields and functions, the Flutter framework automatically rebuilds the widget tree
- These changes are reflected in the UI.
- What is the difference between hot reload, hot restart, and full restart?
  - *Hot reload loads code changes into the VM and re-builds the widget tree, preserving the app state; it doesn't rerun main() or initState().*
  - *Hot restart loads code changes into the VM, and restarts the Flutter app, losing the app state.*
  - *Full restart restarts the iOS, Android, or web app. This takes longer because it also recompiles the Java / Kotlin / ObjC / Swift code.*

Hot reload and Hot restart



Pre-made Flutter Widgets

SafeArea The SafeArea widget is necessary for today's devices such as the iPhone X or Android devices with a notch (a partial cut-out obscuring the screen usually located on the top portion of the device).

The SafeArea widget automatically adds sufficient padding to the child widget to avoid intrusions by the operating system. You can optionally pass a minimum amount of padding or a Boolean value to not enforce padding on the top, bottom, left, or right.

Pre-made Flutter Widgets

Text

The Text constructor takes the arguments string, style, maxLines, overflow, textAlign, and others.

TextAlign: It is used to specify how our text is aligned horizontally. It also controls the text location.

Overflow: It is used to determine when the text will not fit in the available space. It means we have specified more text than the available space.

MaxLines: It is used to determine the maximum number of lines displayed in the text widget.

Style: It can do styling by specifying the foreground and background color, font size, font weight, letter and word spacing, locale, shadows, etc.

Pre-made Flutter Widgets

```
style: TextStyle(
fontSize: 24.0,
color: Colors.deepPurple,
decoration: TextDecoration.underline,
decorationColor: Colors.deepPurpleAccent,
decorationStyle: TextDecorationStyle.dotted,
fontStyle: FontStyle.italic,
fontWeight: FontWeight.bold,
),
```

Layout Widgets

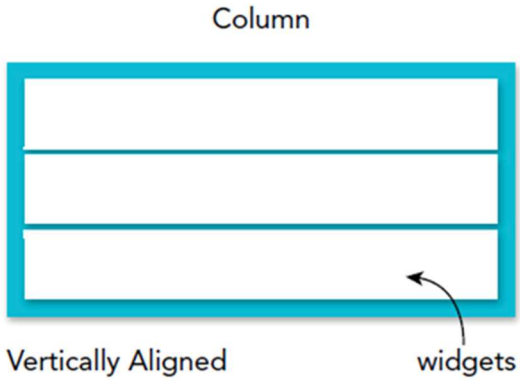
Column widget

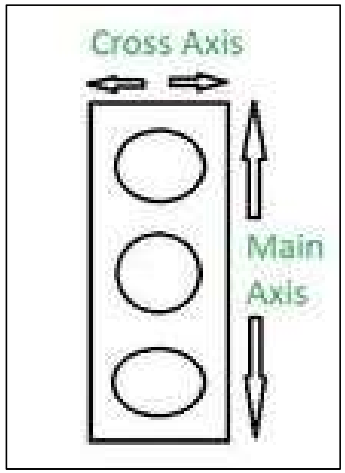
A Column widget displays its children vertically.

It takes a **children** property containing an array of List<Widget>, meaning you can add multiple widgets.

The children align vertically without taking up the full height of the screen.

CrossAxisAlignment, MainAxisAlignment, and MainAxisSize can be used to align and size how much space is occupied on the main axis.

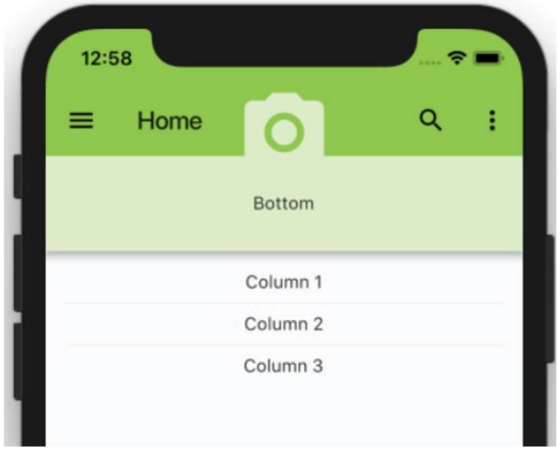




Layout Widgets

Column widget

```
Column(
children: <Widget>[
Text('Column 1'),
Divider(),
Text('Column 2'),
Divider(),
Text('Column 3'),],),
```



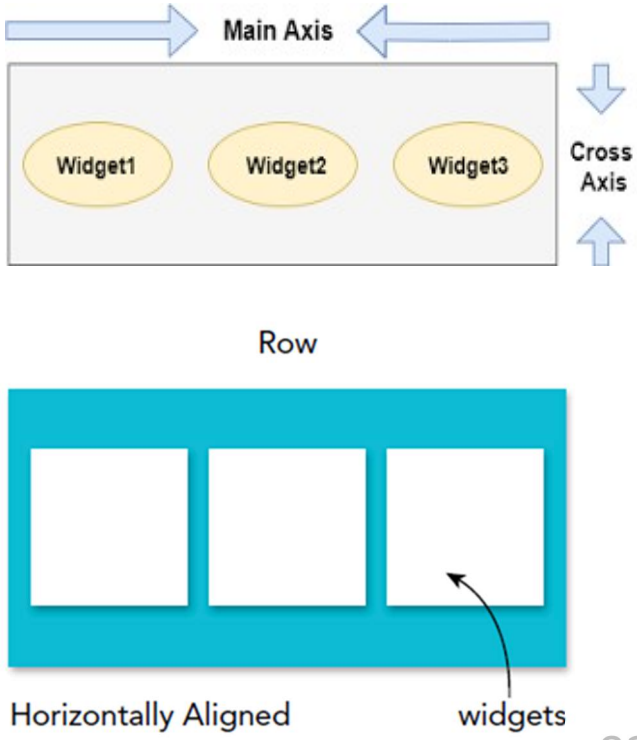
Layout Widgets

Row widget

A Row widget displays its children horizontally.

It takes a **children** property containing an array of List<Widget>.

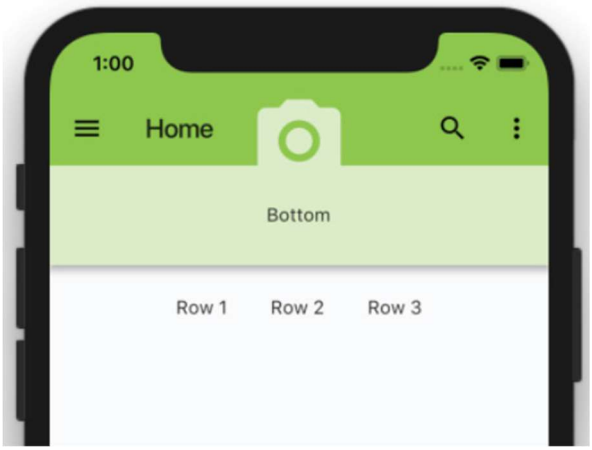
The same properties that the Column contains are applied to the Row widget with the exception that the alignment is horizontal, not vertical.



Layout Widgets

Row widget

```
Row(
children: <Widget>[
Text('Row 1'),
Padding(padding: EdgeInsets.all(16.0),),
Text('Row 2'),
Padding(padding: EdgeInsets.all(16.0),),
Text('Row 3'),],),],),
```



Aligning children widgets

MainAxisAlignment enum

Values:

start → Places the children as close to the start of the axis as possible.

end → Places the children as close to the end of the axis as possible.

center → Places the children as close to the middle of the axis as possible.

spaceBetween → Places the free space evenly between the children.

spaceAround → Places the free space evenly between the children as well as half of that space before and after the first and last child.

spaceEvenly → Places the free space evenly between the children as well as before and after the first and last child.

Aligning children widgets

CrossAxisAlignment enum

Values:

start → Places the children as close to the start of the axis as possible.

end → Places the children as close to the end of the axis as possible.

center → Places the children as close to the middle of the axis as possible.

stretch → Requires the children to fill the cross axis.

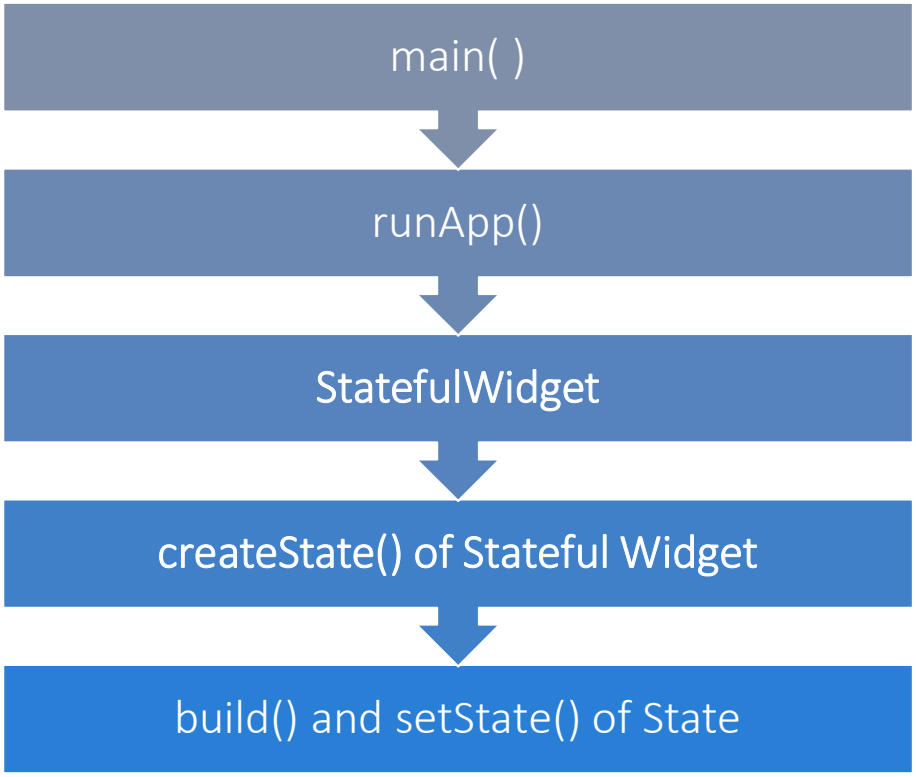
baseline → Places the children along the cross axis such that their baselines match.

Stateless vs. Stateful Widgets

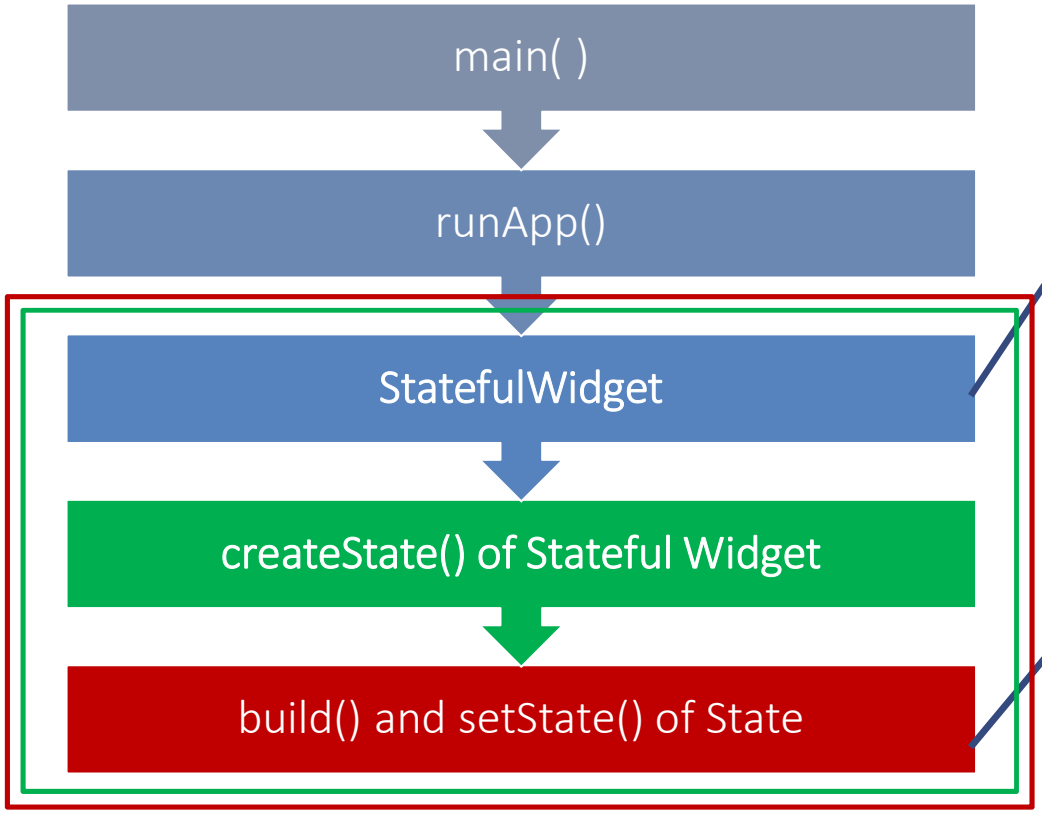
State defines the current properties and their values of the Widget.

