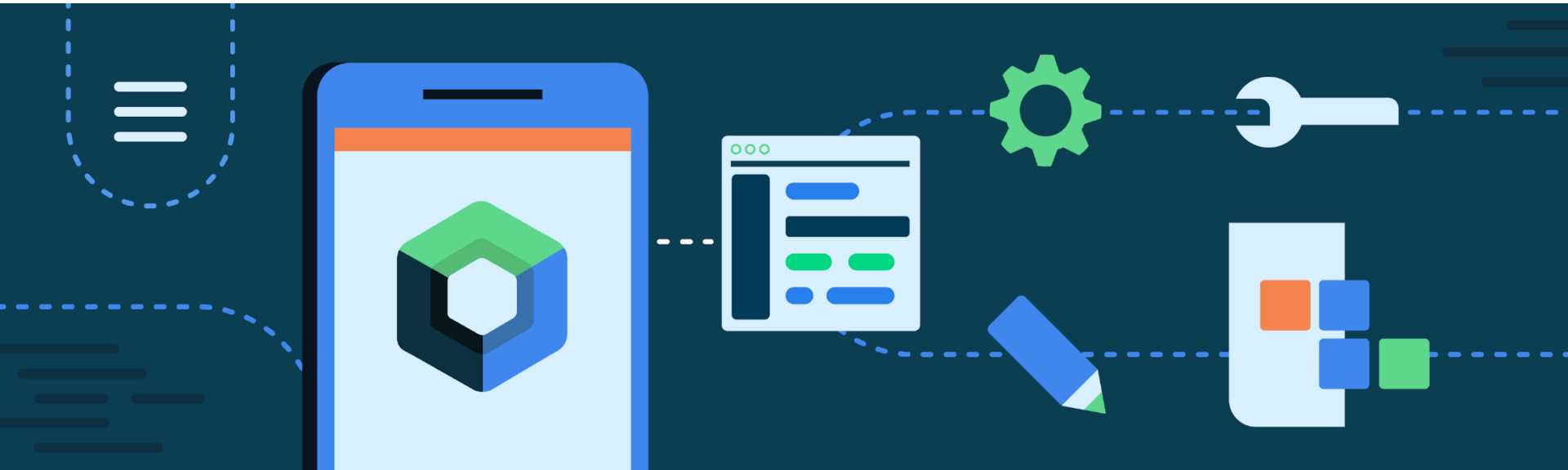


# CMPS 312



## Declarative UI using Jetpack Compose

Dr. Abdelkarim Erradi  
CSE@QU

# Outline

1. Jetpack Compose Key Concepts
2. UI Components
3. Layouts
4. Modifiers
5. State

# Jetpack Compose Key Concepts



<https://developer.android.com/jetpack/compose/mental-model>

# Declarative UI is a major trend

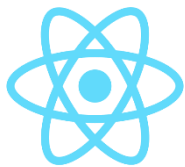
- Describe WHAT to see NOT HOW



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



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



React: A JavaScript library for building user interfaces



Jetpack Compose: a **modern toolkit** for building native Android UI ([released July 2021](#))

# Jetpack Compose

- Jetpack Compose is a **modern UI toolkit** for Android
  - It simplifies UI development with less code and intuitive Kotlin APIs that follow **best practices**
- A **declarative component-based programming model**
  - UI is built using composable functions
    - Each function define a piece 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 views)
- Inspired by/similar to other declarative UI frameworks such as React and Flutter

# Declarative UI Programming Model

- App is composed of one or more **screens** (called **Activity**)
- An **activity** has:

## (1) UI Components

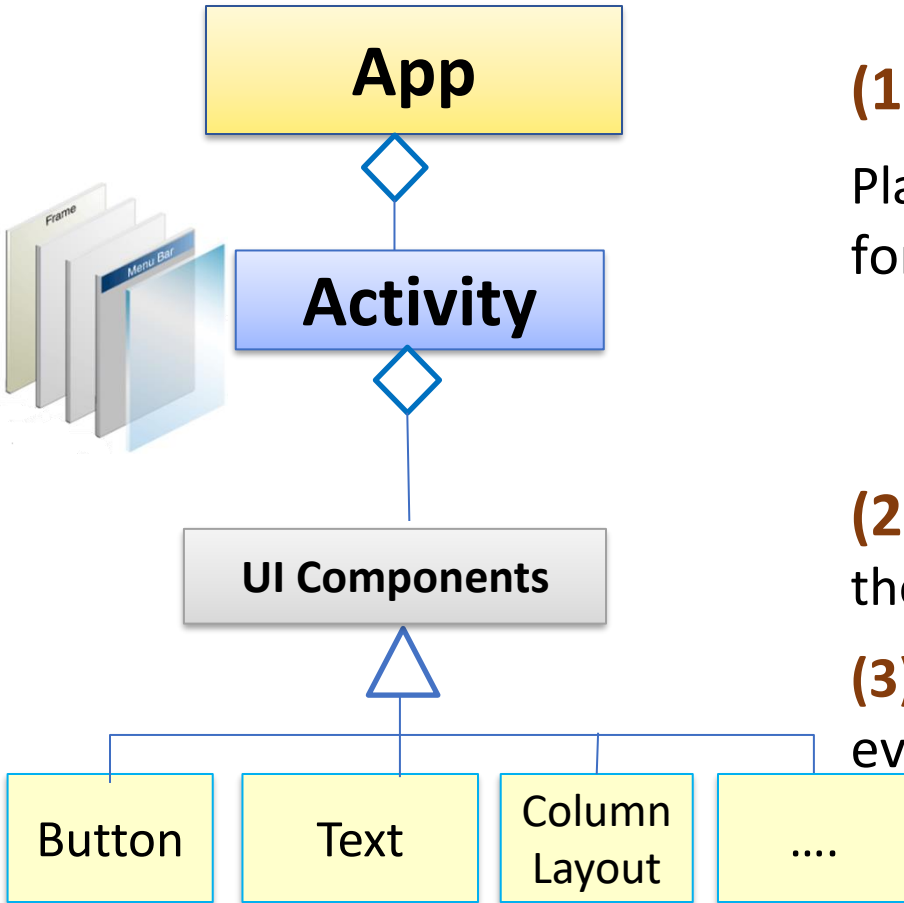
Placed in a **Layout** that acts as a **container** for UI Components

