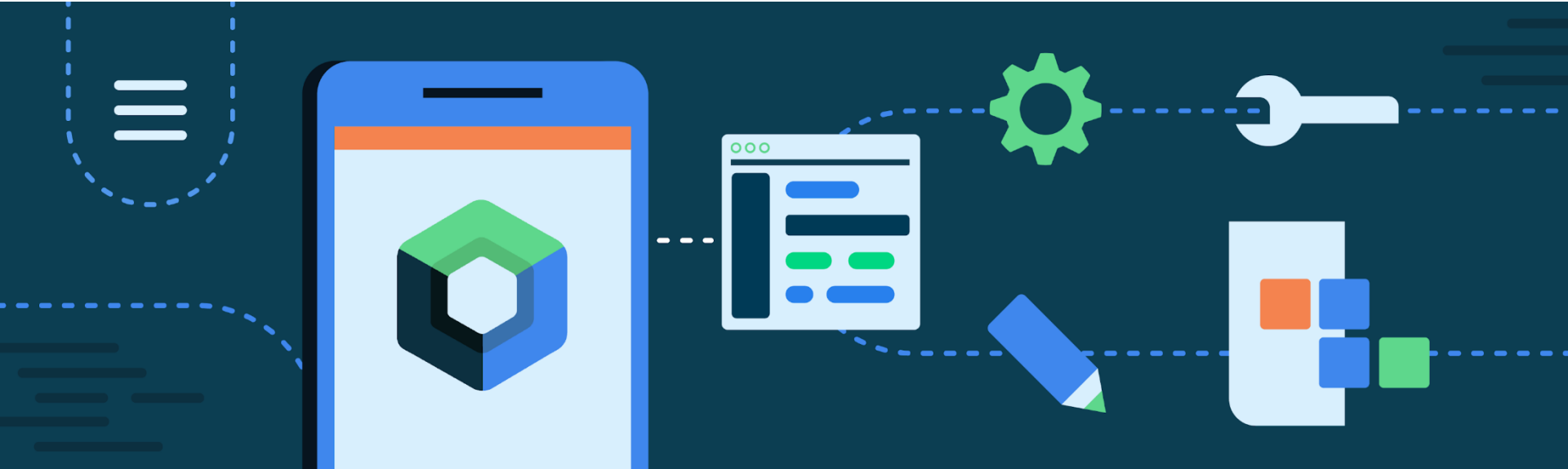


CMPS 312



Flutter Fundamentals

Dr. Abdelkarim Erradi
CSE@QU

Outline

1. Flutter Key Concepts
2. Widgets (UI Components)
3. Layouts
4. State

Flutter Key Concepts



Declarative UI is a major trend

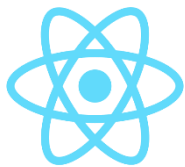
- Describe WHAT to see NOT HOW



Flutter: Google's UI toolkit for building natively compiled applications for mobile, web and desktop from a single codebase