Stateless Widgets	Stateful Widget
• A widget in which the properties can not be changed at runtime is known as a Stateless widget. • Stateless widgets are used when there is no need to change the UI dynamically. • The widget class must be a subclass of **StatelessWidget** class and its **build**() method is overridden. • Examples: Text, Icon, IconButton	• A widget in which the properties can be changed at runtime is known as a Stateful widget. • Stateful widgets are used when there is a need to change the UI dynamically. • The widget class must be a subclass of **Statefulwidget** class and its **createState**() method is overridden. • Examples: TextField, RadioButton, CheckBox

Stateful Widgets



Creating and using Stateful widgets



1. Create a subclass of StatefulWidget class.
2. Override createState() method in it
3. Create and return an object of State class from createState() method.

1. Create a subclass of State class
2. Override build() method.
3. Use setState() method to update the UI.

Stateful Widgets

Steps to implement a Stateful Widget:

1. Create a class that extends **StatefulWidget** and override **createState()** method. Create an object of **State** class and return it from createState() method.
2. Create a class that extends State class and override **build()** method.
3. Call setState() method in response to some event to update the UI. Calling setState() will call build() method again.

Stateful Widgets

1. Create a class that extends StatefulWidget and override createState() method.

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

  @override
  UI createState()
  {
        return UI();
  }

}
```

1

2

3

OR

Short-hand notation…

```
UI createState() => UI();
```

UI is a sub-class of State class and it will be defined next.

Stateful Widgets

2. Create a class that extends State class and override build() method.

```
class UI extends State<Demo>
{
    Widget build(BuildContext context)
    {
        setState(()
        {
        debugPrint("setState() called...");

        });

      return MaterialApp(....);
    }
}
```

1

2

3

Stateful Widgets

3. Call setState() method in response to some event to update the UI. Calling setState() will call build() method again.

```
class UI extends State<Demo>
{
        Widget build(BuildContext context)
        {
                onChanged:(bool? value) {
                setState(()
                {

                });
         }
}
```

The setState() method

- A method of State class.
- Notifies the framework that the internal state of this object has changed.
- Syntax: void setState(VoidCallback vfn)
- VoidCallback means a callback which does not take any arguments and does not return anything.
- Calling setState()

```
setState( () { .... } );
```

- It causes the framework to call the build() again for this State object.
- If you just change the state directly without calling setState(), the framework might not schedule a build and the user interface for this subtree might not be updated to reflect the new state.

CheckboxListTile Widget

- In Flutter, there are two types of checkboxes:
  1. Checkbox
  2. CheckboxListTile
- Checkbox does not have a label whereas CheckboxListTile has a title(i.e. a label) and a subtitle.
- Thus, an additional Text widget is required for using it as a label for a Checkbox.
- Therefore, CheckboxListTile is used instead of Checkbox

CheckboxListTile

Important properties of a CheckboxListTile

Attribute	Description
value	It is used to identify whether the checkbox is checked or not. If the checkbox is checked, value is equal to true and vice verse. If the checkbox not checked, value is equal to false and vice verse.
onChanged	It will be called when the value property changes. i.e. whenever the checkbox the is clicked or changed in any other way. setState() method is invoked inside onChanged.
title	It specifies the main title of the list.
subtitle	It specifies the subtitle of the list. It appears below the title. It is used to add the description.
secondary	It is the widget, which is displayed in front of the checkbox. Mostly an icon is used.
selected	This property takes in a boolean value as the object to decide whether the checkbox will be already selected or not.

CheckboxListTile

Handling selection/deselection

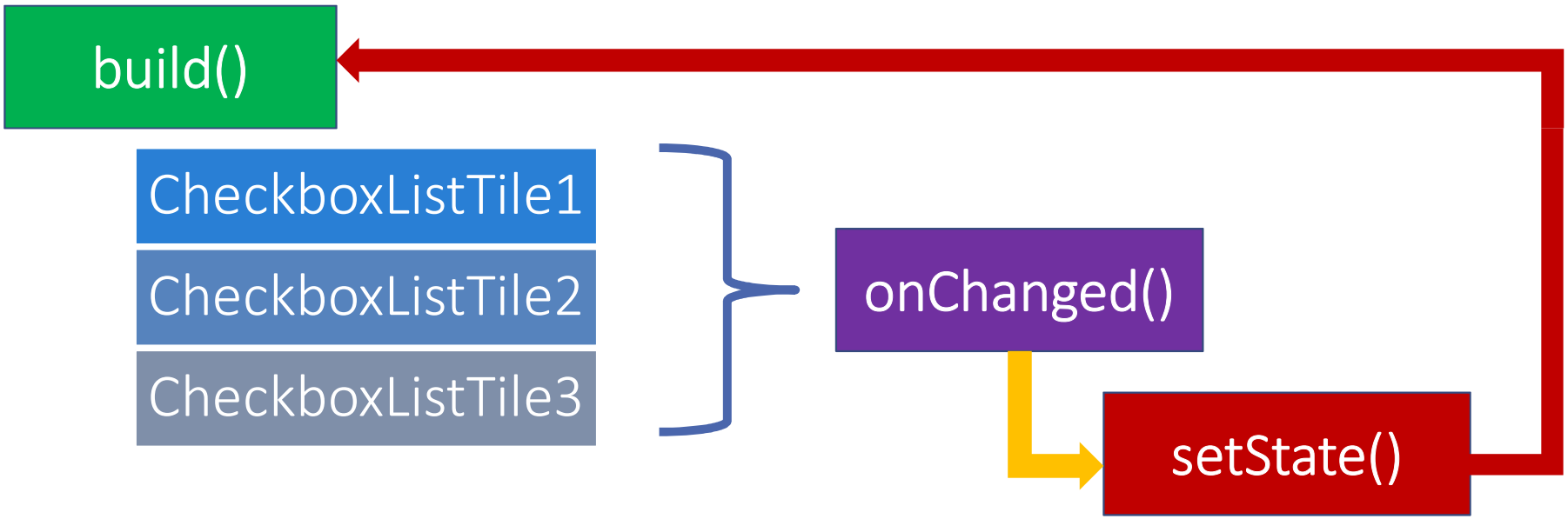
1. Declare a bool variable outside the build() and initialize it to false.
2. Assign this variable declared outside build() to the value property of CheckboxListTile.
3. Call setState() method inside code of onChanged() of CheckboxListTile.
4. Inside setstate() method, check the value property.
   - *If value=true, the CheckboxListTile is currently selected.*
   - *If value=false, the CheckboxListTile is currently NOT selected.*

CheckboxListTile

Important note:

1. The onChanged() method is called whenever a CheckboxListTile is selected or deselected.
2. The onChnaged() method calls the setState() method.
3. The setState() method calls the build() method.
4. Each CheckboxListTile has its onChanged() method but it is called only of that CheckboxListTile which is selected/deselected.

CheckboxListTile

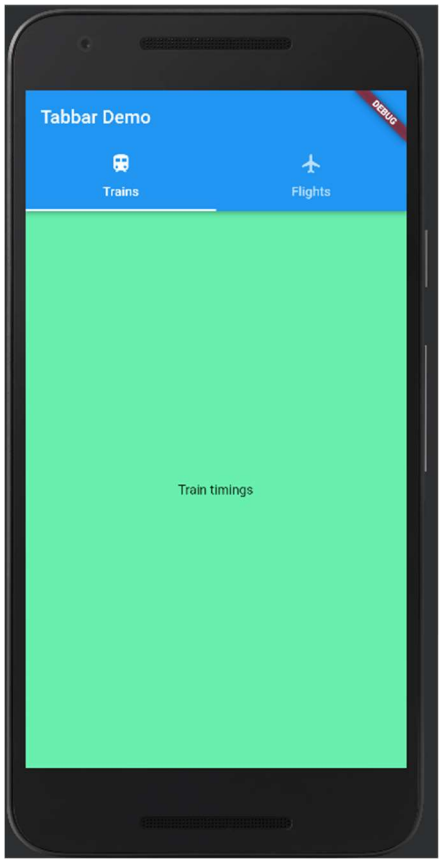


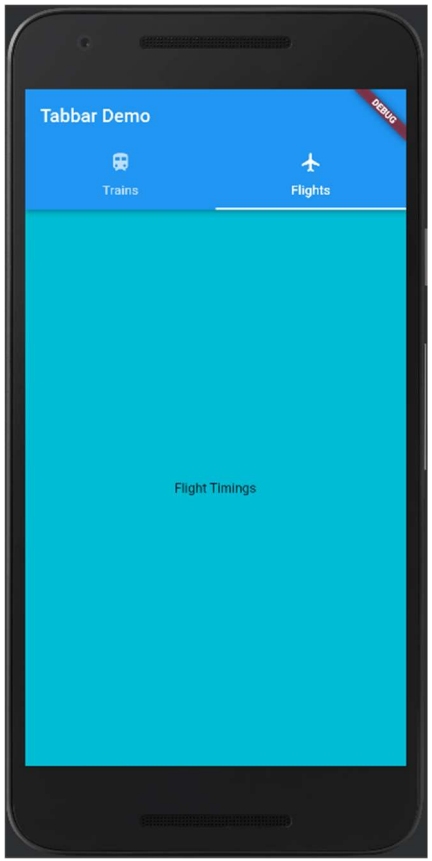
RadioListTile

Important properties of a RadioListTile

Attribute	Description
value	The value which is returned when the radio button is selected. **Each radio must have a different value.**
groupValue	The variable in which the **currently selected value** will be stored. **All the radio buttons must have the same variable name as its groupValue.**
onChanged	It will be called when the radio button is selected.
title	It specifies the main title of the list.
subtitle	It specifies the subtitle of the list. It appears below the title. It is used to add the description.
secondary	It is the widget, which is displayed in front of the radio button. Mostly an icon is used.
selected	This property takes in a boolean value as the object to decide whether the radio button will be already selected or not.

Using Tabs within a screen





Using Tabs within a screen

- Create one file for each tab contents.
- Create main. dart.
- Import all the files created for tab contents in main.

Using Tabs within a screen

```
//tab1.dart
import 'package:flutter/material.dart';
class Tab1 extends StatelessWidget {
  @override
  Widget build(BuildContext context)
  {
    return Container(color:Colors.greenAccent,
        child: Center(
            child: Text("Train timings")));
  }
}
```

Using Tabs within a screen

```
//tab2.dart
import 'package:flutter/material.dart';
class Tab2 extends StatelessWidget {
  @override
  Widget build(BuildContext context)
  {
    return Container(color:Colors.cyan,
        child: Center(
            child: Text("Flight Timings")));
  }
}
```

Using Tabs within a screen

//main.dart

- Create Tab objects.
- Create a TabBar object.
- Create AppBar with its ***bottom*** property set to TabBar object.
- Create TabBarView object and set its ***children*** property to objects of Tab content classes created in other dart files.
- Create a Scaffold object and set its ***body*** property to TabBarView object.
- Create a DefaultTabController object and set its ***child*** property to Scaffold object.
- Create a MaterialApp object and set its ***home*** property to DefaultTabController object.

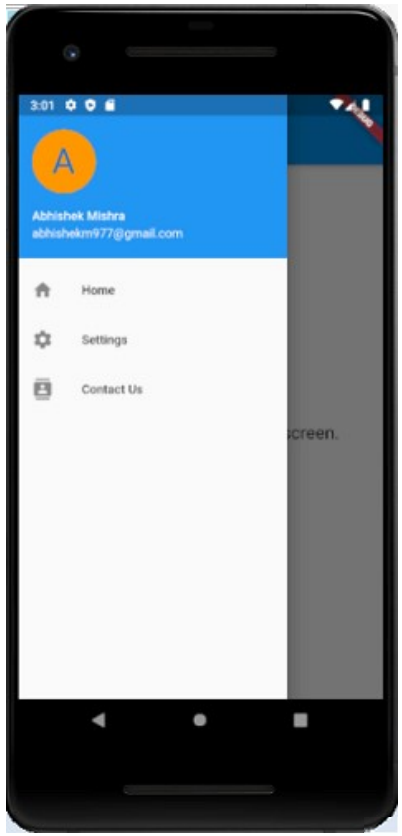
Code of build() method in main.dart

