SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A++' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER APPLICATIONS

23CAT607- CROSS-PLATFORM APP DEVELOPMENT

I YEAR II SEM

UNIT 3 – INTRODUCTION TO LAYOUTS

TOPIC 2 – Single Child Widgets, Multiple Child Widgets

SINGLE-CHILD WIDGETS FOR RESPONSIVE LAYOUT

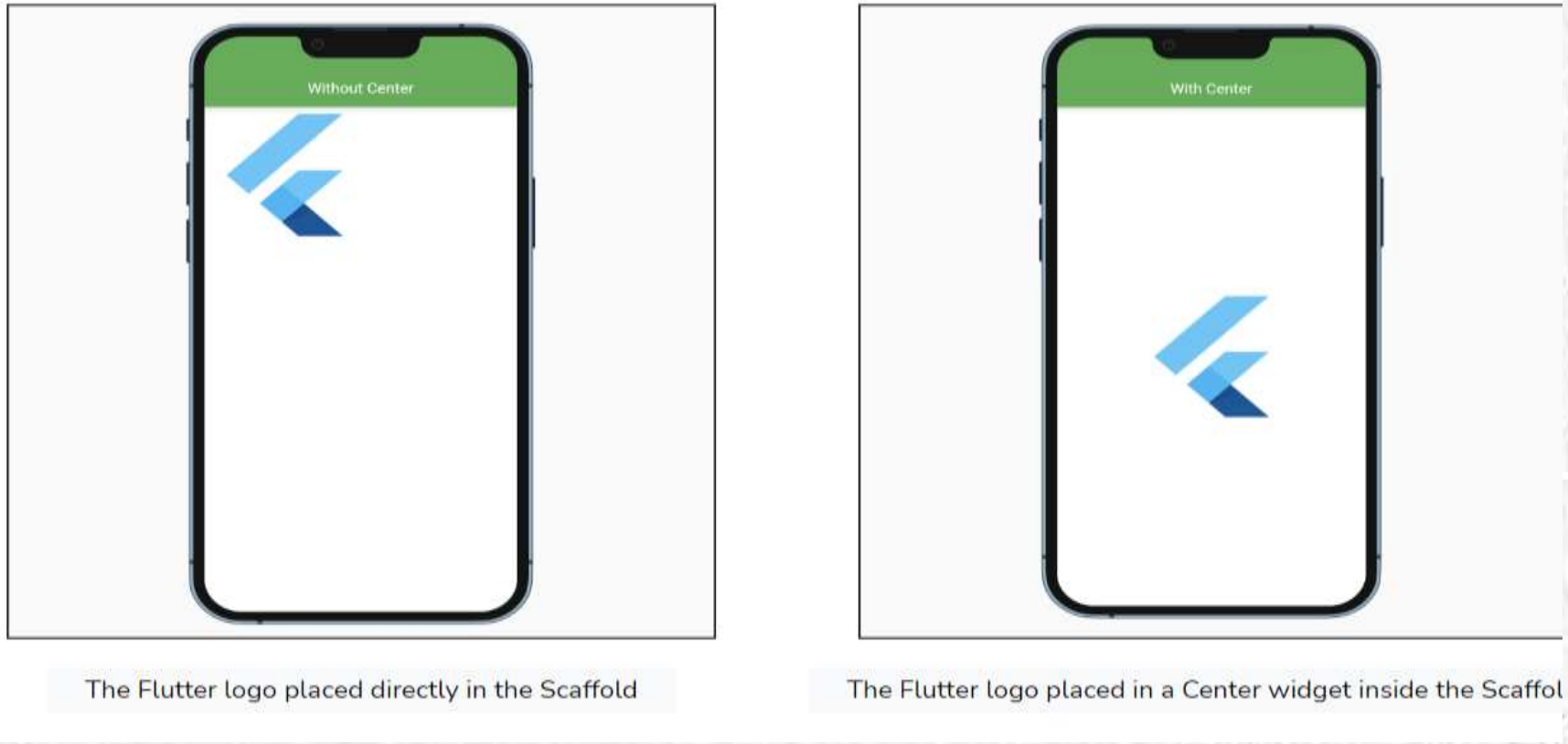
A **single-child layout widget** allows us to change the position or the size of its child. For instance, we can use Center to change the position of a widget to be centered in its parent.

```
Scaffold(
  appBar: AppBar(
    title: const Text('Without Center'),
  ),
  body: const FlutterLogo(size: 200.0),
),

Scaffold(
  appBar: AppBar(
    title: const Text('With Center'),
  ),
  body: const Center(
    child: FlutterLogo(size: 200.0),
  ),
),
```