SwiftUI: Apple's declarative framework for creating apps that run on iOS



React: A JavaScript library for building user interfaces

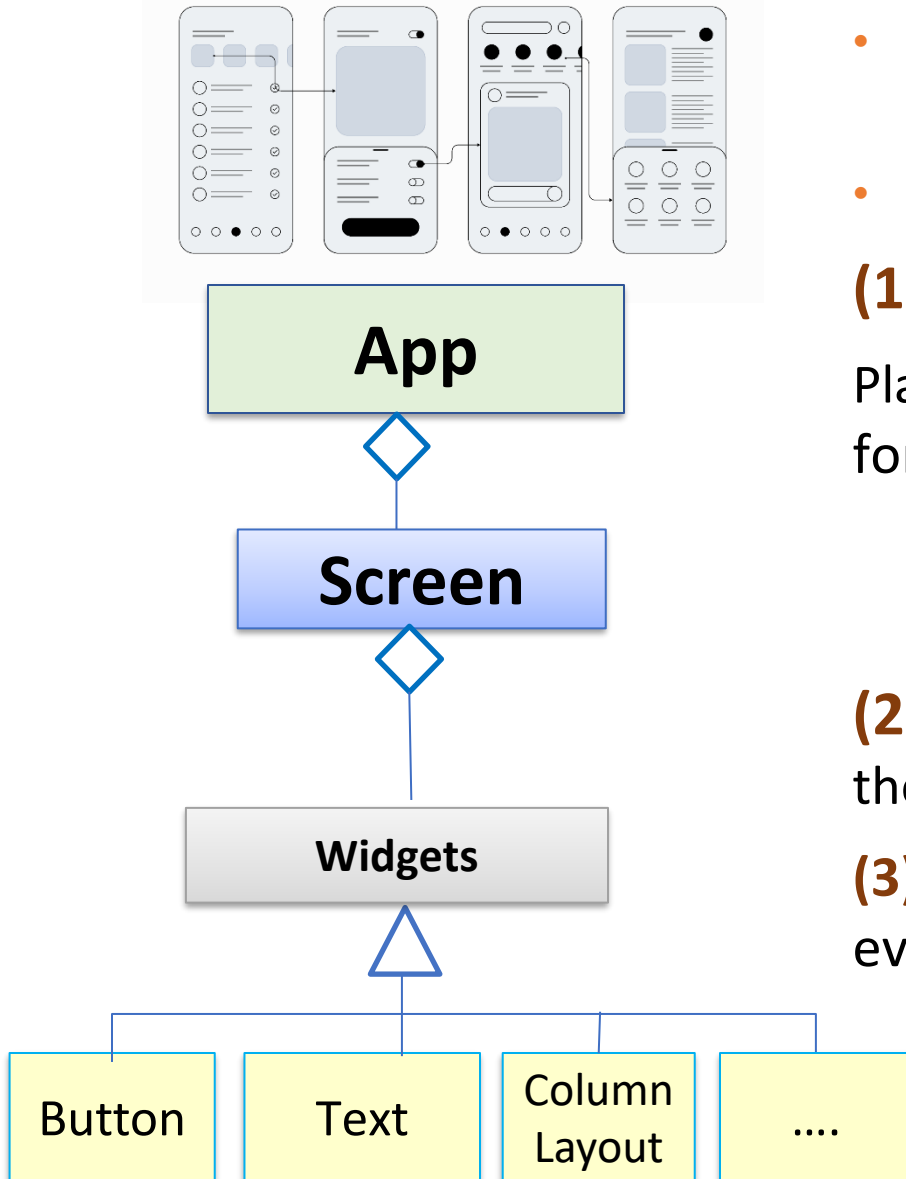


Jetpack Compose: a **toolkit** for building native Android UI

Flutter

- Flutter is a **UI framework** (i.e., Widgets, Rendering Engine and DevTools) to simplify UI development that follow **best practices**
- **A declarative component-based programming model**
 - UI is built using composable widgets
 - Each widget defines a piece of the app's UI programmatically by **describing WHAT to see** (layout/ look and feel) **NOT HOW**
 - Compiler takes care of the HOW and constructs UI elements
 - As state changes the UI automatically updates (Reactive UI) (without imperatively mutating UI components)
- Inspired by/similar to other declarative UI frameworks such as React and Jetpack Compose

Declarative UI Programming Model



- App is composed of one or more **screens** (also called pages)
- A **screen** has:

(1) **Widgets** (UI Components)

Placed in a **Layout** that acts as a **container** for UI Components

- Layout decides the size and position of widgets placed in it

(2) State objects that provides the data to the UI

(3) **Event Handlers** to respond to the UI events

- Widgets **raise Events** when the user interacts with them (such as a Pressed event is raised when a button is pressed)



How to define a piece of UI?

