Flutter Lecture Finals Compilation

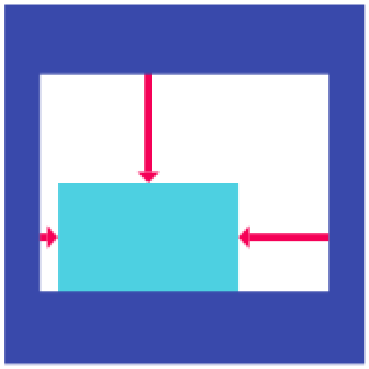
Flutter Layouts

- Widgets are classes used to build UIs.
- Widgets are used for both layout and UI elements.
- Compose simple widgets to build complex widgets.

Flutter Layout Categories

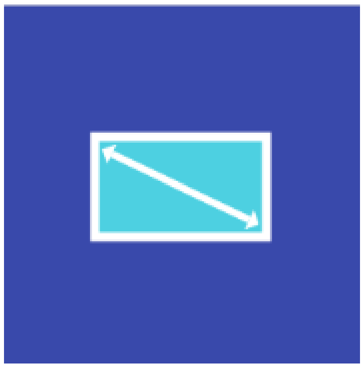
- Single-child layout widgets
- Multi-child layout widgets
- Sliver widgets

Single Child Layout Widgets



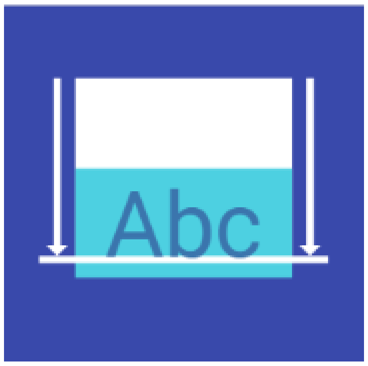
Align

A widget that aligns its child within itself and optionally sizes itself based on the child's size.



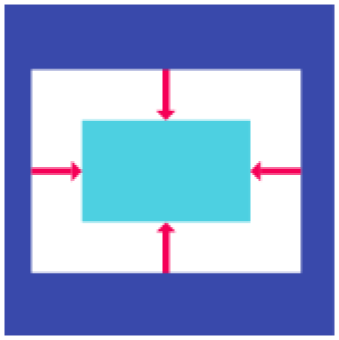
AspectRatio

A widget that attempts to size the child to a specific aspect ratio.



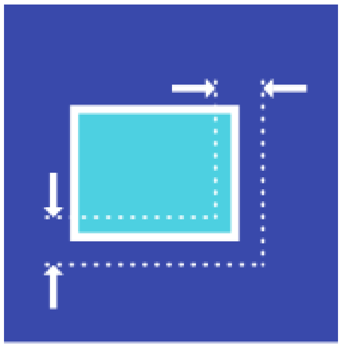
Baseline

A widget that positions its child according to the child's baseline.



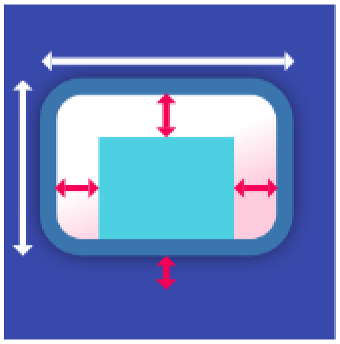
Center

A widget that centers its child within itself.



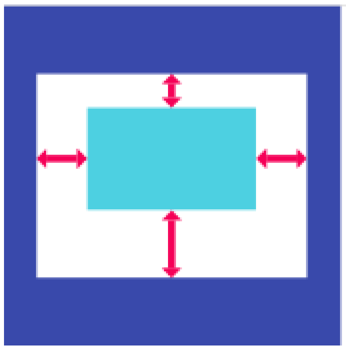
ConstrainedBox

A widget that imposes additional constraints on its child.



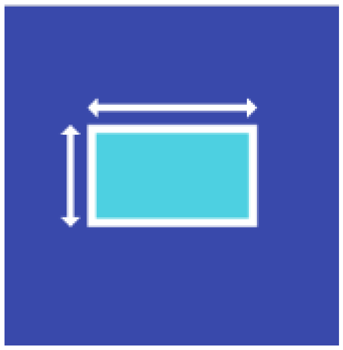
Container

A convenience widget that combines common painting, positioning, and sizing widgets.



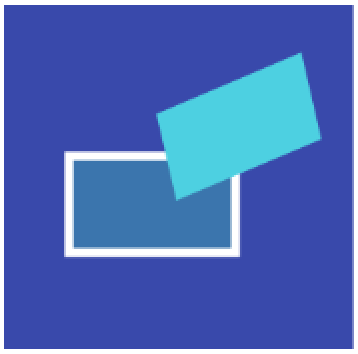
Padding

A widget that insets its child by the given padding.



