

Building User Interface with Flutter & Object Oriented Concept

Week 7-8

Presented by: Carissa C. Morano
ASC/ MSCpE/ MIT
Faculty Member




Building User Interface



LEARNING OUTCOMES

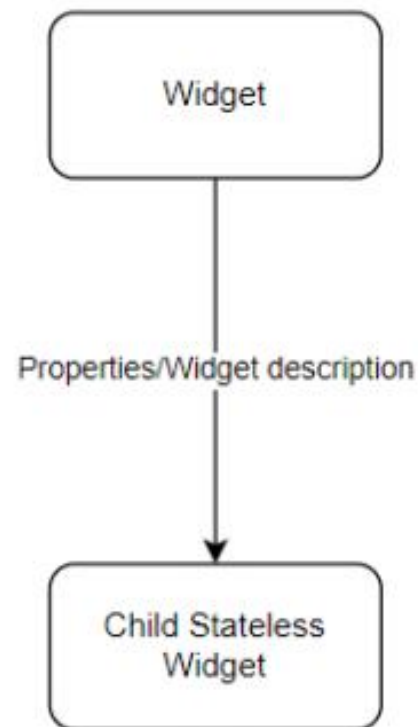
Here's what I will teach you in this course material:

- Understand the differences of stateful widget and stateless widget.
 - Explain the basic widget and properties to build visually appealing UI.
 - Understand how the widget propagate down to widget tree structure.
- 

STATELESS WIDGET VS STATEFUL WIDGET

Stateless widgets

A typical user interface is made up of various widgets, with some of them having fixed properties that don't change after creation or being instantiated. This means stateless widgets don't have a state of their own and control of their appearance to their parent widgets in the widget tree. An example of a stateless widget is as follows.



The description of child widgets is determined by the parent widget, and they cannot change it on their own. This means in code that stateless widgets have properties that are fixed during construction, and they are the only properties that need to be displayed on the device screen.

- A StatelessWidget is immutable, meaning it cannot change once it is built.
- It only depends on the input parameters (constructor values).
- If you want to update the UI, you must recreate the widget by calling setState() in a parent widget.

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

class MyStatelessWidget extends StatelessWidget {
  final String text;

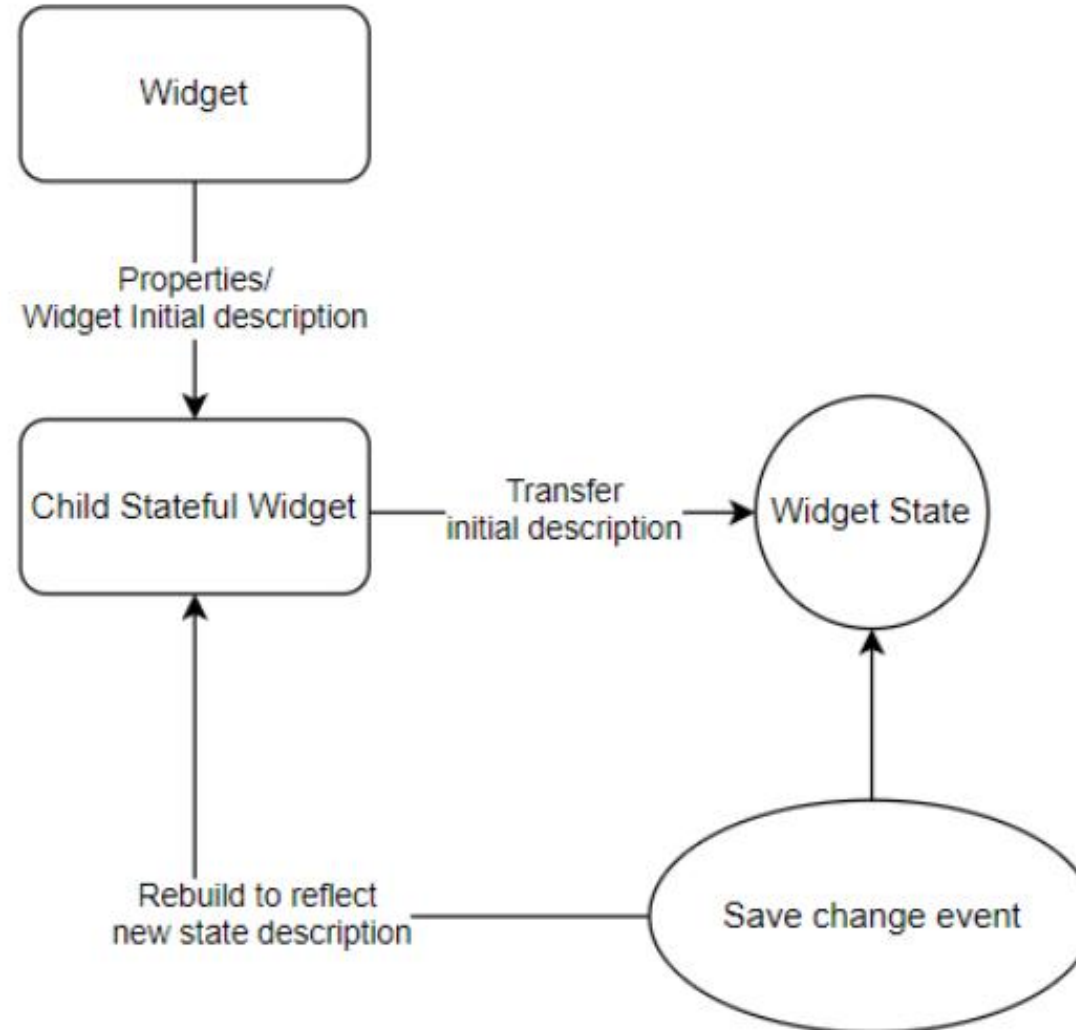
  MyStatelessWidget({required this.text});