```
Tab t1 = const Tab(icon: Icon(Icons.train), text: 'Trains');
Tab t2 = const Tab(icon: Icon(Icons.flight), text: 'Flights');
TabBar tb = TabBar(tabs: [t1, t2]);
AppBar ab = AppBar(title: const Text('Tabbar Demo'), bottom: tb);
TabBarView tbv = TabBarView(children: [Tab1(), Tab2()]);
Scaffold sf = Scaffold(appBar:ab,body: tbv);
DefaultTabController dtc = DefaultTabController(length: 2, child: sf);

MaterialApp m = MaterialApp(home: dtc);
return m;
```

Drawer widget

- A drawer is an invisible side screen.
- It is a sliding left menu that generally contains important links in the application and occupies half of the screen when displayed.
- A drawer is used inside a Scaffold object.
- The most important property of a drawer is **child**. This property is usually initialized to a **ListView** object.



Drawer widget

It is a property of Scaffold widget.

However, when you add a drawer to Scaffold, the menu icon will appear on **AppBar**.

Constructor: **Drawer(double elevation, Widget? child)**

The elevation property is used to raise the Drawer panel with shadow, you need to pass the double value which determines the height of elevation.

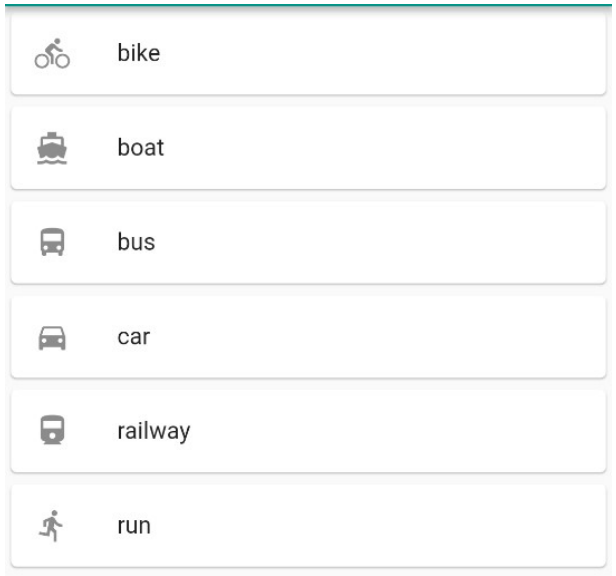
The child property is used to pass the contents widget of Drawer.

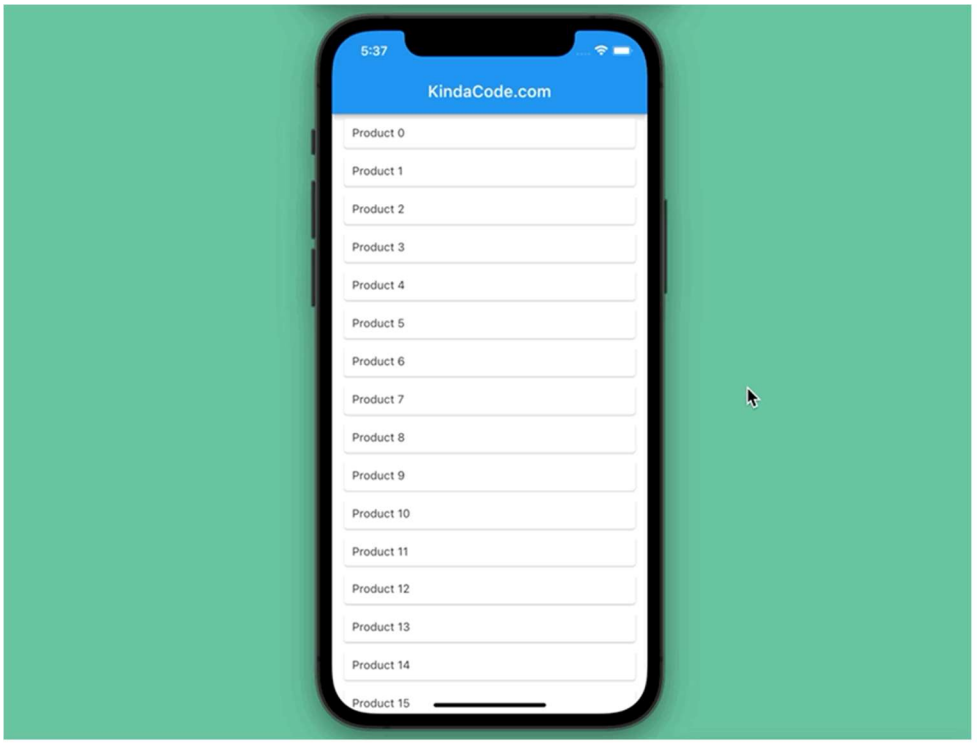
ListView widget

- ListView is used to create a list of fixed or dynamic length.
- ListView of fixed length is created with the help of the ListView **constructor**. The *children* property is used to add items in such a list.
- The item can be any reasonable widget such as ListTile, button, checkbox etc.
- ListView of dynamic length i.e. a dynamic list is created with the help of the **builder()** method.
- The main difference between ListView and ListView.builder is:

ListView creates all items **at once**, whereas the ListView.builder() creates items **when the list is scrolled and the item becomes visible on the screen.**

ListView widget





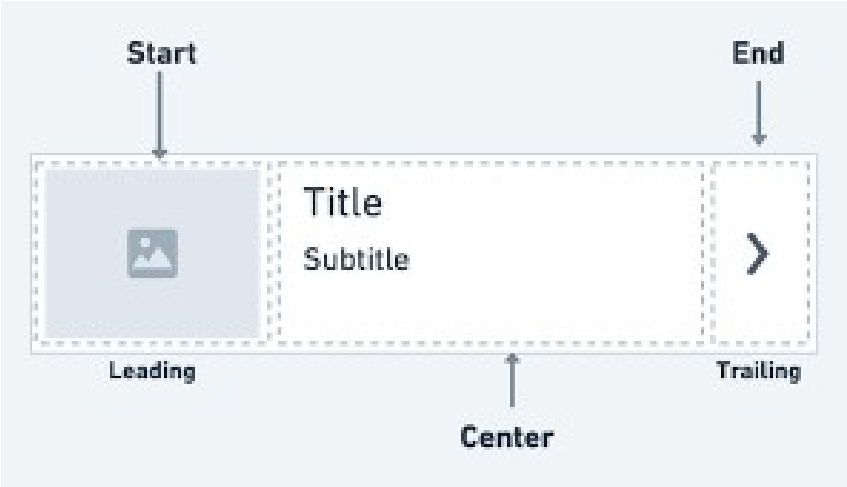
ListTile widget

A ListView is normally populated with ListTile widgets.

A ListTile contains one to three lines of text optionally along with icons or other widgets, such as check boxes.

The icons (or other widgets) for the tile are defined with the **leading** and **trailing** parameters.

The first line of text is not optional and is specified with **title**. The value of **subtitle**, which is optional, appears under the title.



Listview.builder()

ListView lv = ListView.builder(**itemCount**:_____,**itemBuilder**:_____)

itemCount takes an integer value which is the total no. of items in the list.

The builder() method has a required argument-**itemBuilder**.

This argument is set to a **function** which create and **returns a widget**.

itemBuider: (context,index)

```
{
   ListTile item = ListTile(title:Text("Item $index"));
    return item;
}
```

The itemBuilder function is called repeatedly to create and add more widgets to the list.

Adding a scrollbar to Listview

- By default, a ListView does not have any scrollbar.
- To add a scrollbar to a ListView:
  - Create a **ScrollController** object first.
  - Create a **ScollBar** object, set its ***controller*** property to the **ScrollController** object and its ***child*** property to a ListView object.
  - Set Listview's ***controller*** property to the **ScrollController** object.

Drawer-ListView-ListTile

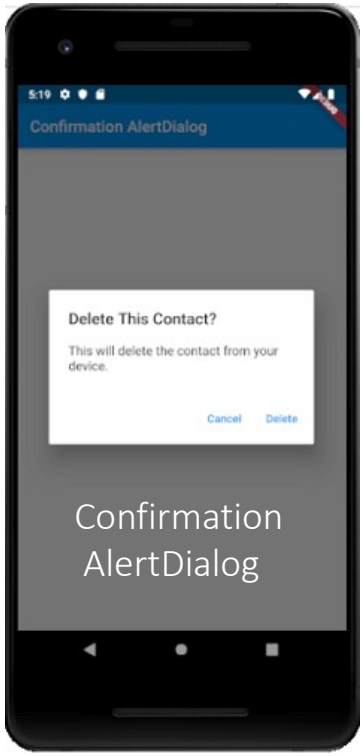
Flutter Alert Dialogs

We can categories the alert dialog into multiple types, which are given below:

1. Basic AlertDialog
2. Confirmation AlertDialog
3. Select AlertDialog
4. TextField AlertDialog

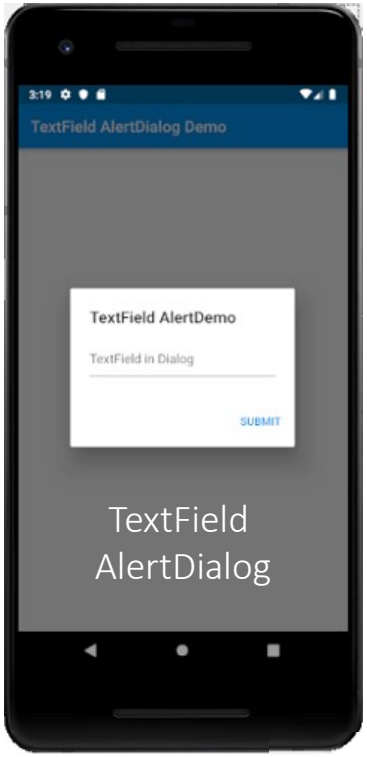
Flutter Alert Dialogs

Simple AlertDialog



Confirmation AlertDialog

Select AlertDialog



TextField AlertDialog

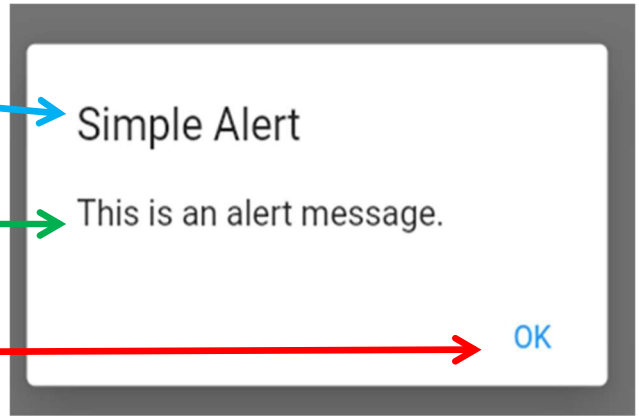
Flutter Alert Dialogs

Important properties of a AlertDialog

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

content: Detailed message OR Widget such as TextField

actions: It is mostly one or more buttons.



Flutter Alert Dialogs

Step-by-step Procedure

Step-1 Create buttons for actions attribute of AlertDialog.

Step-2 Create AlertDialog object. Use buttons created in Step-1 as the value for *actions* attribute.

Step-3 Define showDialog() method and return the AlertDialog object created in Step-2 from this method.

Step-4 Call showAlertDialog(context) and add code created in Step-1 to 3 inside this method.

Flutter Alert Dialogs

Step-1 Create button(s) for the dialog box