- UI is **composed** of small reusable **components** called widgets
- **Widget:** a class that extends [StatelessWidget](#) or [StatefulWidget](#) depending on whether it manages internal state
 - Each component renders a portion of the UI, transforming the app's data (state) into visual elements
- **UI = f(state) : UI is a visual representation of state** (e.g., display a tweet and associated comments)



- **State-Driven UI Updates**
 - State changes trigger automatic update of the UI

Stateless Widget

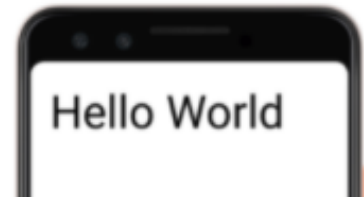
String



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



stdout



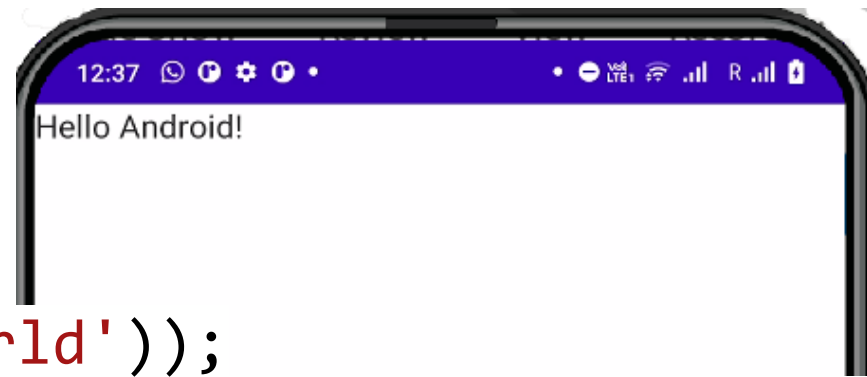
```
class Greeting extends StatelessWidget {
    final String name;
    const Greeting(this.name);

    @override
    Widget build(BuildContext context) {
        return Text('Hello, $name');
    }
}
```