Two Scaffold widgets with and without a Center Child

In the code above, there are two versions of a Scaffold containing just the Flutter logo. In the first version, the logo is placed in the default position. In the second version on line 12, the logo is wrapped in a Center widget, a single-child layout widget, which changes the position of the logo to be centered in the Scaffold. The result is shown in the images below.



The Flutter logo placed directly in the Scaffold

The Flutter logo placed in a Center widget inside the Scaffol

In the first version of the Scaffold on the left side, the logo is placed in the default position—top left. In the second version on the right side, the logo is wrapped in the Center widget.

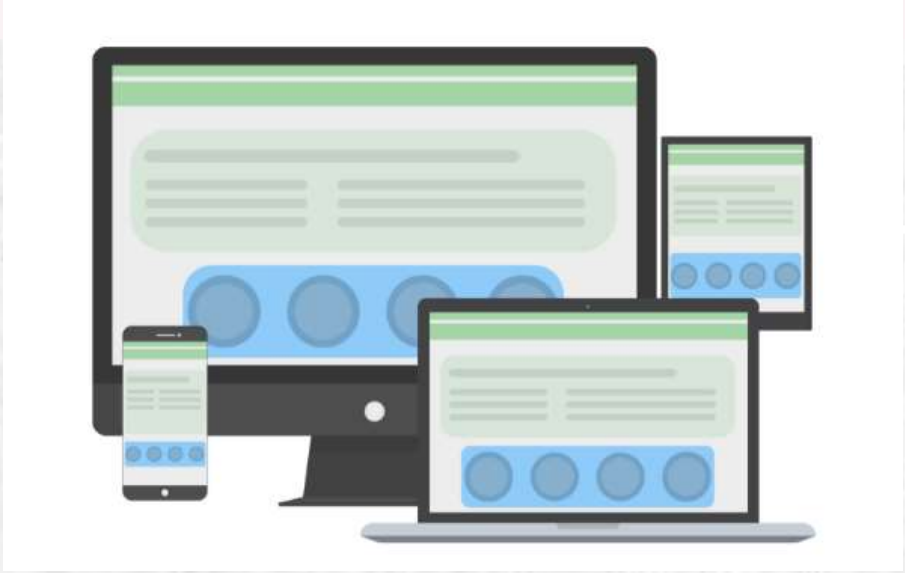
LAYOUT WIDGETS FOR RESPONSIVE DESIGN

Flutter application without knowing we could exploit them to make our application responsive.
The single-child widgets we'll see in this chapter are as follows:

- Align
- AspectRatio
- ConstrainedBox
- Expanded
- Flexible

MULTI-CHILD WIDGETS FOR RESPONSIVE LAYOUT

It’s very common for application layouts to combine different UI components. Think of lists, grids, and custom widgets displayed in a column or row. The Flutter widgets that replicate these layouts must handle multiple children widgets simultaneously. Some of them can be exploited to work well on both small and large screens.



The multi-child widgets we'll see in this chapter are as follows:

1. Column
2. Row
3. ListView
4. GridView
5. Stack
6. Table
7. Wrap

SINGLE CHILD SCROLL VIEW CLASS

This widget is useful when you have a single box that will normally be entirely visible, for example a clock face in a time picker, but you need to make sure it can be scrolled if the container gets too small in one axis (the scroll direction).

In that case, you might pair the SingleChildScrollView with a ListBody child.

Persisting the scroll position during a session

Scroll views attempt to persist their scroll position using PageStorage. This can be disabled by setting ScrollController.keepScrollOffset to false on the controller. If it is enabled, using a PageStorageKey for the key of this widget is recommended to help disambiguate different scroll views from each other.

- ListView, which handles multiple children in a scrolling list.
- GridView, which handles multiple children in a scrolling grid.
- PageView, for a scrollable that works page by page.
- Scrollable, which handles arbitrary scrolling effects

Inheritance

- Object
- DiagnosticableTree
- Widget
- StatelessWidget
- SingleChildScrollView

PROPERTIES

child → Widget?