  @override
  Widget build(BuildContext context) {
    return Text(text);
  }
}
```

This widget cannot change its text value unless it is rebuilt by its parent.

Stateful widgets

The stateful widget can change their descriptions dynamically over time. Unlike stateless widgets, which receive descriptions from their parent and remain unchanged throughout their lifetime. The stateful widget has an accompanying State class that represents their current state, and the widget's state can be updated as needed. The following diagram shows the concept of stateful widget.



- A StatefulWidget is mutable, meaning it can change after being built.
- It maintains state, which allows it to update the UI dynamically without needing to be rebuilt by its parent.
- It consists of two classes:
 - The widget class (MyStatefulWidget) → This is immutable.
 - The state class (_MyStatefulWidgetState) → This is mutable.

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

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

```
class _MyStatefulWidgetState extends State<MyStatefulWidget> {  
  int counter = 0;  
  
  void increment() {  
    setState(() {  
      counter++;  
    });  
  }  
}
```

```
@override  
Widget build(BuildContext context) {  
  return Column(  
    children: [  
      Text('Counter: $counter'),  
      ElevatedButton(  
        onPressed: increment,  
        child: Text('Increment'),  
      ),  
    ],  
  );  
}
```


Stateless widget

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Flutter Demo',  
      theme: ThemeData(  
        primarySwatch: Colors.blue,  
      ),  
      home: MyHomePage(title: 'Flutter Demo Home Page'),  
    );  
  }  
}
```

Stateful widget



The State object `_MyHomePageState` contains properties on how the widget looks. The example class `MyHomePage` below extends the `StatefulWidget`. The `createState` is a valid State object that the class must return an instance which `_MyHomePageState`.

```
class MyHomePage extends StatefulWidget {  
  MyHomePage({Key key, this.title}) : super(key: key);  
  final String title;  
  
  @override  
  _MyHomePageState createState() => _MyHomePageState();  
}
```