Greeting class uses the input data to render a Text widget on the screen

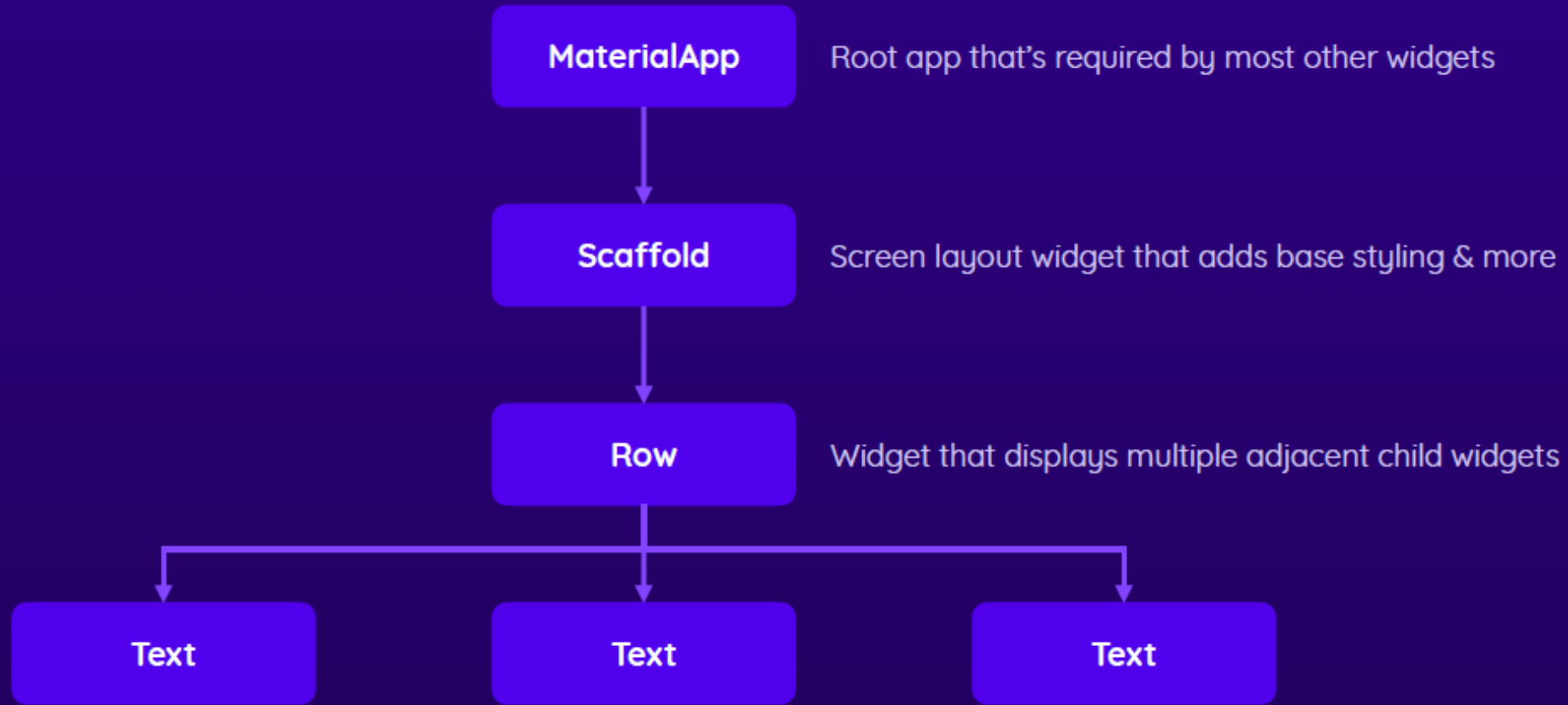
App Entry Point

- The `main()` function is the app entry point
 - Inside it you call the `runApp()` function to launch the app and display the UI on the screen
 - `runApp()` takes a widget (the root widget) and displays the app UI
 - The root widget can be anything, but typically it's a `MaterialApp`, which is a pre-built app structure, including theming, navigation, and more



```
void main() {  
    runApp(const Greeting('World'));  
}
```

Widget Tree



Widgets that display some text on the screen

BuildContext

- **BuildContext** represents the location of a widget within the widget tree, serving as a link between the widget and its surrounding environment. It plays a critical role in giving the widget access to:
 - **Theme**: used to customize the app's look and feel, such as colors, fonts.
 - **MediaQuery**: provides information about the screen size, device orientation to enable responsive UI that adapt to different screen sizes
 - **Navigator**: used for navigating between screens

BuildContext usage example

```
class Greeting extends StatelessWidget {  
  final String name;  
  
  const Greeting(this.name);  
  
  @override  
  Widget build(BuildContext context) {  
    return Text(  
      'Hello, $name',  
      // Using context to access theme data  
      style: Theme.of(context).textTheme.headlineLarge,  
    );  
  }  
}
```

Stateless vs Stateful widgets

- A stateless widget doesn't hold any state
 - The caller controls and manages the state
- Stateful widget holds **mutable state**, which can be modified using **setState()** to trigger a rebuild

Stateless vs Stateful



Stateless Widgets

Don't manage any internal data

Only update the screen if parent Widgets were updated ("re-rendered")



Should be your default:
Use as often as possible



Stateful Widgets

Do manage internal data ("state")

When state changes, the Widget is re-rendered & the UI is updated



Use whenever you have changing data that should cause UI updates



UI = Composition of Widgets

- The top-level widget describes the UI by calling other widgets and passing them the appropriate data

Your name



Welcome Android!