- The widget that scrolls.
- final

clipBehavior → Clip

- The content will be clipped (or not) according to this option.
- final

controller → ScrollController?

- An object that can be used to control the position to which this scroll view is scrolled.
- final

dragStartBehavior → DragStartBehavior

- Determines the way that drag start behavior is handled.
- final

hashCode → int

- The hash code for this object.
- no setterinherited

key → Key?

- Controls how one widget replaces another widget in the tree.
- finalinherited

keyboardDismissBehavior → ScrollViewKeyboardDismissBehavior

- ScrollViewKeyboardDismissBehavior the defines how this ScrollView will dismiss the keyboard automatically.
- final

padding → EdgeInsetsGeometry?

- The amount of space by which to inset the child.
- final

physics → ScrollPhysics?

- How the scroll view should respond to user input.
- final

primary → bool?

- Whether this is the primary scroll view associated with the parent PrimaryScrollController.
- final

restorationId → String?

- Restoration ID to save and restore the scroll offset of the scrollable.
- final

reverse → bool

- Whether the scroll view scrolls in the reading direction.
- final

runtimeType → Type

- A representation of the runtime type of the object.
- no setterinherited

scrollDirection → Axis

- The Axis along which the scroll view's offset increases.
- final

METHODS

build(BuildContext context) → *Widget*

- Describes the part of the user interface represented by this widget.
- override

createElement() → *StatelessElement*

- Creates a StatelessElement to manage this widget's location in the tree.
- inherited

debugDescribeChildren() → *List*<*DiagnosticsNode*>

- Returns a list of DiagnosticsNode objects describing this node's children.
- inherited

debugFillProperties(DiagnosticPropertiesBuilder properties) → *void*

- Add additional properties associated with the node.
- inherited

noSuchMethod(Invocation invocation) → *dynamic*

- Invoked when a nonexistent method or property is accessed.
- inherited

toDiagnosticsNode({String? name, DiagnosticsTreeStyle? style}) → *DiagnosticsNode*

- Returns a debug representation of the object that is used by debugging tools and by DiagnosticsNode.toStringDeep.
- inherited

toString({DiagnosticLevel minLevel = DiagnosticLevel.info}) → *String*

- A string representation of this object.
- inherited

toStringDeep({String prefixLineOne = '', String? prefixOtherLines, DiagnosticLevel minLevel = DiagnosticLevel.debug}) → *String*

- Returns a string representation of this node and its descendants.
- inherited

toStringShallow({String joiner = ', ', DiagnosticLevel minLevel = DiagnosticLevel.debug}) → *String*

- Returns a one-line detailed description of the object.
- inherited

toStringShort() → *String*

- A short, textual description of this widget.
- inherited

OPERATORS

operator ==(Object other) → *bool*

- The equality operator.
- inherited

Single-Child Widgets

Single-child widgets are those that can only contain one direct child widget. These are often used to apply some form of transformation or constraint to their single child.

Examples of single-child widgets include Container, Padding, Center, and Align.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Single Child Widget Example'),
        ),
```

```
body: Center(
        child: Container(
          width: 200,
          height: 200,
          color: Colors.blue,
          child: Center(
            child: Text(
              'Hello, Flutter!',
              style: TextStyle(color: Colors.white),
            ),
          ),
        ),
      ),
    ),
  ); } }
```

In this example,

Container is a single-child widget that contains a Center widget, which in turn contains a Text widget.

Multi-Child Widgets

Multi-child widgets can contain multiple direct children. These are typically used for layout purposes and include widgets such as **Column, Row, Stack, and ListView.**

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Multi-Child Widget Example'),
        ),
```

```
body: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Container(
            width: 100,
            height: 100,
            color: Colors.red,
            child: Center(
              child: Text(
                'One',
                style: TextStyle(color: Colors.white),
              ),
            ),
          ),
```