SizedBox

A box with a specified size. If given a child, this widget forces its child to have a specific width and/or height.



Transform

A widget that applies a transformation before painting its child.

CustomSingleChildLayout

A widget that defers the layout of its single child to a delegate.

Expanded

A widget that expands a child of a Row, Column, or Flex.

FittedBox

Scales and positions its child within itself according to fit.

FractionallySizedBox

A widget that sizes its child to a fraction of the total available space. For more details about the layout algorithm, see RenderFractionallySizedOverflowBox.

IntrinsicHeight

A widget that sizes its child to the child's intrinsic height.

IntrinsicWidth

A widget that sizes its child to the child's intrinsic width.

LimitedBox

A box that limits its size only when it's unconstrained.

Offstage

A widget that lays the child out as if it was in the tree, but without painting anything, without making the child available for hit...

OverflowBox

A widget that imposes different constraints on its child than it gets from its parent, possibly allowing the child to overflow the parent.

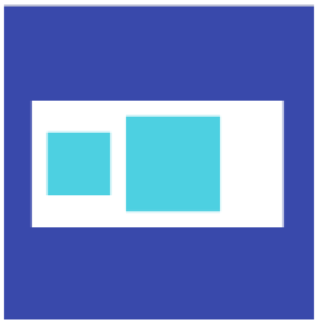
SizedOverflowBox

A widget that is a specific size but passes its original constraints through to its child, which will probably overflow.

Multi-child layout widgets

Column

Layout a list of child widgets in the vertical direction,

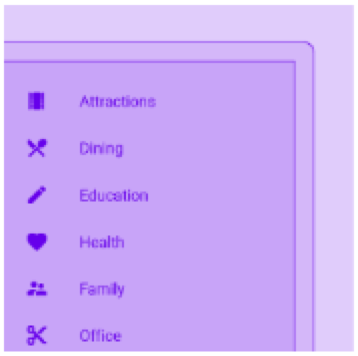


Row

Layout a list Of Child widgets in the horizontal direction.

GridView

A grid list consists of a repeated pattern of cells arrayed in a vertical and horizontal layout. The GridView Widget implements this component.



ListView

A scrollable, linear list of widgets. ListView is the most commonly used scrolling widget. It displays its children one after another in the scroll direction.

Stack

This class is useful if you want to overlap several children in a simple way.

Table

A widget that uses the table layout algorithm for its children.

CustomMultiChildLayout

A widget that uses a delegate to size and position multiple children.

Flow

A widget that implements the flow layout algorithm.

IndexedStack

A Stack that shows a single child from a list of children.

LayoutBuilder

Builds a widget tree that can depend on the parent widget's size.

ListBody

A widget that arranges its children sequentially along a given axis, forcing them to the dimension of the parent in the other axis.

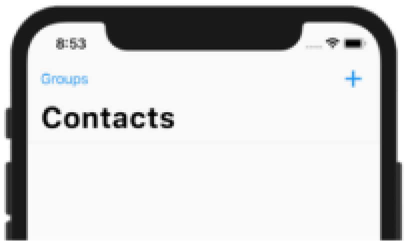
Table

A widget that uses the table layout algorithm for its children.

Wrap

A widget that displays its children in multiple horizontal or vertical runs.

Sliver Layouts

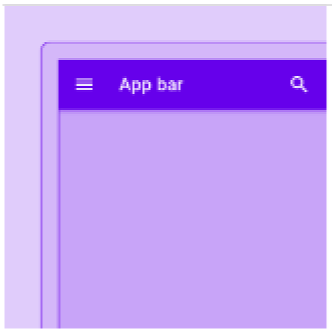


CupertinoSliverNavigationBar

An iOS-styled navigation bar with iOS-I I-style large titles using slivers.

CustomScrollView

A ScrollView that creates custom scroll effects using slivers.



SliverAppBar

A material design app bar that integrates with a CustomScrollView.

SliverChildBuilderDelegate

A delegate that supplies children for slivers using a builder callback.

SliverChildListDelegate

A delegate that supplies children for slivers using an explicit list.

SliverFixedExtentList

A sliver that places multiple box children with the same main axis extent in a linear array.

SliverGrid

A sliver that places multiple box children in a two dimensional arrangement.

SliverList

A sliver that places multiple box children in a linear array along the main axis.

SliverPadding

A sliver that applies padding on each side of another sliver.

SliverPersistentHeader

A sliver whose size varies when the sliver is scrolled to the edge of the viewport opposite the sliver's GrowthDirection.

SliverToBoxAdapter

A sliver that contains a single box widget.

Standard widgets

Container

Adds padding, margins, borders, background color, or other decorations to a widget.

GridView

Lays widgets out as a scrollable grid.