Change Color (clicked 9 times)

`@Composable`

```
fun WelcomeScreen() {  
    var userName by remember { mutableStateOf( value: "Android") }  
    Column { this: ColumnScope  
        NameEditor(name = userName, nameChange = { newName -> userName = newName })  
        Welcome(userName)  
    }  
}
```

`@Composable`

```
fun NameEditor(name: String, nameChange: (String) -> Unit) {...}
```

`@Composable`

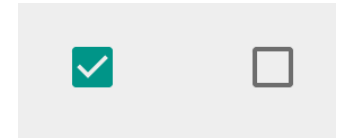
```
fun Welcome(name: String) {...}
```

Widgets

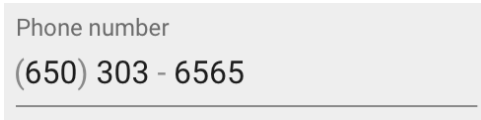
Button



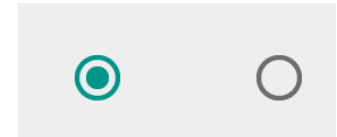
CheckBox



TextField



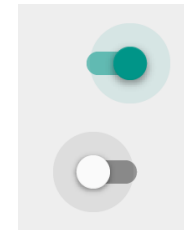
RadioButton



Slider



Switch



**See more details in slides
'05 Widgets-Layouts'**

Full list available at [link](#)

Layouts

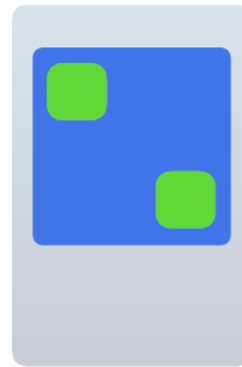
- Use a Layout to size & **position** UI elements on the screen
- **Row** - position elements horizontally
- **Column** - position elements vertically
- **Stack** - stack elements on top of each other
- Many more...



Column



Row



Stack



Other
Layouts

See more details in slides '05 Widgets-Layouts' & this [link](#)

Column and Row

Column() & Row() can be used to place multiple child widgets next to each other



Column()

Main Axis: **Vertical** Axis

Cross Axis: Horizontal Axis



By default, occupies the **entire available height** but **only the width required** by its content (children)



Row()

Main Axis: **Horizontal** Axis

Cross Axis: Vertical Axis




By default, occupies the **entire available width** but **only the height required** by its content (children)

State



<https://developer.android.com/jetpack/compose/state>

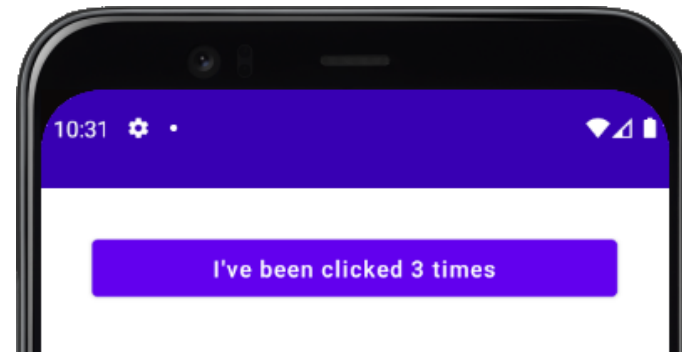
State

- State = any value that can change overtime
- State variables must be declared in class that extends **State** base class
 - They should be changed inside **setState(...)** method that act as **Change Notifiers** to notify Flutter runtime
 -  ○ Any change of a state variable (inside **setState** method) will trigger the **recomposition** of any widgets that **reads** the state variable
=> UI is **auto-updated** to reflect the updated app state
- UI in Flutter is immutable
 - In Flutter you cannot access/update UI elements directly (as done in the imperative approach)
 - The only way to update the UI is by updating the state variable(s) used by the UI elements – this triggers automatic UI update
 - E.g., displayed **counter text** can only be changed by updating the **counter** state variable

Widget Rebuilding

- When the user interacts with the UI, the widgets raises events such as onChanged
 - Those events should notify the app logic, which can then change the app's state
 - When the state changes it causes the widgets build methods to be automatically called again with the new data => this causes the UI elements to be redrawn
- Flutter intelligently rebuilds only the components that changed