Below is the `_MyHomePageState` class representing the State object.

```
class _MyHomePageState extends State<MyHomePage> {  
  int _counter = 0;  
  
  void _incrementCounter() {  
    setState(() {  
      _counter++;  
    });  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(widget.title),  
      ),  
      body: Center(  

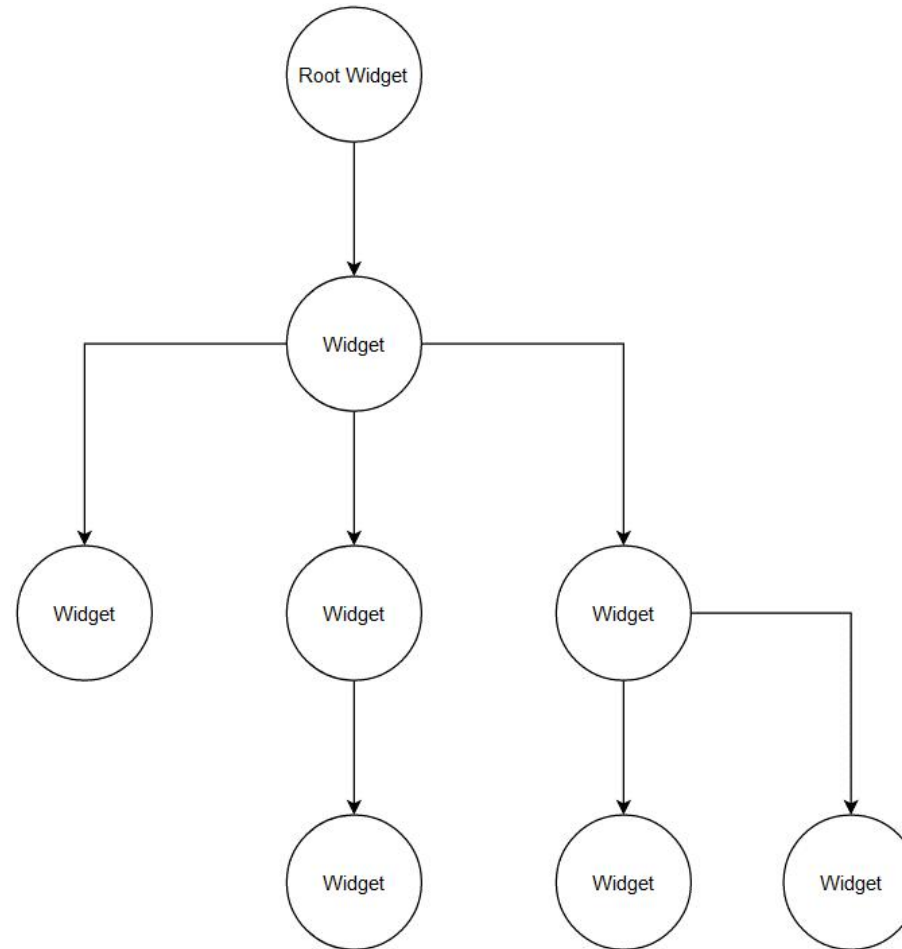
```

Feature	Stateless Widget	Stateful Widget
Mutability	Immutable (cannot change)	Mutable (can change)
State	No internal state	Has internal state
Rebuild	Rebuilt by parent widget	Can rebuild itself using <code>setState()</code>
Use Case	Static UI elements (text, icons, images)	Dynamic UI elements (buttons, animations, forms)

INHERITED WIDGETS

`InheritedWidget` is another type of widget besides `StatelessWidget` and `StatefulWidget`. Sometimes we need to replicate the information of parent widget down to another widget in the widget tree.



Widget Key Property

The key property helps the preservation of a widget's state between rebuilds. This is an important property for Flutter widgets. It facilitates the transition from the widgets tree to the element tree. The key is most used when dealing with collections of widgets of the same type; without keys, the element tree would not know which state corresponds to which widget because they would all be of the same type. For example, whenever a widget changes its position or level in the widgets tree, a matching operation is performed in the elements tree to determine what needs to be updated in the screen to reflect the new widget structure.

Why Do We Need key?