ListView

Lays widgets out as a scrollable list.

Stack

Overlaps a widget on top of another.

Material widgets

Card

Organizes related info into a box with rounded corners and a drop shadow.

ListTile

Organizes up to 3 lines of text, and optional leading and trailing icons, into a row.

Container

use Containers to separate widgets using padding, or to add borders or margins. You can change the device’s background by placing the entire layout into a Container and changing its background color or image.

Uses

- Add padding, margins, borders
- Change background color or image
- Contains a single child widget, but that child can be a Row, Column, or even the root of a widget tree
- Diagram showing: margin, border, padding, and content

The Material Library

- The Material library implements widgets that follow Material Design principles.
- When designing your UI, you can exclusively use widgets from the standard widgets library, or you can use widgets from the Material library.
- You can mix widgets from both libraries, you can customize existing widgets, or you can build your own set of custom widgets.

Material Components Widgets

Visual, behavioral, and motion-rich widgets implementing the Material Design guidelines.

1. App structure and navigation
2. Buttons
3. Input and selections
4. Dialogs, alerts, and panels
5. Information displays
6. Layout

Lifecycle of a Stateful Widget

init State ()

The init State gets triggered implicitly as soon as the State initially get initialized. It is used when we want something to happen the moment our stateful widget is created.

build ()

The build method gets triggered when the widgets are constructed and appear on the screen. It is used when we want something to happen every single time when our stateful widget gets rebuild.

deactivate()

Deactivate method gets called when the stateful widget gets destroyed (just like destructor). It is used when we want something to happen just before our stateful widget gets destroyed.

Lifecycle of Flutter App

Simplified Steps

- createState()
- mounted == true
- initState()
- didChangeDependencies()
- build()
- didUpdateWidget()
- setState()
- deactivate()
- dispose()
- mounted == false

Why Are StatefulWidget and State Separate Classes?

State isn't thrown away, it can constantly be rebuilding it's Widget in response to data changes, and when required.

createState()

When Flutter is instructed to build a StatefulWidget, it immediately calls createState(). This method must exist.

mounted == true

- When createState creates the state class, a buildContext is assigned to that state.
- All widgets have a bool this.mounted property. It is turns true when the buildContext is assigned.
- It is an error to call setState when a widget is unmounted.
- This property is useful when a method on your state calls setState() but it isn't clear when or how often that method will be called.
- Its being called in response to a stream updating.
- You can use if (mounted) {... to make sure the State exists before calling setState().

initState()

- This is the first method called when the widget is created (after the class constructor, of course.)
- initState is called once and only once. It must also call super.initState().
- This @override method is the best time to:
- Initialize data that relies on the specific BuildContext for the created instance of the widget.
- Initialize properties that rely on this widgets 'parent' in the tree.
- Subscribe to Streams, ChangeNotifiers, or any other object that could change the data on this widget.

didChangeDependencies()

- The didChangeDependencies method is called immediately after initState on the first time the widget is built.
- It will also be called whenever an object that this widget depends on data from is called. For example, if it relies on an InheritedWidget, which updates.
- build is always called after didChangeDependencies is called, so this is rarely needed.
- This method is the first change you have to call BuildContext.inheritFromWidgetOfExactType.
- This essentially would make this State 'listen' to changes on a Widget it's inheriting data from.
- The docs also suggest that it could be useful if you need to do network calls (or any other expensive action) when an InheritedWidget updates.

build()

- This method is called often (think fps + render). It is a required, @override and must return a Widget.
- Remember that in Flutter all gui is a widget with a child or children, even 'Padding', 'Center'.

didUpdateWidget(Widget oldWidget)

- didUpdateWidget() is called if the parent widget changes and has to rebuild this widget (because it needs to give it different data), but it's being rebuilt with the same runtimeType, then this method is called.
- This is because Flutter is re-using the state, which is long lived. In this case, required is to initialize some data again, as one would in initState().
- If the state's build() method relies on a Stream or other object that can change, unsubscribe from the old object and re-subscribe to the new instance in didUpdateWidget().
- Tip: This method is basically the replacement for 'initState()' if it is expected the Widget associated with the widget's state needs to be rebuilt!
- Flutter always called build() after this, so any subsequent
- further calls to setState is redundant.

setState()

- The 'setState()' method is called often from the Flutter framework itself and from the developer.
- It is used to notify the framework that "data has changed", and the widget at this build context should be rebuilt.
- setState() takes a callback which cannot be async. It is for this reason it can be called often as required, becauserepainting is cheap.

deactivate()

- This is rarely used.
- deactivate()' is called when State is removed from the tree, but it might be reinserted before the current frame change is finished. This method exists basically because State objects can be moved from one point in a tree to another.

dispose()

- dispose() is called when the State object is removed, which is permanent.
- This method is where to unsubscribe and cancel all animations, streams, etc.

mounted == false

- The state object can never remount, and an error is thrown is setState() is called.