Widget Rebuilding Example



- Every time the button is clicked, the button widget raises **onPressed** event to notify the app logic, which increments **clicksCount** state variable
- This causes a **Widget Rebuilding** to take place, i.e., the **ClickCounter** build function is automatically called again to redraw the widget

```
class ClickCounter extends StatefulWidget {  
  const ClickCounter();  
  @override  
  _ClickCounterState createState()  
    => _ClickCounterState();  
}
```

```
class _ClickCounterState extends State<ClickCounter> {  
  int clicksCount = 0;  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: const Text('Click Counter'),  
        centerTitle: true),  
      body: Center(  
        child: ElevatedButton(  
          onPressed: () {  
            setState(() {  
              clicksCount += 1;  
            });  
          },  
          child: Text("I've been clicked $clicksCount times"),  
        ),  
      ),  
    );  
  }  
}
```

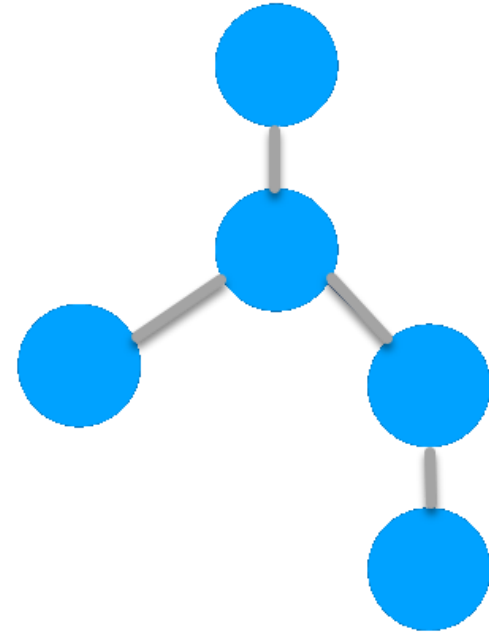
Tip Calculator Example

- In the example below, notice no Compute/OK button, any change of input auto-recomputes and re-displays the tip value
 - Like Excel way: changing a cell value triggers auto-update of formulas and graphs referencing it
- Plus, the code is much more concise and elegant (see posted example)

The screenshot shows a mobile application interface for a tip calculator. At the top is a purple header bar with the title "Tip Calculator" on the left and a fork and knife icon and a menu icon on the right. Below the header is a white input field labeled "Bill Amount". Underneath that is a yellow rounded rectangle containing the text "How was the service?" followed by three radio button options: "Okay (10%)", "Good (15%)", and "Amazing (20%)". The "Good (15%)" option is selected, indicated by a teal dot. At the bottom of the screen is a toggle switch labeled "Round up tip?", which is currently turned off.

How recomposition works

1. Creates an abstract tree representation of the UI and renders it
2. When a change occurs, it creates a new tree representation
3. Computes the differences between the two representations
4. Renders the differences [if any]



For more details about [Jetpack Compose Runtime](#), watch this [video](#)

Stateful versus Stateless

- **stateful** widgets can hold and manage internal mutable state using the `State` class
 - You update the state using the `setState()` method, which re-renders the widget when the state changes.
 - Reduced reusability: the state is internal and not exposed, making it hard to reuse the widget in different contexts or with different external state.
 - Harder testing: Testing stateful widgets is more complex because you need to simulate the state transitions to verify behavior.
 - => Where possible, manage state externally and pass it to widgets to improve reusability and testability.
- A **stateless** composable that doesn't hold any state
 - The caller controls and manages the state
 - State hoisting is a pattern where the state is "hoisted" or moved from a widget into its parent, so the child widgets become stateless
 - The widget that previously managed state now takes the state as an input from the parent