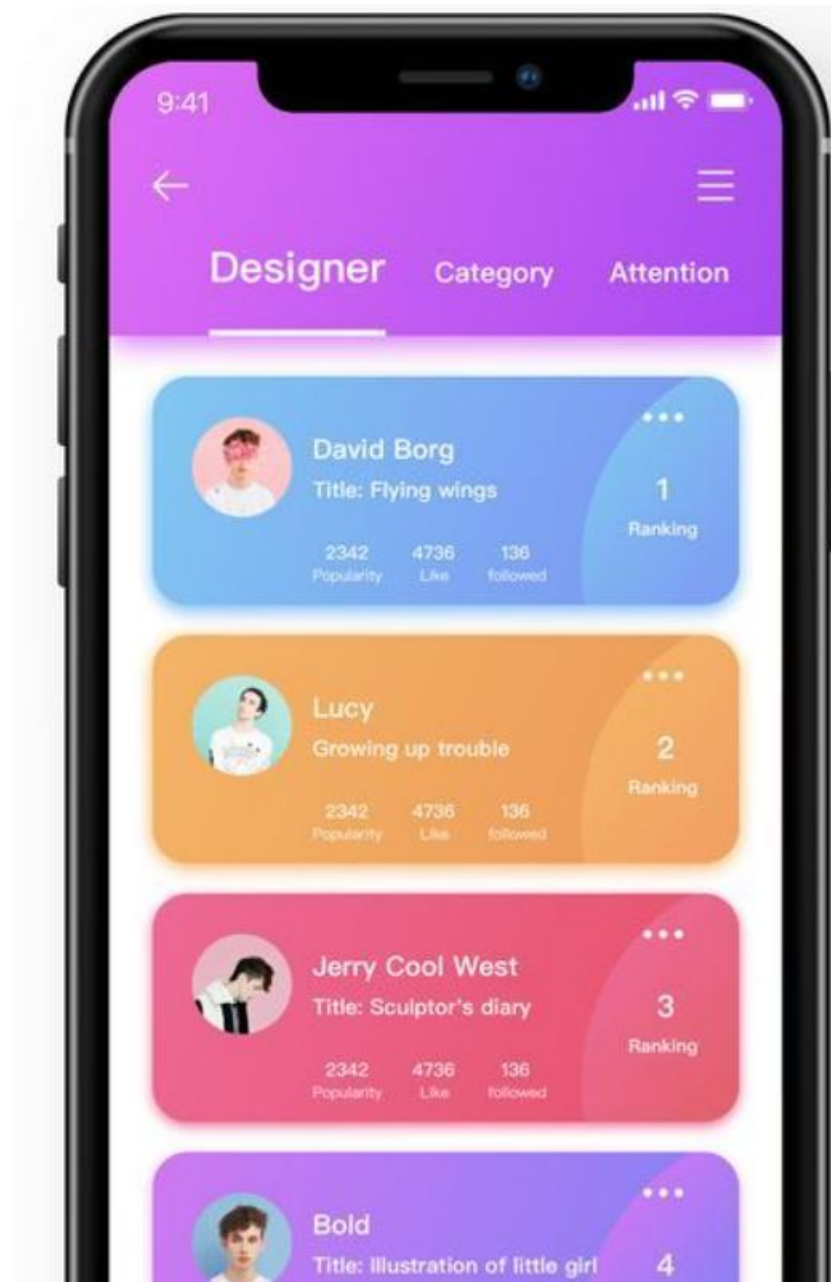
Flutter's widget tree rebuilding process sometimes replaces existing widgets with new ones. Without a key, Flutter may unnecessarily recreate widgets, losing their state. A key ensures that a widget maintains its identity, preventing unnecessary rebuilds.

BASIC WIDGET

A widget is simply an instruction that you put in place in your code, and they are the basic building blocks of a Flutter application's UI. A widget is a reusable component that can display and/or interact with data such as text, images, and other content. In the context of mobile app development, widgets are often used to create UI elements such as buttons, text boxes, sliders, and more.