```
// Create button(s)
  Widget okButton = ElevatedButton(
    child: Text("OK"),
    onPressed: () {
      Navigator.of(context).pop();
    },
  );
```

Flutter Alert Dialogs

Step-2 Create AlertDialog object

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

Flutter Alert Dialogs

Step-3 Define the showDialog() method

```
// show the dialog
 showDialog(
   context: context,
   barrierDismissible=false; // User must respond to the dialog.
   builder: (BuildContext context) {
     return alert;
   },
 );
```

Flutter Alert Dialogs

Step-4 Define the showAlertDialog() method

```
showAlertDialog(BuildContext context)
{
// Code to create button(s)
// Code to create AlertDialog
// Call to the showDialog() method
}
```

Call the showAlertDialog() method in onPressed callback of a button.

AlertDialog with a TextField

Step-1 Create TextField and button(s) for the dialog box

```
// Create a TextField and button(s)
TextEditingController _textFieldController = TextEditingController();
TextField tf = TextField(controller: _textFieldController,decoration: InputDecoration(hintText: "TextField in Dialog"),);

 Widget submitButton = ElevatedButton(
  child: Text(“Submit"),
  onPressed: () {
   // Code to process input
    Navigator.of(context).pop();
  },
);
```

AlertDialog with a TextField

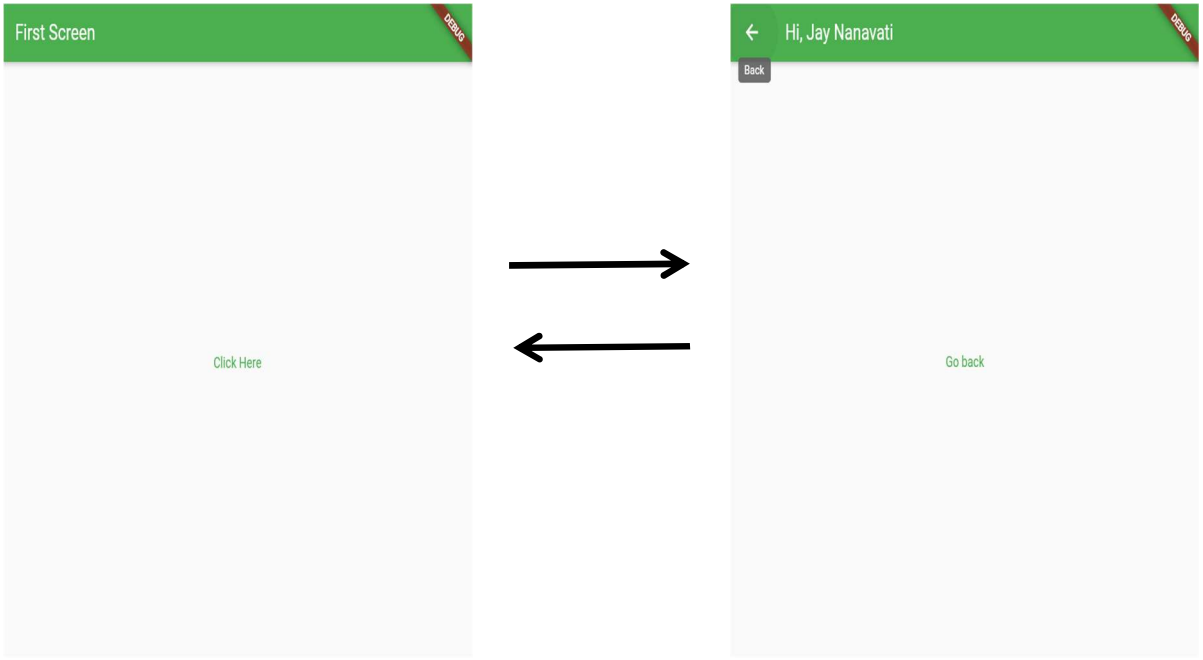
Step-2 Create AlertDialog object

```
// Create AlertDialog
 AlertDialog alert = AlertDialog(
   title: const Text(“TextField Alert"),
   content: tf,
   actions: [ submitButton ]
 );
```

Flutter Navigation and Routing

- In Flutter, each screen is known as route, and the process of moving from one screen to other is known as Navigation.
- Flutter provides a basic routing class **MaterialPageRoute** and two methods **Navigator.push() and Navigator.pop()** that shows how to navigate between two routes.

Flutter Navigation and Routing



Flutter Navigation and Routing

Step-by-step Procedure

Step-1 Create routes i.e. one class per screen.

Step-2 Navigate to one route from another route i.e. from the given screen to the next by using the Navigator.push() method.

Step-3 Navigate back to the previous route by using the Navigator.pop() method.

Flutter Navigation and Routing

Step-1 Create routes i.e. one class per screen.

class Screen1 extends StatelessWidget { ……. }

class Screen2 extends StatelessWidget { ……. }

Flutter Navigation and Routing

Step-2 Navigate to one route from another route i.e. from the given screen to the next by using the Navigator.push() method.

```
TextButton(child: const Text('Go to Screen-2'),
onPressed: () {
Navigator.push(
context,
MaterialPageRoute(builder:(context)=> Screen2()));
}
);
```

Flutter Navigation and Routing

Step-3 Navigate back to the previous route by using the Navigator.pop() method.

```
TextButton(child: const Go to Screen-1’),
onPressed: () {
Navigator.pop( context);
}
);
```

Navigation with Named Routes

- In a multi-screen app, if there is a need to redirect to a particular screen and not just move to previous or next screen, push() and pop() methods will not be useful.
- Use of named route is required to directly move to a particular screen.
- Named routes are defined in the top-level MaterialApp. The ***routes*** is assigned pairs of names and corresponding routes i.e. path and targets.
- **pushNamed()** of Navigator is used to redirect to a named route.

```
onPressed: (){ Navigator.pushNamed(context,route-name); }
```

Navigation with Named Routes

```
@override
Widget build(BuildContext context)
{
  MaterialApp m =  MaterialApp(home:const Screen1(),
      routes:{
    "/s1": (context)=>const Screen1(),
    "/s2": (context)=>const Screen2(),
    "/s3": (context)=>const Screen3(),
    "/s4": (context)=>const Screen4(),

  });  // MaterialApp
  return m;
}
```

Passing data with Named Routes

On Sender screen:

Use *arguments:* _____ while calling the pushNamed()

```
onPressed: (){ Navigator.pushNamed(context,route-name, arguments: data); }
```

On Receiver screen:

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

OR

```
Type data_received = ModalRoute.of(context)!.settings.arguments as Type;
```

Form widget

- Quite similar to an HTML form.
- A group of many input elements.
- Is used to process multiple input elopements together.
- Can validate, reset and save form data.
- GlobalKey of form:

When we want to perform any action, we need to refer to our form using some unique identifier. All of this is done by using Flutter's "GlobalKey" class.

GlobalKey lets us generate a unique, app-wide ID that we can use with our form.

final **formKey** = GlobalKey<**FormState**>();

Later, Form form = Form(**key:formKey**,child: cl);

Creating and using a Form widget

Steps:

In State class

1. Create global key for the form.

2. Create all the children widgets and add them to a collection such Row, Column or Container.

(Use TextFormField instead of TextField.)

3. Create a Form widget and set its **child** property to collection of children.

4. Define validator function for TextFormFields.

5. Call validate() and reset() in the State class.

TextFormField widget

Quite similar to TextField with only one important difference:

TextFormField has a validator() function, whereas TextField does not have it.

```
TextFormField t1 = TextFormField(
    validator:(value) {
     if (value!.isEmpty) {
       return 'Name can not be empty!';
     }
     return null;
    },
```

The validator() returns an error string to display if the input is invalid, or *null* otherwise.

Form validation and reset

- To validate all the fields i.e. to call validator function of each control,

```
formKey.currentState!.validate();
```

It returns true if the input data passes all the validation.

- To reset all the fields i.e. to empty each control,

```
formKey.currentState!.reset();
```

Slider

- An input control
- A value can be set from a range of values by dragging the *thumb* or clicking on the desired position on the *track*.
- The track has an *active* as well as an *inactive* portion.
- A slider can be used for selecting a value from a continuous or discrete set of values.
- Practical applications:

✓ Numeric range

✓ Volume level control

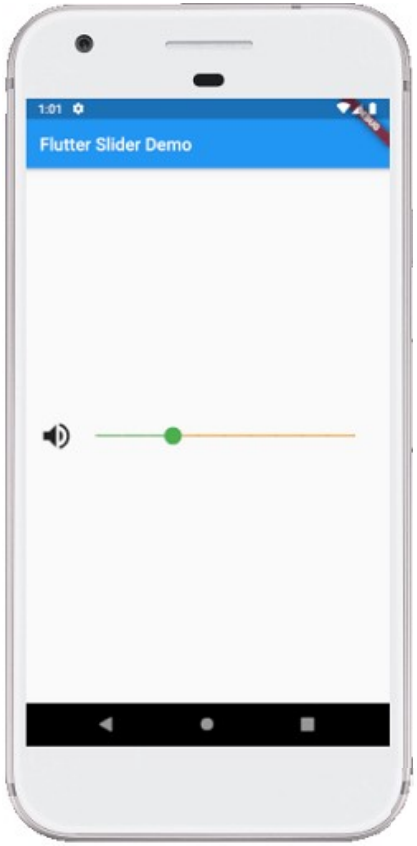
✓ Brightness level control

✓ Playback control

✓ Image view control

✓ Image slider

✓ Color shades control



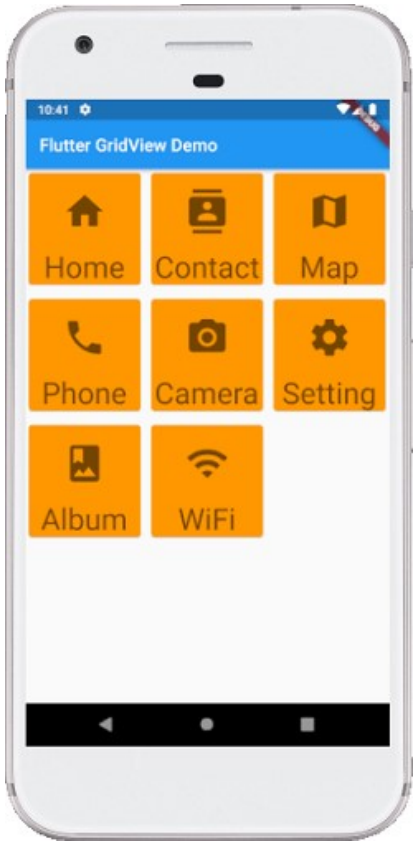
Slider

Attributes	Type	Descriptions
value	double	It is a required argument and used to specify the slider's current value.
onChanged	double	It is a required argument and called during dragging when the user selects a new value for the slider. If it is null, the slider is disabled.
max	double	It is an optional argument and determines the maximum value that can be used by the user. By default, it is 1.0. It should be greater than or equal to min.
min	double	It is an optional argument that determines the minimum value that can be used by the user. By default, it is 0.0. It should be less than or equal to max.
divisions	int	It determines the number of discrete divisions. If it is null, the slider is continuous.
label	string	It specifies the text label that will be shown above the slider. It displays the value of a discrete slider.
activeColor	class Color	It determines the color of the active portion of the slider track.
inactiveColor	class Color	It determines the color of the inactive portion of the slider track.

GridView

- GridView arranges its child controls in a grid i.e. in rows and columns.
- It can have text, image, icon, button etc. as its children.
- The contents of a GridView can be scrolled vertically or horizontally.
- The grid view can be implemented in various ways, which are given below:

1. count()
2. builder()
3. custom()
4. extent()



GridView.builder()

- The builder() method creates a GridView in which the children widgets are created on-demand i.e. dynamically when they become visible on the screen.