State Hoisting

- To make a widget stateless, **extract** its state and **move it to the parent**
- Then **pass the state** to the widget as a parameter, along with a callback function that the widget can call to update that state in response to events (e.g., `onValueChange`, `onExpand` and `onCollapse`) e.g.,
 - **`name: String`** - the current value to display
 - **`onNameChange: (String) -> Unit`** - a callback that requests the value to change
- Hoisted state variables are owned by the Caller and can be passed to other widgets

State Hoisting - Example

```
@Composable
fun HelloScreen() {
    var name by remember { mutableStateOf("") }

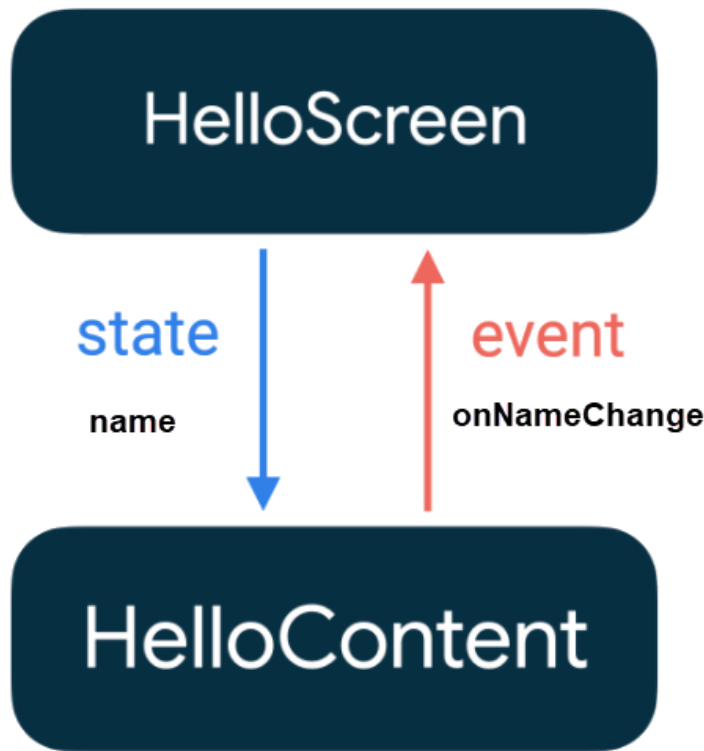
    HelloContent(name = name, onNameChange = { name = it })
}

@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.h5
        )
        OutlinedTextField(
            value = name,
            onValueChange = onNameChange,
            label = { Text("Name") }
        )
    }
}
```

Unidirectional Data Flow

= a design where **state flows down** and **events flow up**

```
var name by remember { mutableStateOf("") }  
HelloContent(name = name, onChange = { name = it })
```



State flows down via widget parameter



(e.g., *name*)

(State change) Event flows up via callback function

(e.g., *onChange*)

By hoisting the state out of `HelloContent`, it can be **reused** in different situations, and it is easier to test

Summary

- Declarative UI is the trend for UI development
 - UI is composed of small reusable widgets
 - **Stateless widgets** don't hold state, making them more reusable and test-friendly
 - **Stateful widgets** manage their own state but are harder to reuse and test
 - **State hoisting** shifts state management to the parent, enhancing the flexibility of child widgets
- Layouts are used to size position widgets on the screen
- Widget is **immutable**
 - It only accepts state & exposes events
 - **Unidirectional Data Flow** pattern:
 - State flows down via parameters
 - Events flow up via callbacks
- .. mastering Flutter will take some time and practice   ...

Resources

- Jetpack compose tutorial

<https://developer.android.com/jetpack/compose/tutorial>

- Jetpack compose Code Labs

<https://developer.android.com/courses/pathways/compose>

<https://developer.android.com/courses/android-basics-compose/course>

- Jetpack Compose Playground - UI component examples

<https://foso.github.io/Jetpack-Compose-Playground/>

<https://github.com/Foso/Jetpack-Compose-Playground>

<https://github.com/Gurupreet/ComposeCookBook>

- Compose Samples

<https://github.com/android/compose-samples>