1. Container: A container is one of the most useful Flutter widgets. A container can be used to store one or more widgets and dress them according to our choice. The container is composed of some properties such as width, height, margin, and padding. These are just the basic properties of containers.

```
container(  
  
  width: double.infinity,  
  
  height: 200,  
  
  margin: EdgeInsets.all(20),  
  
  padding: EdgeInsets.all(20),  
  
  decoration: BoxDecoration(  
  
    borderRadius: BorderRadius.circular(10),  
  
    color: Colors.red  
  
  ),  
  
  child: Text(  
  
    'Hello World',  
  
    style: TextStyle(  
  
      fontSize: 20,  
  
      color: Colors.white  
  
    ),  
  
  ),  
)
```



2. Text: Every application you create has text on it. A text widget is used to display string of

characters in our application.

The text has a constructor that takes the arguments such as

string, style, maxLines, textAlign and others.

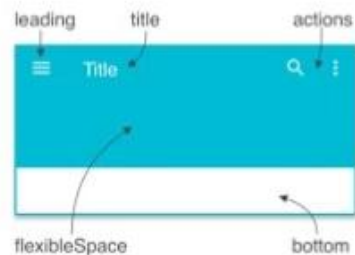
```
Text (  
  'Hello World',  
  style: TextStyle(  
    fontSize: 20,  
    fontWeight: FontWeight.bold,  
    color: Colors.red  
  ),  
)
```