```
GridView grid = GridView.builder(gridDelegate: __, itemCount: __ , itemBuilder: (context,index){
});
```

- The common attributes of this widget are:

1. itemCount: It is used to define the amount of data to be displayed.
2. **gridDelegate**: It determines the grid or its divider. Its argument should not be null. It is a required argument.
3. **itemBuilder**: It is used to create items that will be displayed on the grid view. It will be called only when the indices >= zero && indices < itemCount. It is a required argument.
4. scrollDirection:The direction in which the items on GridView scrolls. By default, it scrolls in a **vertical** direction.

GridView.builder()

Step-by-step Procedure:

Step-1 Create object for gridDelegate.

Step-2 Create list of widgets to be added to the grid.

Step-2 Define itemBuilder callback.

GridView.builder()

Step-1

gridDelegate is usually specified with a SliverGridDelegateWithFixedCrossAxisCount.

SliverGridDelegateWithFixedCrossAxisCount d = const SliverGridDelegateWithFixedCrossAxisCount (

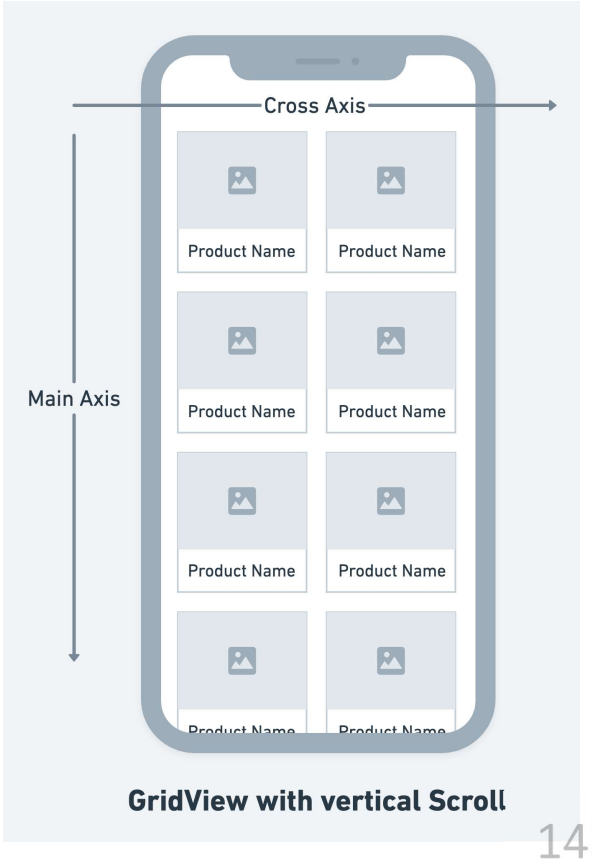
crossAxisSpacing: 20,

mainAxisSpacing: 20,

crossAxisCount: 2,

);

crossAxisCount specifies no. of columns.



GridView with vertical Scroll

GridView.builder()

```
// Create widgets to display in the grid
 List<Widget> data =
   [
     const ListTile(leading:Icon(Icons.man,size:80,color:Colors.blue),title:Text('Profile')),
     const ListTile(leading:Icon(Icons.double_arrow,size:80,color:Colors.red),title:Text('Money Transfer')),
     const ListTile(leading:Icon(Icons.money,size:80,color:Colors.deepPurpleAccent),title:Text('Deposits')),
     const ListTile(leading:Icon(Icons.help,size:80,color:Colors.cyan),title:Text('Help'))
   ];

// Define itemBuilder callback
itemBuilder: (context,index)
   {
      return data[index];
   }
```

Dart programming language fundamentals

- Built-in types
- Variables and constants
- Loops
- Functions
- Object-oriented Programming

Built-in types in Dart

- In Dart, the data type of the variable is defined by its value.
- Dart is an optionally typed language. If the type of a variable is not explicitly specified, the variable's type is automatically **determined depending on the value assigned**.
- Dart supports the following built-in Data types.
  1. Number (**int, double**)
  2. Strings (**String**,Either pair of double-quotes ("hello") or single-quotes('hello')
  3. Boolean (bool,true or false)
  4. List (List [])
  5. Set ({ Value1, Value2,…. })
  6. Map (Map { 'Key' : 'Value' })
  7. Runes (Dart uses the name 'rune' for an integer representing a Unicode code point.)
  8. Symbols (String names used in reflecting out metadata from a library.)

Declaring variables using var keyword

Declaration of variables *without* datatype using var keyword

var <variable_name> = <value>;

OR

var <variable_name>; // Will be initialized to null.

The datatype of such a variable is automatically assigned depending on the value assigned.

Examples:

var n = 100; // The datatype of n becomes int

var firstname = “Jiyan”; // The datatype of firstname becomes String

var weight = 65.50 ; // The datatype of weight becomes double

Declaring *flexible* variables using dynamic

- dynamic allows a variable **with flexible datatype**.
- This means you can assign value of different types to the same variable again and again.

```
void main() {
  dynamic v = 42; // v is assigned an int value.
  print(v); // 42

  v = "Hello, Dart!"; // Now, v is assigned a String value.
  print(v); // Hello, Dart!
}
```

Declaring variables with type

Declaration of variables with datatype

<type> <variable_name> ; // Non-nullable variable

// OK. Uninitialized variable and hence must be assigned a value before using it

Example:

```
int k;
print(“k = $k”);          // Error
k = 100;
print(“k = $k”);          // Valid
```

Declaring variables using type

Declaration of variables with datatype

<type> <variable_name> = <value>;

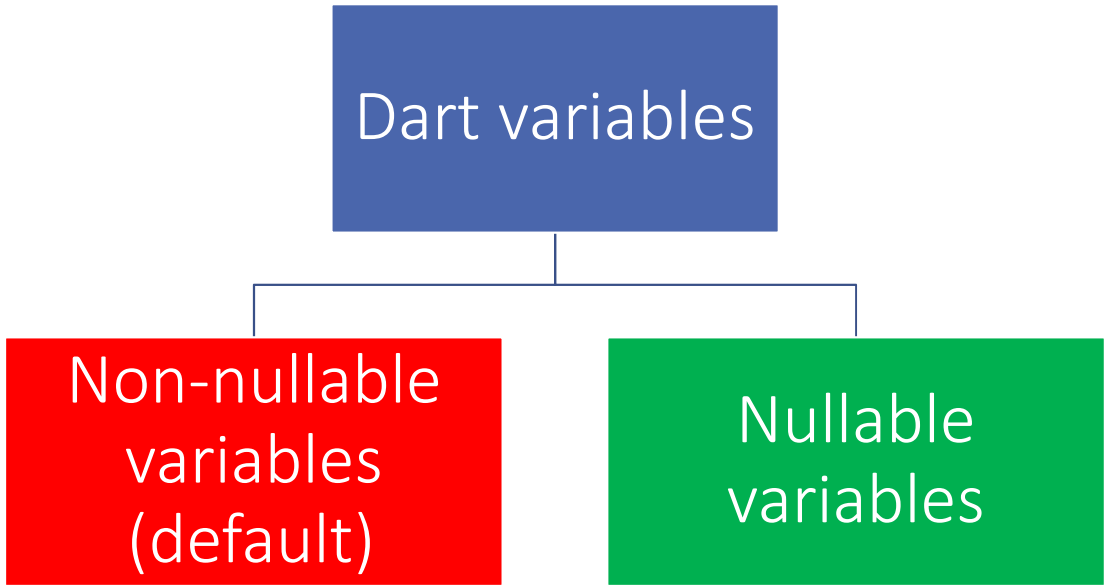
Example:

```
int k = 100;
print("k = $k");          // Valid. It will display k = 100
```

Null safety in dart

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

Null safety in dart



Null safety in dart

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

Example:

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

Null safety in dart

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

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

OR

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

- It means that they can have null value.

Example:

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

Null safety in dart

k is not a nullable variable.	z is a nullable variable.
int k; print(“k = $k”); // Error – k is not initialized.	int? z; print(“z = $z”); // Valid. It will print: z = null

Late variables – Late or Lazy initialization

- If you are sure that a variable will be assigned a value before it is used, but if the compiler gives you an error, you can fix this problem by declaring the variable as **late**.

late <type> <variable_name>;

- Declaring a variable as late gives an assurance to the compiler that the variable will surely be assigned a value before its use.

Late variables – Late or Lazy initialization

Lazy initialization

- When you declare a variable as late but initialize it at its declaration, then the initializer runs the first time the variable is used.
- It means that if the variable is not used at all, the variable is not initialized at all.

late String merit_no_1 = findTopper(exam_score);

Here, the findTopper() function will be called only if the variable merit_no_1 is used.

If the variable merit_no_1 is not used, the resource-consuming findTopper() function will not be called.

const vs. final in Dart

const	final
A variable with the const keyword is initialized at **compile-time** and is already assigned by the time the program is executed.	A final variable will be initialized at **runtime**.
You can't define const inside a class. But you can in a function.	You can define a final variable inside a function as well as a class.
When the state is updated, the const objects are **not** created again.	When the state is updated, the final objects are created **again**.

const total = 100; // Correct
const date = DateTime.now(); // Error
	final date = DateTime.now();

Operators

Arithmetic operators

Dart supports the usual arithmetic operators, as shown in the following table.

Operator	Meaning
`+`	Add
`-`	Subtract
`-expr`	Unary minus, also known as negation (reverse the sign of the expression)
`*`	Multiply
`/`	Divide
`~/`	Divide, returning an integer result
`%`	Get the remainder of an integer division (modulo)

Operators

Equality and relational operators

The following table lists the meanings of equality and relational operators.

Operator	Meaning
==	Equal; see discussion below
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Operators

Type test operators

The `as`, `is`, and `is!` operators are handy for checking types at runtime.

Operator	Meaning
`as`	Typecast (also used to specify library prefixes)
`is`	True if the object has the specified type
`is!`	True if the object doesn't have the specified type

Operators

Assignment operators

As you've already seen, you can assign values using the = operator. To assign only if the assigned-to variable is null, use the ??= operator.

```
// Assign value to a
a = value;
// Assign value to b if b is null; otherwise, b stays the same
b ??= value;
```

Operators

Logical operators

You can invert or combine boolean expressions using the logical operators.

Operator	Meaning
`!expr`	inverts the following expression (changes false to true, and vice versa)
`||`	logical OR
`&&`	logical AND

Operators

Conditional expressions

Dart has two operators that let you concisely evaluate expressions that might otherwise require if-else statements:

`condition ? expr1 : expr2`
If *condition* is true, evaluates *expr1* (and returns its value); otherwise, evaluates and returns the value of *expr2*.

`expr1 ?? expr2`
If *expr1* is non-null, returns its value; otherwise, evaluates and returns the value of *expr2*.

When you need to assign a value based on a boolean expression, consider using `?` and `:`.

Operators

Operator	Name	Meaning
`()`	Function application	Represents a function call
`[]`	Subscript access	Represents a call to the overridable `[]` operator; example: `fooList[1]` passes the int `1` to `fooList` to access the element at index `1`
`?[]`	Conditional subscript access	Like `[]`, but the leftmost operand can be null; example: `fooList?[1]` passes the int `1` to `fooList` to access the element at index `1` unless `fooList` is null (in which case the expression evaluates to null)
`.`	Member access	Refers to a property of an expression; example: `foo.bar` selects property `bar` from expression `foo`
`?.`	Conditional member access	Like `.`, but the leftmost operand can be null; example: `foo?.bar` selects property `bar` from expression `foo` unless `foo` is null (in which case the value of `foo?.bar` is null)
`!`	Non-null assertion operator	Casts an expression to its underlying non-nullable type, throwing a runtime exception if the cast fails; example: `foo!.bar` asserts `foo` is non-null and selects the property `bar`, unless `foo` is null in which case a runtime exception is thrown

Libraries & imports

- Every Dart file (plus its parts) is a library.
- The library helps you create a modular and shareable code base.
- Libraries not only provide APIs, but are a unit of privacy: variables that start with an **underscore** (_) are visible **only inside the library**.
- Libraries can be distributed using packages.
- To use a library, corresponding package is imported with the help of import statement.

Libraries & imports

- For built-in libraries, the URI has the special dart: scheme. For example,

```
import 'dart:html';
```

- For other libraries, you can use a file system path or the package: scheme. The package: scheme specifies libraries provided by a package manager such as the pub tool. For example,

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

Libraries & imports

- If you import two libraries that have conflicting identifiers, then you can specify a prefix for one or both libraries. For example, if library1 and library2 both have an Element class, then you might have code like this:

```
import 'package:lib1/lib1.dart';

import 'package:lib2/lib2.dart' as lib2;

Element element1 = Element();          // Uses Element from lib1.

lib2.Element element2 = lib2.Element(); // Uses Element from lib2.
```

Libraries & imports

- If you import two libraries that have conflicting identifiers, then you can specify a prefix for one or both libraries. For example, if library1 and library2 both have an Element class, then you might have code like this:

```
import 'package:lib1/lib1.dart';

import 'package:lib2/lib2.dart' as lib2;

Element element1 = Element();           // Uses Element from lib1.

lib2.Element element2 = lib2.Element(); // Uses Element from lib2.
```

Libraries & imports

- If you want to use only part of a library, you can selectively import the library. For example:

```
// Import only xyz.
import 'package:lib1/lib1.dart' show xyz;


// Import all names EXCEPT xyz.
import 'package:lib2/lib2.dart' hide xyz;
```

Deferred loading of a library

- Deferred loading (also called lazy loading) allows a web app to load a library on demand, if and when the library is needed.
- Here are some cases when you might use deferred loading:

✓ To reduce a web app's initial startup time.

✓ To perform A/B testing—trying out alternative implementations of an algorithm, for example.

✓ To load rarely used functionality, such as optional screens and dialogs.

Deferred loading of a library

To lazily load a library, you must first import it using deferred as:

```
import 'package:greetings/hello.dart' deferred as hello;
```

When you need the library, invoke loadLibrary() using the library's identifier.