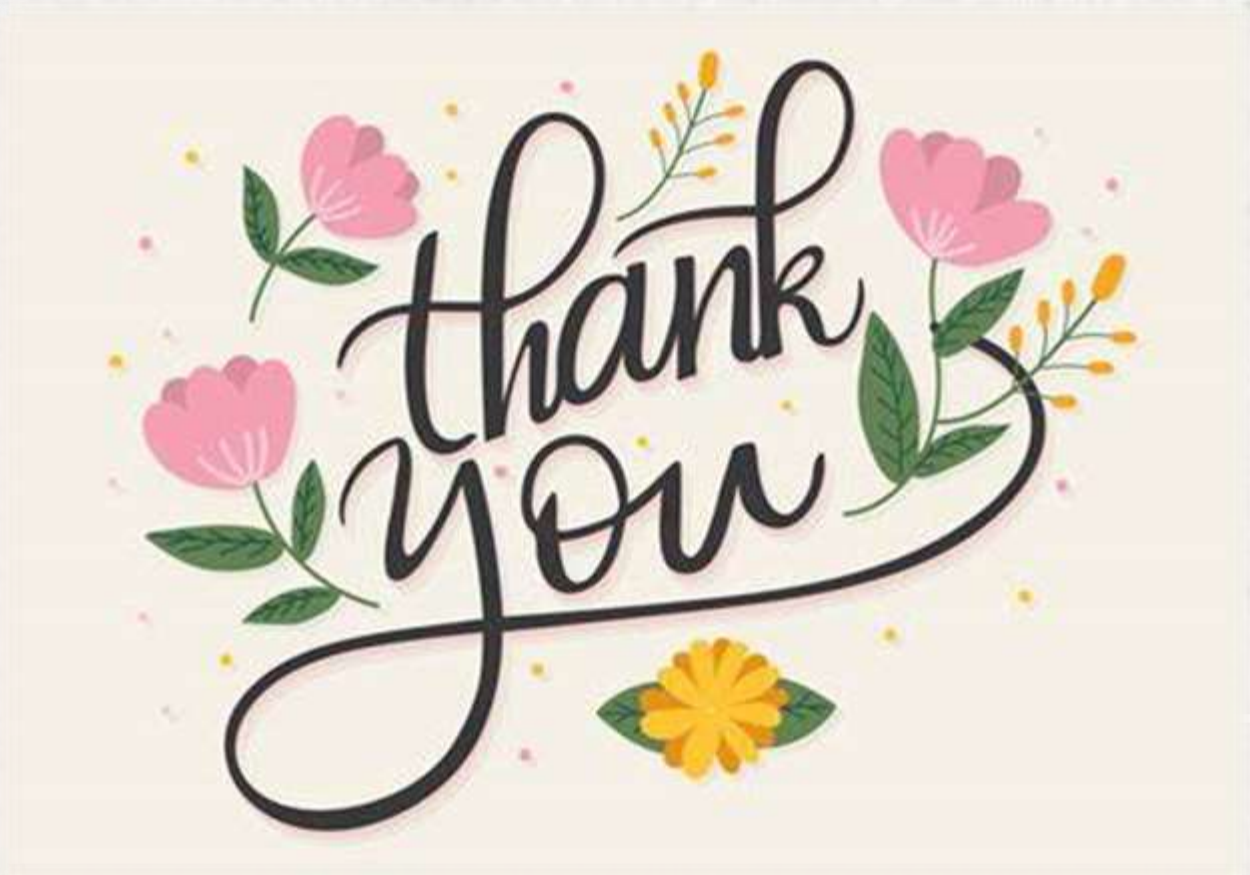
```
Container(
            width: 100,
            height: 100,
            color: Colors.green,
            child: Center(
              child: Text(
                'Two',
                style: TextStyle(color: Colors.white),
              ),
            ),
          ),
```

```
Container(
            width: 100,
            height: 100,
            color: Colors.blue,
            child: Center(
              child: Text(
                'Three',
                style: TextStyle(color: Colors.white),
              ),
            ),
          ),
        ],
      ),
    ),
  );
 }
}
```

In this example, Column is a multi-child widget that contains three Container widgets. Each container displays a different text and has a different background color.

KEY DIFFERENCES

Child Capacity:	Single-child widgets can have only one direct child.	Multi-child widgets can have multiple direct children.
Use Cases:	Single-child widgets are used when you need to apply a constraint, alignment, padding, or other modification to a single widget.	Multi-child widgets are used for layouts that require multiple widgets to be displayed in a specific order or arrangement.
Examples:	Single-child widgets: Container, Padding, Center, Align.	Multi-child widgets: Column, Row, Stack, ListView.


thank
you