Scaffold



UI / UX

AppBar



Text

```
onPanUpdate:  
DragUpdateDetails(Offset(0.3, 0.0))
```

RichText

Flutter World for **Mobile**

SafeArea

No SafeArea



With SafeArea



Column

Column



Vertically Aligned

Row

Row



Horizontally Aligned

Container



Button

Barista

Default

Barista

StadiumBorder

Barista

UnderlineInputBorder

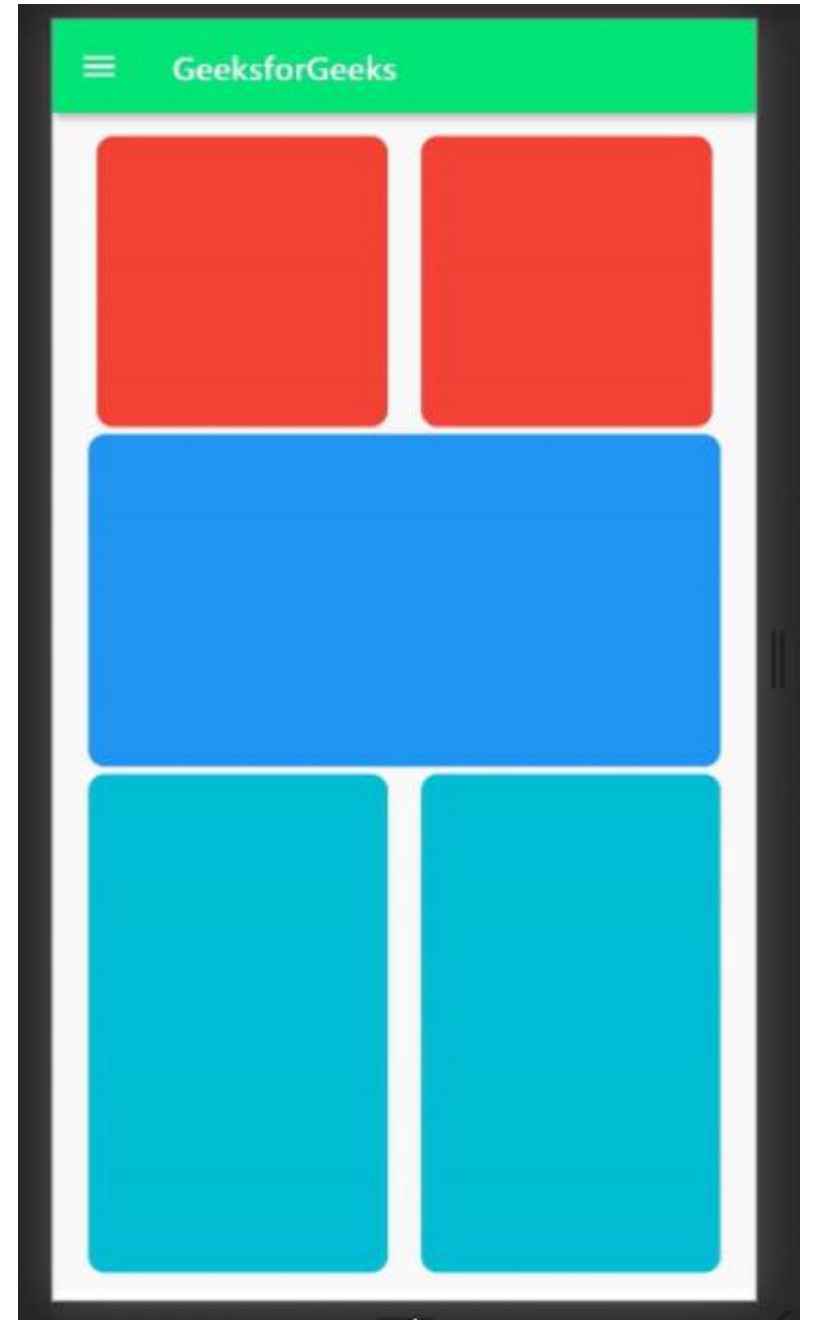
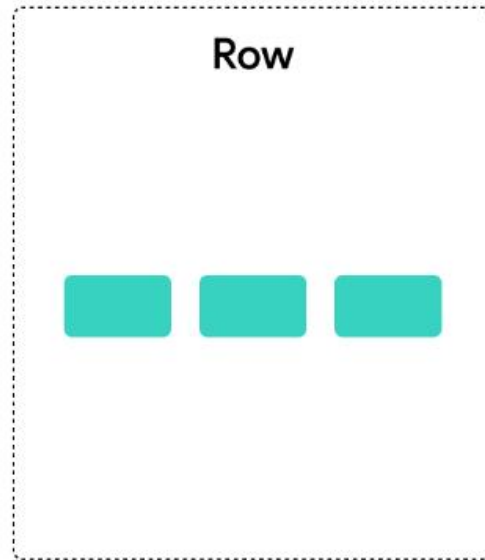
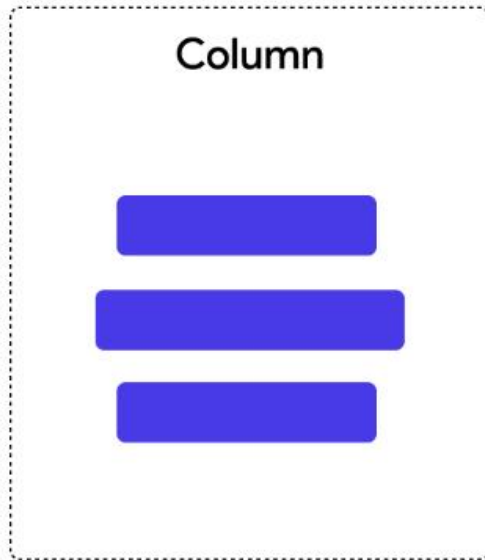
Barista

OutlineInputBorder

3. Row and Column: Row and Column is widget used to display the children in vertical or horizontal position. It takes the children contain an array of List<widget>. The children take up the height or the width of the screen where each widget can be embedded with an Expandedwidget to fill the available space.

```
Row(  
  mainAxisAlignment: MainAxisAlignment.spaceBetween,  
  children: <Widget>[  
    Container(  
      width: 100,  
      height: 100,  
      color: Colors.red,  
    ),  
    Container(  
      width: 100,  
      height: 100,  
      color: Colors.blue,  
    ),  
    Container(  
      width: 100,  
      height: 100,  
      color: Colors.green,  
    ),  
  ],  
)
```