```
Future<void> greet() async
{
  await hello.loadLibrary();
  hello.printGreeting();
}
```

In the given code, the await keyword pauses execution until the library is loaded.

Loops in Dart

for loop

for..in loop (for-each loop)

while loop

do while loop

Functions in Dart

A function is a block of code which performs specific task.

Syntax:

return-type function-name (arguments)

{

    return statement;

}

Anonymous Function

An anonymous function is a block of code which does not have any name.

It can have zero or any number of arguments.

Syntax:

(parameter_list) {

  statement(s)

}

The entire function definition is placed at the point of use.

It can be passed as an argument to another function.

```
void main() {
// Anonymous function
  var cube = (int number) {
    return number * number * number;
  };

  print("The cube of 2 is ${cube(2)}");
  print("The cube of 3 is ${cube(3)}");
}
```

Function with optional parameters

Wrapping a set of function parameters in **[ ]** marks them as optional positional parameters.

If you don't provide a default value, their types must be nullable as their default value will be null.

Function with optional parameters

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

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

Function with named parameters

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

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

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

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

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

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

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

OR

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

Function with named parameters

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

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

Function with required parameters

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

The following example makes the user and password parameters required:

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

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

Object-oriented Programming in Dart

Object-oriented Programming Concepts:

- Class
- Object
- Fields / Data members / Attributes / Properties
- Methods / Member Functions
- Inheritance
- Overloading* vs. Overriding

* Overloading is not supported by Dart. (!)

Object-oriented Programming in Dart

Class

```
class ClassName
{
   <fields>
   <getters/setters>
  <constructor>
 <functions>
}
```

getters() and setter() methods are implicitly added to the class by Dart compiler.

Note - According to the naming convention rule of identifiers, the first letter of the class name must be capital and use no separators.

A class or any of its method can not be declared as final.

Object-oriented Programming in Dart

Dart does not provide private, protected or public keywords for specifying access.

Instead, you can use _ (underscore) at the start of the name to make a data member of a class private.

In Dart, the privacy is at library level rather than class level. It means other classes and functions in the same library still have the access.

So, a data member is either public (if not preceded by _) or private (if preceded by _)

Constructors in Dart

Constructor

- A constructor is used to initialize fields of an object.
- 3 types of constructors in Dart:
  1) Generative Constructor
  2) Unnamed Constructor and
  3) Named Constructor
- Constructors can NOT be overloaded in dart.(!)

Constructors in Dart

Generative Constructor

Syntax:

```
Classname(this.field1,this.field2,...);
```

- It initializes the fields of a class.
- The field names must be prefixed with this.
- It does not have constructor body with { }. Therefore, it can not perform complex initialization.
- It is invoked before other constructors.

Constructors in Dart

Generative Constructor

Example:

```
//Definition inside class
Student(this.id,this.name,this.percentage);

// Use in main or somewhere else
Student s1 = Student(1009,"Akash Shah",83); // Calls GC
```

Constructors in Dart

Unnamed Constructor

Syntax:

```
Classname(this.field1,this.field2,...)
{
   // Complex initialization
}
```

- It initializes the fields of a class.
- It has constructor body with { }. Therefore, it can perform complex initialization.
- Type and no. of parameters of a generative constructor and an unnamed constructor must be different.

Constructors in Dart

Unnamed Constructor

Example:

```
// Definition inside class
Student(this.id,this.name,this.percentage)
 {
   print("Unnamed Constructor called...");
   if(percentage>=70)
   {
     result="Distinction";
   }
 }
```

```
// Use
Student s1 = Student(1009,"Akash Shah",83); // Calls UNC
```

Constructors in Dart

```
//Named Constructor
Syntax:
Classname.Constr-name(this.field1,this.field2,…)
{
   // Complex initialization
}
```

- It has a name other than classname.
- It initializes the fields of a class.
- It has constructor body with { }. Therefore, it can perform complex initialization.

Constructors in Dart

Named Constructor

Example:

```
// Definition inside class
Student.makeResult(this.id,this.name,this.percentage)
 {
   print(“Named Constructor called...");
   if(percentage>=70)
   {
     result="Distinction";
   }
 }
```

```
// Use
Student s1 =
Student.makeResult(1009,"Akash
Shah",83);
```

Constructors in Dart

Constant Constructor

Syntax:

```
const Classname(this.field1,this.field2,...)
```

- It initializes the fields of a class.
- It can not have a body { }.
- All the fields of the class must be declared as final.
- An object created with the help of a constant constructor can not be changed.

Inheritance in Dart

In Dart, one class can inherit another class i.e. dart can create a new class from an existing class. We make use of extend keyword to do so.

```
class parent_class {
...
}

class child_class extends parent_class {
...
}
```

- Child classes inherit all properties and methods except constructors of the parent class.
- Like Java, Dart also doesn't support multiple inheritance.

Inheritance in Dart

Use of super keyword in Dart:

- In Dart, super keyword is used to refer immediate parent class object.
- It is used to call properties and methods of the superclass.
- It can be used to access the data members of parent class when both parent and child have member with same name.
- It can be used to call parameterized constructor of parent class.

- Syntax:

```
// To access parent class variables
super.variable_name;

// To access parent class method
super.method_name();
```

Exception handling

An exception (or runtime error) is an abnormal situation that arises during the execution of a program.

When an exception occurs the normal flow of the program is disrupted and the program terminates abnormally.

Syntax errors are identified by the compiler. However, the compiler can not detect probable exceptions.

Therefore, it is necessary to write code which handles the exception. i.e. systematically notify the user of the problem and take necessary actions. The practice of writing such code is known “Exception handling.”

Exception handling

Four things associated with an Exception:

1. Cause of an exception
2. Statement which throws an exception
3. Effect of exception i.e. the statements which are not executed after an exception is thrown.
4. The code which is executed in response to exception (Exception handler)

Exception handling

Normal program execution

```
void main()
{
  String t1 = "23";
  int age = int.parse(t1);
  print("age = $age Years");
}
```

```
age = 23 Years
```

Output

Exception handling

Program in which an exception is thrown:

```
void main()
{
  String t1 = "a23";
  int age =  int.parse(t1);
  print("age = $age Years");
}
```

Cause of Exception

Exception occurs here

So, this statement is NOT executed at all.

Output

```
Unhandled exception:
FormatException: Invalid radix-10 number (at character 1)
a23
^
```

Exception handling

Normal program execution

```dart
void main()
{
  int a = 5;
  int b = 2;
  int result = a~/b;
  // ~/ gives integer result i.e. quotient

  print("a = $a");
  print("b = $b");
  print("result = $result");
}
```

```
a = 5
b = 2
result = 2
```

Output

Program in which an exception is thrown

```dart
void main()
{
  int a = 5;
  int b = 0;
  int result = a~/b;
  // ~/ gives integer result i.e. quotient

  print("a = $a");
  print("b = $b");
  print("result = $result");
}
```

```
Unhandled exception:
IntegerDivisionByZeroException
#0      int.~/ (dart:core-patch/integers.dart:30:7)
#1      main (package:exceptionhandling/main.dart:5:17)
```

Exception handling

The try / on / catch / finally blocks

The **try** block embeds code that might possibly result in an exception.

The **on** block is used when the exception **type** needs to be specified.

There can be more than one **on** blocks after a try block.

The **catch** block is used when the handler needs the exception **object**.

There can be more than one **catch** blocks after a try block.

The **finally** block includes code that should be executed irrespective of an exception's occurrence. The optional **finally** block executes unconditionally after **try/on/catch**.

The try block must be followed by either at least one on / catch block or a finally block.

Exception handling using try-on

```
try
{
// Code which may throw exception
}
on Exception-type1
{
// Code to execute when exception of type 1 is thrown
}
on Exception-type2
{
// Code to execute when exception of type 2 is thrown
}
```

Exception handling using try and single on block

```
void main()
{
  String t1 = "a23";
  try
  {
    int age = int.parse(t1);
    print("age = $age Years");
  }
  on FormatException
  {
      print("FormatException occurred...");
   }
}
```

Output

```
FormatException occurred...
```

Exception handling using try and single on block

```dart
void main()
{
  String t1 = "23";
  int a = 5;
  int b = 0;

  try
  {
    int result = a~/b;
    print("result = $result");

    int age = int.parse(t1);
    print("age = $age Years");
  }
  on FormatException
  {
      print("FormatException occurred...");
  }

}
```

Output

```
Unhandled exception:
IntegerDivisionByZeroException
#0      int.~/ (dart:core-patch/integers.dart:30:7)
#1      main (package:exceptionhandling/try-single-on.dart:9:19)
```

Exception handling using try and multiple on blocks

```
void main() {
  String t1 = "a23";
  int a=5;
  int b=2;
  try
  {
    int age = int.parse(t1);
    print("age = $age Years");

    int result = a~/b;
    print("result = $result");
  }
  on FormatException
  {
      print("FormatException occurred...");
  }
  on UnsupportedError
  {
    print("UnsupportedError occurred...");
  }
}
```

Exception handling using try and single catch

```
try
{
// Code which may throw exception
}
catch(e)
{
// All-in-one catch – The same block is executed for all types of exception….
// Code to execute when any type of exception is thrown
}
```

Exception handling using try-catch

```dart
void main()
{
  String t1 = "a23";
  int a=5;
  int b=2;
  try
  {
    int age = int.parse(t1);
    print("age = $age Years");

    int result = a~/b;
    print("result = $result");
  }
  catch (e)
  {
    print("Some exception occurred!");
  }
}
```

Output

```
Some exception occurred!
```

```dart
void main()
{
  String t1 = "23";
  int a=5;
  int b=0;
  try
  {
    int age = int.parse(t1);
    print("age = $age Years");

    int result = a~/b;
    print("result = $result");
  }
  catch (e)
  {
    print("Some exception occurred!");
  }
}
```

Exception handling using try-on-catch