Gestures

- Gestures represent Semantic actions (for example, tap, drag, and scale) that are recognized from multiple individual pointer events, potentially even multiple individual pointers.
- Gestures can dispatch multiple events, corresponding to the lifecycle of the gesture (for example, drag start, drag update, and drag end).
- using Material Components, many ofthose widgets already respond to taps or gestures.
- For example, IconButton and TextButton respond to presses (taps), and ListView responds to swipes to trigger scrolling.

GestureDetector

- Flutter provides an excellent support for all type of gestures through its exclusive widget, GestureDetector.
- GestureDetector is a non-visual widget primarily used

for detecting the user’s gesture.
- To identify a gesture targeted on a widget, the widget can be placed inside GestureDetector widget.
- GestureDetector will capture the gesture and dispatch multiple events based on the gesture.

2 Separate Layers for the Gesture System

1. The first layer has raw pointer events that describe the location and movement of pointers (for example, touches, mice, and styli) across the screen.
2. The second layer has gestures that describe semantic actions that consist of one or more pointer movements.

List of Gestures

Tap

onTapDown

A pointer that might cause a tap has contacted the screen at a particular location.

onTapUp

A pointer that will trigger a tap has stopped contacting the screen at a particular location.

onTap

The pointer that previously triggered the onTapDown has also triggered onTapUp which ends up causing a tap.

onTapCancel

The pointer that previously triggered the onTapDown will not end up causing a tap.

Double tap

onDoubleTap

The user has tapped the screen at the same location twice in quick succession.

Long press

onLongPress

A pointer has remained in contact with the screen at the same location for a long period of time.

Vertical drag

onVerticalDragStart

A pointer has contacted the screen and might begin to move vertically.

onVerticalDragUpdate

A pointer that is in contact with the screen and moving vertically has moved in the vertical direction.

onVerticalDragEnd

A pointer that was previously in contact with the screen and moving vertically is no longer in contact with the screen and was moving at a specific velocity when it stopped contacting the screen.

Horizontal drag

onHorizontalDragStart

A pointer has contacted the screen and might begin to move horizontally.

onHorizontalDragUpdate

A pointer that is in contact with the screen and moving horizontally has moved in the horizontal direction.

onHorizontalDragEnd

A pointer that was previously in contact with the screen and moving horizontally is no longer in contact with the screen and was moving at a specific velocity when it stopped contacting the screen.

Pan

onPanStart

A pointer has contacted the screen and might begin to move horizontally or vertically. This callback causes a crash if onHorizontalDragStart or onVerticalDragStart is set.

onPanUpdate

A pointer that is in contact with the screen and is moving in the vertical or horizontal direction. This callback causes a crash if onHorizontalDragUpdate or onVerticalDragUpdate is set.

onPanEnd

A pointer that was previously in contact with screen is no longer in contact with the screen and is moving at a specific velocity when it stopped contacting the screen. This callback causes a crash if onHorizontalDragEnd or onVerticalDragEnd is set.

Pointers

Pointers represent raw data about the user’s interaction with the device’s screen. There are four types of pointer events:

PointerDownEvent

The pointer has contacted the screen at a particular location.

PointerMoveEvent

The pointer has moved from one location on the screen to another.

PointerUpEvent

The pointer has stopped contacting the screen.

PointerCancelEvent

Input from this pointer is no longer directed towards this app.

- On pointer down, the framework does a hit test on your app to determine which widget exists at the location where the pointer contacted the screen.
- The pointer down event (and subsequent events for that pointer) are then dispatched to the innermost widget found by the hit test.
- From there, the events bubble up the tree and are dispatched to all the widgets on the path from the innermost widget to the root of the tree. There is no mechanism for canceling or stopping pointer events from being dispatched further.
- To listen to pointer events directly from the widgets layer, use a Listener widget.

Gesture disambiguation

- At a given location on screen, there might be multiple gesture detectors.
- All of these gesture detectors listen to the stream of pointer events as they flow past and attempt to recognize specific gestures.
- The GestureDetector widget decides which gestures to attempt to recognize based on which of its callbacks are non-null.
- When there is more than one gesture recognizer for a given pointer on the screen, the framework disambiguates which gesture the user intends by having each recognizer join the gesture arena.

When there is more than one gesture recognizer for a given pointer on the screen, the framework disambiguates which gesture the user intends by having each recognizer join the gesture arena. The gesture arena determines which gesture wins using the following rules:

- At any time, a recognizer can declare defeat and leave the arena. If there's only one recognizer left in the arena, that recognizer is the winner.
- At any time, a recognizer can declare victory, which causes it to win and all the remaining recognizers to lose.