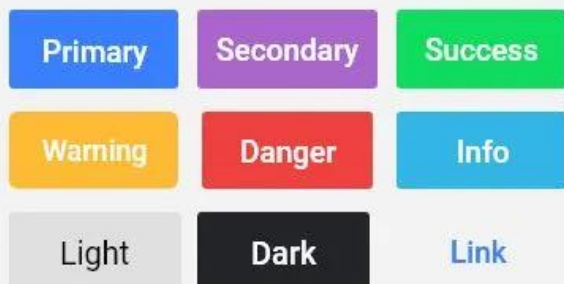
```
Column(  
  mainAxisAlignment: MainAxisAlignment.center,  
  children: <Widget>[  
    Container(  
      width: 100,  
      height: 100,  
      color: Colors.red,  
    ),  
    Container(  
      width: 100,  
      height: 100,  
      color: Colors.blue,  
    ),  
    Container(  
      width: 100,  
      height: 100,  
      color: Colors.green,  
    ),  
  ],  
)
```



4. Button: Flutter has variety of buttons to choose from such as TextButton, ElevatedButton, OutlineButton, TextButton, FloatingActionButton and IconButton. Buttons are the Flutter widget that trigger an event such as making choice or takin action.

```
ElevatedButton(  
  
  onPressed: () {  
  
    print('Button pressed');  
  
  },  
  
  child: Text(  
  
    'Click Me',  
  
    style: TextStyle(  
  
      color: Colors.white,  
  
      fontSize: 20  
  
    ),  
  
  ),  
  
  color: Colors.red,  
)
```

GW Buttons



5. Image: This is used to display images in different formats such as JPEG, PNG, GIF, and BMP.

The image to be shown can come from different source using the following:

- `Image.asset()` – retrieve image from `AssetBundle` class.

Sample:

```
Image.network(  
  'https://picsum.photos/200',  
  fit: BoxFit.cover,  
)
```

6. Icon: The icon widget is drawn with a glyph from a font described in `IconData`. Flutter has the full list of icons available from the font `MaterialIcons`.
`Icon (`

```
Icons.add,
```

```
size: 40,
```

```
color: Colors.red,  
)
```



Icon

A Material Design icon.



Image

A widget that displays an image.

7. TextField: Is a widget that allows users to collect inputs .

```
TextField(  
  
  decoration: InputDecoration(  
  
    hintText: 'Enter your name',  
  
    border: OutlineInputBorder(  
  
      borderRadius: BorderRadius.circular(10),  
  
    ),  
  
    ),  
  
    onChanged: (value) {  
  
      print(value);  
  
    },  
  
  )
```

8. ListView: Is the most used scrolling widget which is used for creating scrollable lists. It has children that display one after another in scroll direction.

```
ListView(  
  children: [  
    Text("Item 1"),  
    Text("Item 2"),  
    Text("Item 3"),  
  ],  
)
```


9. Card: Is like a container widget but you have various properties such as elevation, shape, and border radius. Card widget is graphical user interface that provides a container for other widgets such text, images, and other content used to organize information.

```
Card(  
  
  child: Column(  
  
    children: [  
  
      Image.network("https://flutter.dev/assets/flutter_lockup_x0002_c13da9c9303e26b8d5fc208d2a1fa20c1ef47eb021ecadf27046dea04c0cebf6.png"),  
  
      Text("Hello, Card!"),  
  
    ],  
  
  ),  
)
```