```
try
{
// Code which may throw exception
}
on Exception-type1 catch(e)
{
// Code to execute when exception of type 1 is thrown
}
on Exception-type2 catch(e)
{
// Code to execute when exception of type 2 is thrown
}
```

Exception handling using try-on-catch

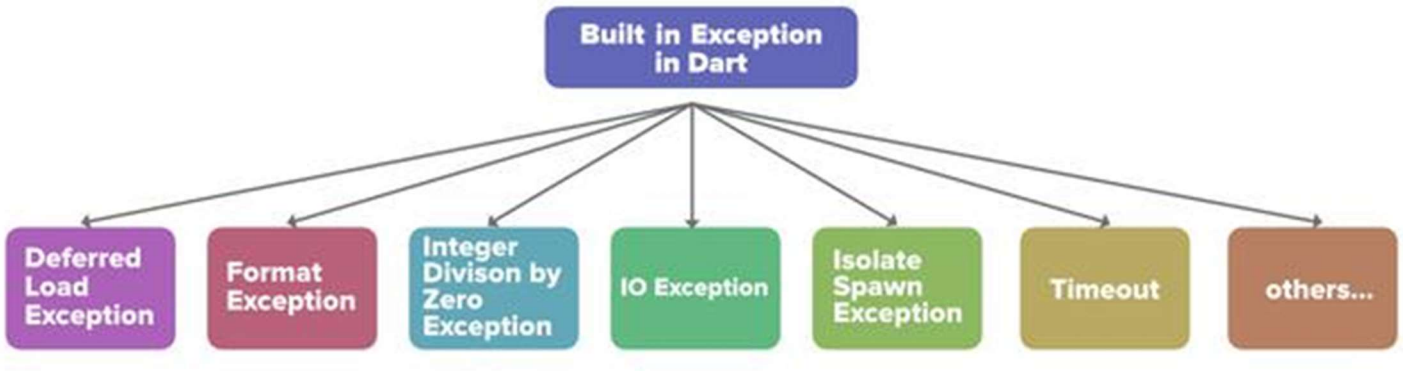
```
void main() {
  String t1 = "a23";
  int a=5;
  int b=2;
  try
  {
    int result = a~/b;
    print("result = $result");

    int age = int.parse(t1);
    print("age = $age Years");
  }
  on FormatException catch (e)  {
      print("FormatException occurred...");
      print(e.message);
  }
  on UnsupportedError catch (e)  {
    print("UnsupportedError occurred...");
    print(e.message);
  }
  catch(e) { print("Some other type of exception occurred..."); }
```

Exception Classes

Built-in Dart exceptions:

Every exception in Dart is a subtype of the pre-defined interface **Exception**.



Dart Built-in Exception Classes

Sr. No.	Exceptions & Description
1	**DeferredLoadException*** Thrown when a deferred library fails to load. * Flutter has the capability to build apps that can download additional Dart code and assets at runtime. This allows apps to reduce install apk size and download features and assets when needed by the user. We refer to each uniquely downloadable bundle of Dart libraries and assets as a "deferred component". This is achieved by using Dart's deferred imports
2	**FormatException** Exception thrown when a string or some other data does not have an expected format and cannot be parsed or processed.
3	**IntegerDivisionByZeroException** Thrown when a number is divided by zero.
4	**IOException** Base class for all Input-Output related exceptions.
5	**IsolateSpawnException*** Thrown when an isolate cannot be created. * An isolate is a is a part of the program with its own memory and event loop.
6	**Timeout** Thrown when a scheduled timeout happens while waiting for an async result.

Custom (OR User-defined) Exceptions

Dart supports creating custom exception classes.

A custom exception class is needed to represent deal with a problem which is very specific in nature and which is not covered by built-in exception class.

The syntax for defining a custom exception is given below:

```
class Custom_exception_Name implements Exception
{
  // can contain constructors, variables and methods
}
```

Throwing an Exception

The **throw** keyword is used to explicitly raise an exception. A raised exception should be handled to prevent the program from exiting abruptly.

The syntax for raising an exception explicitly is – throw Exception_name();

Custom Exceptions

```
// Custom Exception Class
class InvalidAmountException implements Exception
{
  String errorMessage = "InvalidAmountException..\nAmount must be greater than 0"
      " AND in multiple of 100";
}
```

Custom Exceptions

```
void main()
{
  try
  {
   double amount=1250;
   if(amount<0 || amount%100!=0)
   {
     throw InvalidAmountException(); // Throwing custom exception...
   }
   print("Rs. $amount credited successfully.");
  }
  on InvalidAmountException catch(e)
  {   print(e.errorMessage);
  }
  catch(e)  {  print("Some other type of exception occurred...");  }
}
```

Asynchronous programming in Dart

Key terms:

synchronous operation: A synchronous operation blocks other operations from executing until it completes.

synchronous function: A synchronous function only performs synchronous operations.

asynchronous operation: An asynchronous operation allows other operations to execute while itself is still working and before it completes.

asynchronous function: An asynchronous function performs at least one asynchronous operation and can also perform synchronous operations.

async function

async is used to make a function asynchronous.

Future<T> OR Future<void> function-name() **async**

```
{
   // Statements before await…
   await _____________;
   // Statements after await….
}
```

If the function has a declared return type, then the return-type will be **Future<T>**, where T is the type of the value that the function returns.

If the function does not explicitly return a value, then the return type will be **Future<void>**.

async function

An asynchronous function cannot immediately return the result.

Therefore it immediately returns a Future which will later be "complete" with the result.

Hence, an object of Future<T> class is used as the return-type of an asynchronous computation.

await

Future<T> OR Future<void> function-name() **async**

{

// Statements before await…

await XYZ;

// Statements after await….

}

await XYZ makes sure that operation/function XYZ completes its execution and then only the next statement is executed.

You can use the await keyword **to get the completed result** of an asynchronous function.

await

```
void main()
{
  String EOTY="Unknown";
  print("Performance Review started...");

  Future.delayed(const Duration(seconds:3),()
  {
    // Let us assume that it takes 3 seconds to compare/analyze performance of
    // large number of employees in an organization.
    EOTY= "Rajat Desai";
  });  // Future.delayed

  print("Performance Review completed...");

  print("Employee of the Year is ** $EOTY **");

}
```

without await

Output

```
Performance Review started...
Performance Review completed...
Employee of the Year is ** Unknown **
```

await

```
void main() async
{
  String EOTY="Unknown";
  print("Performance Review started...");

  // await XYZ makes sure that operation/function XYZ completes its execution and
  // then only the next statement is executed.
  await Future.delayed(const Duration(seconds:3),()
  {
    // Let us assume that it takes 3 seconds to compare/analyze performance of
    // large number of employees in an organization.
    EOTY= "Rajat Desai";
  });  // Future.delayed

  print("Performance Review completed...");

  print("Employee of the Year is ** $EOTY **");

}
```

with await

Output

```
Performance Review started...
Performance Review completed...
Employee of the Year is ** Rajat Desai **
```

The Future<T> class

Future

Future represents a value which may be incomplete at the moment and may be completed in future.

Future<T> represents a value of type T which may be incomplete at the moment and may be completed in future.

Future<T> is used as the return-type of an async function.

await, async and Future

States of Future

Uncompleted

When you call an asynchronous function, it returns an uncompleted future.

That future is waiting for the function's asynchronous operation to finish or to throw an error.

Completed

If the asynchronous operation succeeds, the future completes with a value. Otherwise, it completes with an error.

await, async and Future

Completed Future

A future of type Future<T> completes with a value of type T.

For example, a future with type Future<String> produces a **string** value.

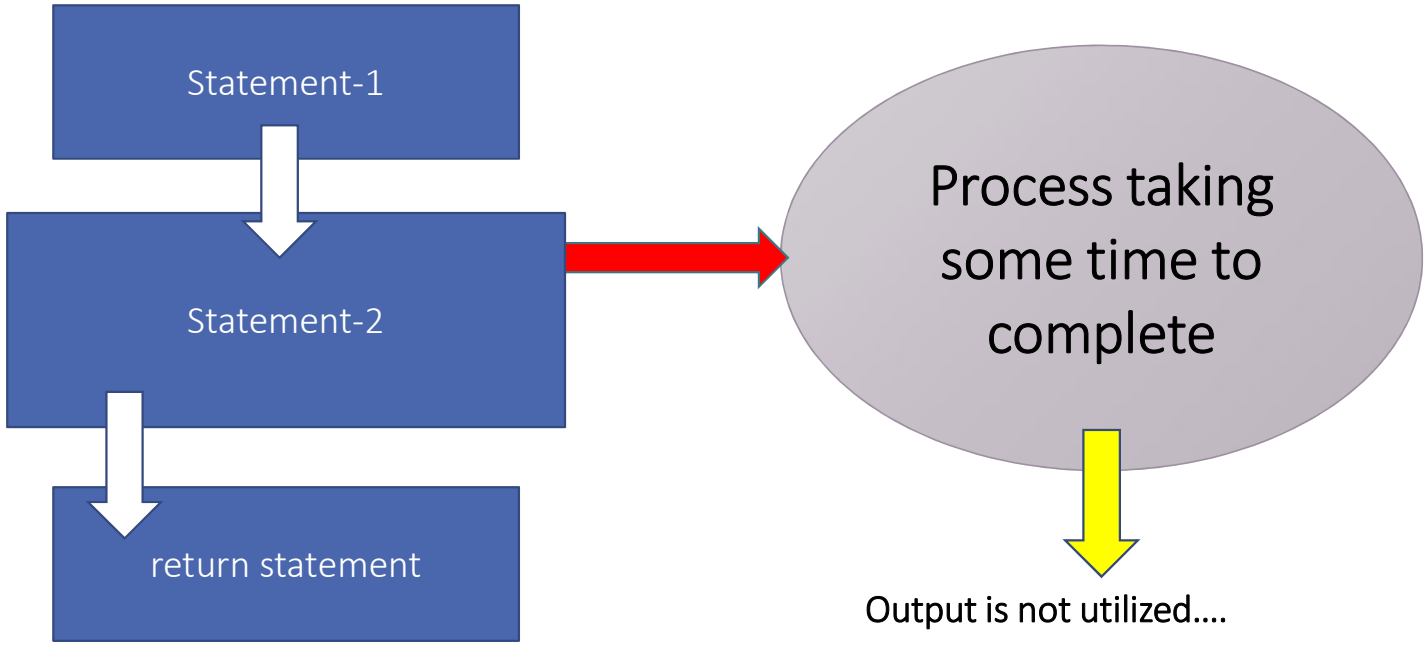
If a future does not produce a usable value, then the future's type is **Future<void>.**

What if the async operation fails?

If the asynchronous operation performed by the function fails for any reason, the future completes with an error.

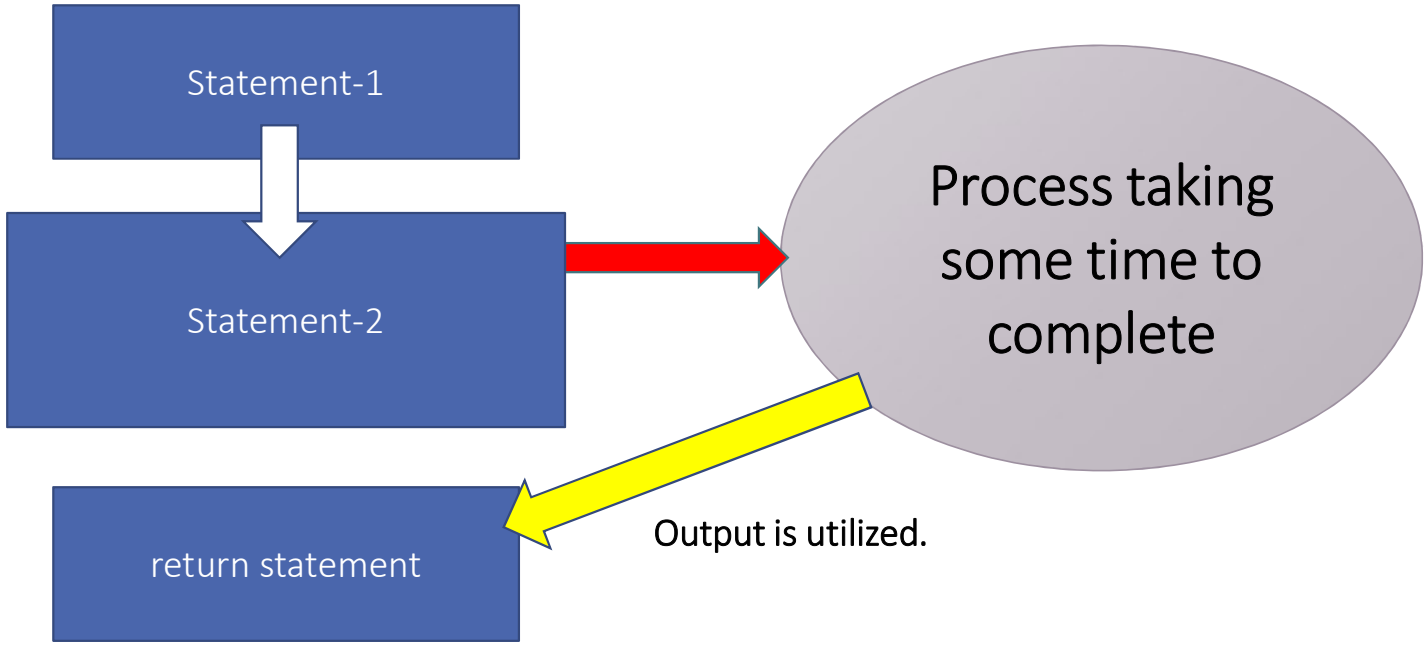
await, async and Future

A function **without** await:



await, async and Future

A function **with** await:



Future.delayed constructor

Creates a future that runs its computation after a delay.

Syntax:

Future.delayed(Duration duration, Function/Code);

Duration can be specified as: Duration(seconds:__)

Future.delayed constructor