GW Cards

Basic cards



Fruit Salad



Some quick example text to build on the card

Recepie

Add to cart



Recipies



Some quick example text to build on the card

Read more >>



Acer

Some quick example text to build on the card

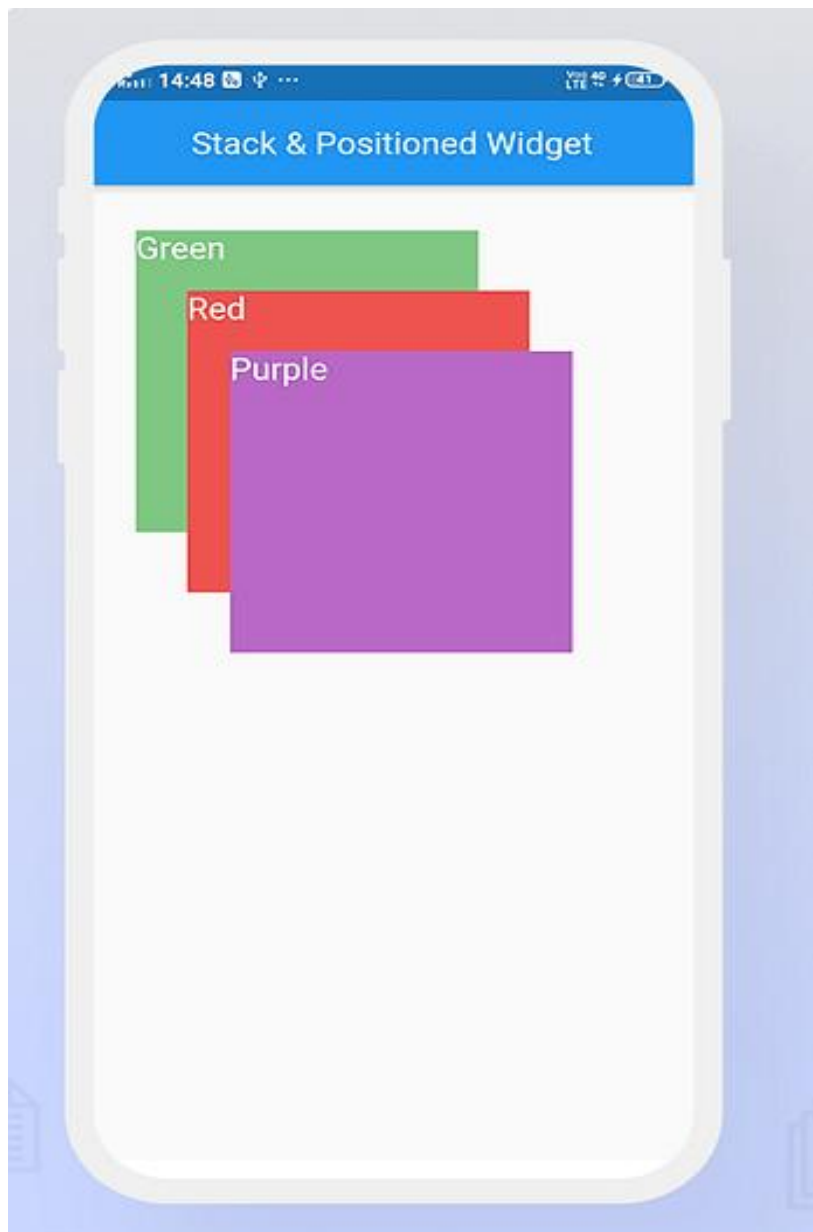
Add to cart

10. Stack: It used to arrange widgets on top of one another and are often overlap the layer elements to create visually appealing and complex user interfaces.

```
Stack(  
  
  alignment: Alignment.center,  
  
  children: [  
  
    Container(  
  
      width: 100.0,  
  
      height: 100.0,  
  
      color: Colors.red,  
  
    ),  
  
    Container(  

```

```
      width: 80.0,  
  
      height: 80.0,  
  
      color: Colors.yellow,  
  
    ),  
  
    Container(  
  
      width: 60.0,  
  
      height: 60.0,  
  
      color: Colors.green,  
  
    ),  
  
  ],  
)
```



11. Wrap: It allows multiple widgets to be arranged in a grid-like layout wrapping the widgets to the next line whenever the available space is exhausted in the main axis. The main axis can be either horizontal or vertical depending on the orientation property. The Wrap widget is useful for creating flexible and dynamic layouts especially in a situation where the number of widgets is unknown or can change dynamically

```
Wrap(  
  spacing: 8.0,  
  runSpacing: 4.0,  
  children: [  
    Container(  
      width: 50.0,  
      height: 50.0,  
      color: Colors.red,  
    ),  
    Container(  
      width: 50.0,  
      height: 50.0,  
      color: Colors.yellow,  
    ),  
    Container(  
      width: 50.0,  
      height: 50.0,  
      color: Colors.green,  
    ),  
    Container(  
      width: 50.0,  
      height: 50.0,  
      color: Colors.blue,  
    ),  
  ],  
)
```



ROW



WRAP





*Thank You
& God bless!*