```
void greet()
{
  print("Hello...after 3 seconds");
}

void main() async
{
  print("Start");
  await Future.delayed(const Duration(seconds:3), greet);
  print("End");
}

/* Output:
Start
Hello...after 3 seconds
End
*/
```

Future.delayed constructor

```dart
int getCount()
{
  int count=100;
  print("Returning 100...after 5 seconds");
  return count;
}

void main() async
{
  int v=0;
  print("Start");
  v = await Future.delayed(const Duration(seconds:5), getCount);
  print("v = $v");
  print("End");
}
/*
Start
Returning 100...after 5 seconds
v = 100
End
*/
```

mixins

Mixins are classes defined with **mixin** keyword, from which we can use methods or variables without having to extend them.

Mixin normally contains common code (i.e. functionality) which is useful to multiple classes which are not related to one another.

It is also an indirect way to use features from multiple classes.

To accomplish this, we use the **with** keyword, followed by name of a mixin, while creating the derived class.

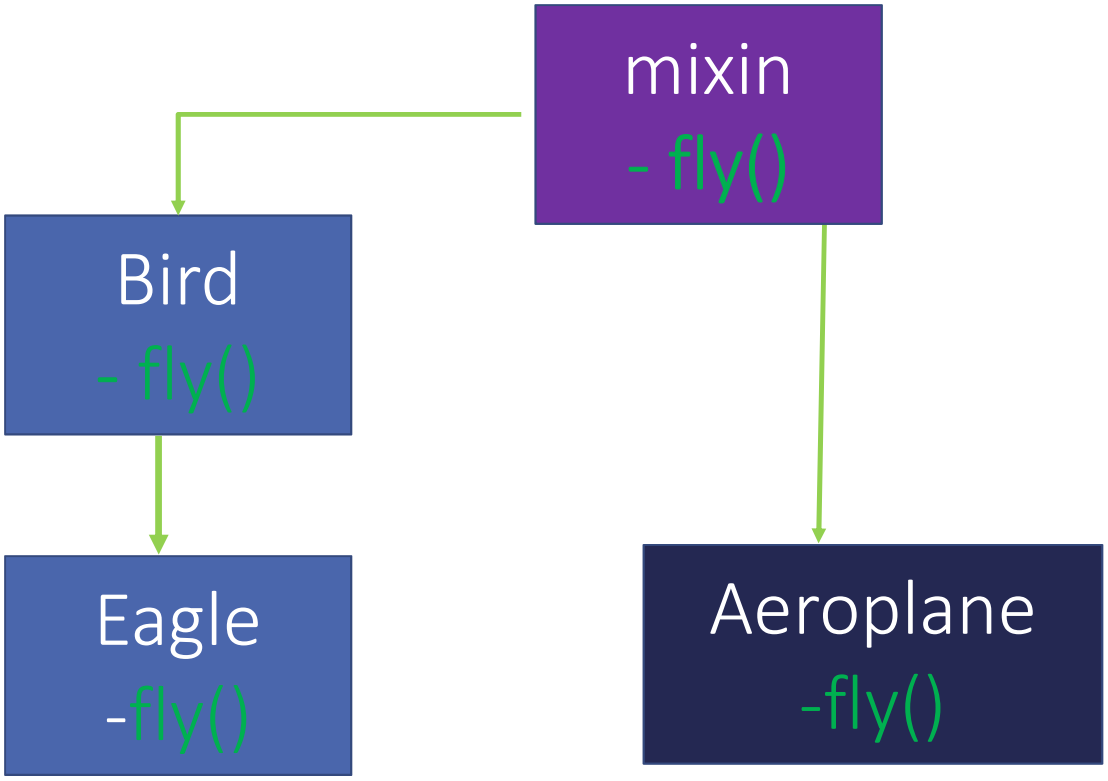
You can not define constructors in mixin.

You can not create objects of a mixin.

One mixin can be created with the help on other mixins using **on** keyword.

mixins

mixins



mixins

```
mixin mixin-name
{
}
class derived-class extends base-class with mixin-name
{
}
class derived-class extends base-class with mixin-name1, mixin-name2,...
{
}
```

Adding or updating a document

Step-1 Create a CollectionReference

CollectionReference coll = FirebaseFirestore.instance.collection(‘collection-name');

Step-2 Create a DocumentReference

DocumentReference dr =

FirebaseFirestore.instance.collection(‘collection-name').doc();

OR

DocumentReference doc =

FirebaseFirestore.instance.collection(‘collection-name').doc(“id”);

Adding or updating a document

Step-3 Create a json

```
final json = {
      'id': doc.id, // This assigns unique id to a document in the collection.
      'title': title
    };
```

Step-4 Call set() or add() on DocumentReference

```
await doc.set(json);          OR     await cr.add(json);
```

Fetching documents

Snapshot

Cloud Firestore retrieves data in a structure called a **Snapshot**.

A DocumentSnapshot contains data for a **single** document.

A QuerySnapshot returns data for **zero or more** documents.

Stream

Streams provide an asynchronous sequence of data.

StreamBuilder

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

When you want to listen to changes constantly, and want the data to get updated without hot reload/restart

Stream and StreamBuilder

Stream<QuerySnapshot> st =
FirebaseFirestore.instance.collection("books").snapshots();

OR

FirebaseFirestore.instance.collection("books").where(…).snapshots();

StreamBuilder sb =

StreamBuilder<QuerySnapshot>(stream:st, builder: (context,snapshot) {….}

No. of documents: snapshot.data!.docs.length

Firebase Authentication

Firebase Authentication is a pre-configured backend service that makes it really easy to integrate with a mobile app using an SDK.

You do not have to maintain any backend infrastructure for the authentication process and Firebase supports integration with popular identity providers such as Google, Facebook, and GitHub.

Firebase Authentication

Using Email/Password

1. Add required dependency to pubspec.yaml.

firebase_auth: ^4.4.0

2. Import firebase_auth.dart file.

import 'package:firebase_auth/firebase_auth.dart';

3. Use appropriate method with FirebaseAuth.instance

FirebaseAuth.instance._____();

Firebase Authentication

Firebase Authentication

1. Register/ Sign-up + Email Verification

```
await FirebaseAuth.instance.createUserWithEmailAndPassword
(email: __ , password: __ );
var user = FirebaseAuth.instance.currentUser;
await user.sendEmailVerification();
```

2. Sign-in

```
await FirebaseAuth.instance.signInWithEmailAndPassword(email: ___, password: ___);
```

3. Sign-out

```
await FirebaseAuth.instance.signOut();
```

Firebase Authentication

await FirebaseAuth.instance.signInWithEmailAndPassword(..);

var user = FirebaseAuth.instance.currentUser;

user. _____

- email
- displayName
- phoneNumber
- photoURL
- **await user!.sendEmailVerification();**

Firebase Authentication

A *FirebaseAuthException* maybe thrown with the following error code:

invalid-email: Thrown if the email address is not valid.

user-disabled: Thrown if the user corresponding to the given email has been disabled.

user-not-found: Thrown if there is no user corresponding to the given email.

wrong-password: Thrown if the password is invalid for the given email, or the account corresponding to the email does not have a password set.

```
try{ …. }
 on FirebaseAuthException catch (error)
     {
        debugPrint(error.code);
     }
```

Authentication with Google

Add the following dependency to pubspec.yaml:

google_sign_in: ^6.1.0

Import the following package:

import 'package:google_sign_in/google_sign_in.dart';

In Firebase:

Fill up SHA-1 and SHA-256 fingerprints

Authentication with Google

GoogleSignIn() - Initializes global sign-in configuration settings.

GoogleSignIn googleSignIn = GoogleSignIn();

SignIn() - Starts the interactive sign-in process.

GoogleSignInAccount? googleSignInAccount = await googleSignIn.signIn();

Authentication with Google

```
GoogleSignInAuthentication googleSignInAuthentication =
        await googleSignInAccount.authentication;

AuthCredential authCredential = GoogleAuthProvider.credential(
            idToken: googleSignInAuthentication.idToken,
            accessToken: googleSignInAuthentication.accessToken);
```

Authentication with Google

```
final UserCredential userCredential =
        await FirebaseAuth.instance.signInWithCredential(authCredential);
```

show vs. hide in import statement

import 'package:google_maps/google_maps.dart' **show LatLng**;

With this you would be able to access **only LatLng but nothing else from that library**. The opposite of this is hide:

import 'package:google_maps/google_maps.dart' **hide LatLng;**

With this you would be able to access everything from that library **except for LatLng.**

as in import statement

as is usually used when there are conflicting classes in your imported library.

For example if you have a library 'my_library.dart' that contains a class named Stream and you also want to use Stream class from dart:async and then:

```
import 'dart:async';
import 'my_library.dart';


void main() {
  Stream stream = new Stream.fromIterable([1, 2]);
}
```

This way, we don't know whether this Stream class is from async library or your own library.

as in import statement

```
import 'dart:async';
import 'my_library.dart' as myLib;

void main() {
  Stream stream = Stream.fromIterable([1, 2]); // from async
  myLib.Stream myCustomStream = myLib.Stream(); // from your library
}
```

Using http with flutter

The http package contains a set of functions and classes that make it easy to consume HTTP resources.

It is multi-platform, and supports mobile, desktop, and the browser.

Dependency: http: ^1.2.1

To import this package: import 'package:http/http.dart' as http;

Also import the convert package: import 'dart:convert';

Using http with flutter

The get() function

Sends an HTTP GET request with the given headers to the given URL.

This automatically initializes a new Client and closes that client once the request is complete.

If you are planning on making multiple requests to the same server, you should use a single Client for all of those requests.

Using http with flutter

```
Uri url = Uri.parse("https://randomuser.me/api/?results=10");

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

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

Using http with flutter

Original data(format) at the resource:

[{"gender":"male","name":{"title":"Mr","first":"Frederikke","last":"Jensen"},

"email":"frederikke.jensen@example.com",

"picture":{"large":"https://randomuser.me/api/portraits/men/97.jpg",

"medium":"https://randomuser.me/api/portraits/med/men/97.jpg",

"thumbnail":"https://randomuser.me/api/portraits/thumb/men/97.jpg"},

Using http with flutter

Accessing data in Flutter

```
data[index]['email']
data[index]['name']['first']
data[index]['picture']['thumbnail']
```

Accessing location of the device

Steps:

1. Add required dependency e.g. location, to pubspec.yaml file.
2. Import required dart file e.g. location.dart file.
3. Use a Stateful widget and write code to get location in State class.
   a) Define an async function and write all the code inside it.
   b) Invoke the async function inside callback of some other widget.

Accessing location of the device

Name of dependency: location

Version: 5.0.3

```yaml
dependencies:
  flutter:
    sdk: flutter
  location: ^5.0.3
```

pubspec.yaml

Accessing location of the device

Inside State class…

```
void getLocationDetails() async
{
   // Create a Location object
   // Check if Location service is enabled.
   // Check if permission to access location is available.
   // Access and display location
}
// Call async function inside callback of some other widget
() { setState( (){ getLocationDetails(); } ) };
```

Accessing location of the device

Create a Location object:

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

Check if the location service is enabled:

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

If not, request to enable it:

```
if (isServiceEnabled==false)
  {
    await location.requestService();
  }
```

Accessing location of the device

Check if permission to access the location is given:

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

If not, request the permission:

```
if (permissionStatus == PermissionStatus.denied)
{
    await location.requestPermission();
}
```

Accessing location of the device

Get the current location:

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

Get the value of latitude and longitude:

```
latitude = currentLocation.latitude.toString();

longitude = currentLocation.longitude.toString();
```

Converting location into address

Dependency

geocoding: any

Import

```
import 'package:geocoding/geocoding.dart' show placemarkFromCoordinates,Placemark;
```

Converting location into address

```
List<Placemark> addresses = await
    placemarkFromCoordinates(currentLocation.latitude, currentLocation.longitude);
- addresses.first;
- addresses.country
```