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

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

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

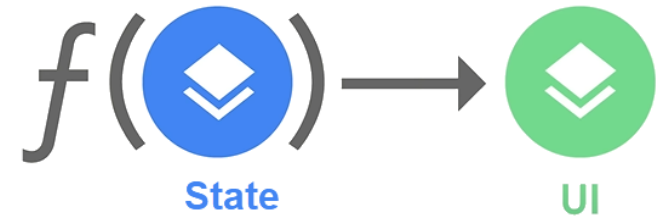
- UI Components **raise Events** when the user interacts with them (such as a Clicked event is raised when a button is pressed)





# How to define a piece of UI?

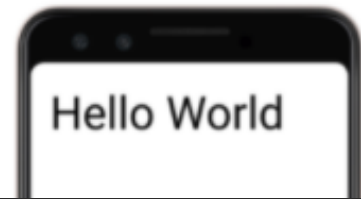
- UI is **composed** of small reusable **components**
- UI Component = Composable **function**:
  - Just a function annotated with **@Composable**
  - Takes some inputs and emits a piece of UI
  - Composable converts the state (i.e., app data) into UI



- **UI = f(state) : UI is a visual representation of state**  
(e.g., display a tweet and associated comments)
- 👍 ○ **State changes trigger automatic update of the UI**

# UI as a function

**String** → `fun Greeting(name: String) =  
println("Hello, $name")` → **stdout**



Mark as a composable

**Data** → `@Composable  
fun Greeting(name: String) =  
Text("Hello, $name")` → **UI**

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

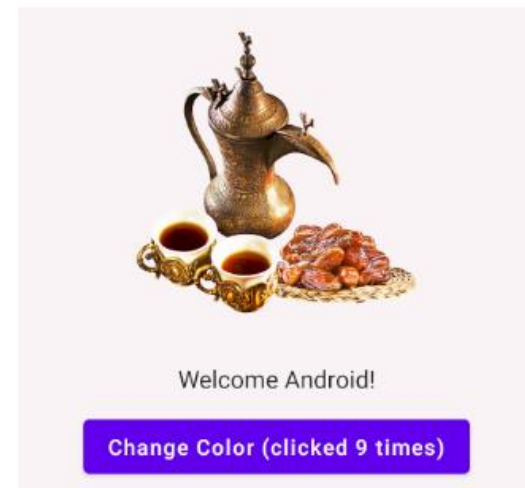




# UI = Composition of UI functions

Your name

- The top-level composable function describes the UI by calling other composables and passing them the appropriate data



@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) {...}
```

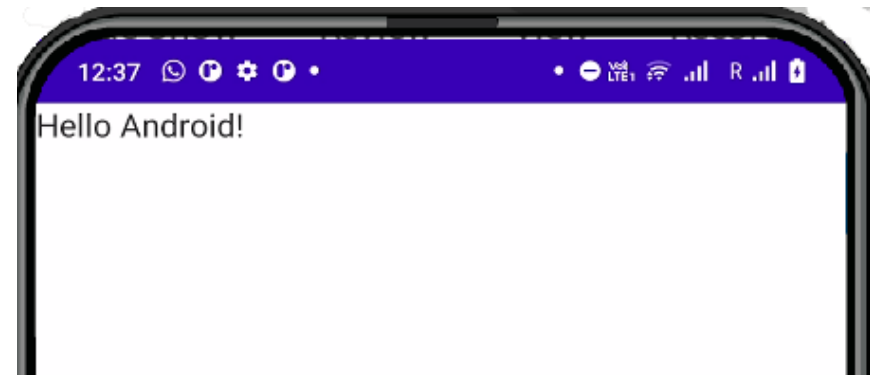
@Composable

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

# App Entry Point

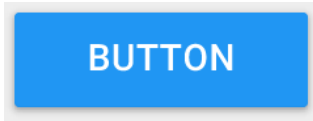
- When the app launches it creates and starts the **Main Activity** (specified in *AndroidManifest.xml*)
- The **Activity** acts as a container to load the UI main screen using **setContent** in the **onCreate** method
  - Modern apps have **1 activity** several and composables that get loaded on-demand and the using interacts with the App

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Greeting("Android")  
        }  
    }  
}  
  
@Composable  
fun Greeting(name: String) {  
    Text(text = "Hello $name!")  
}
```

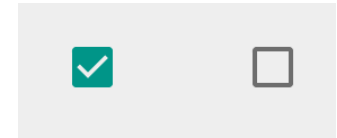


# UI Components

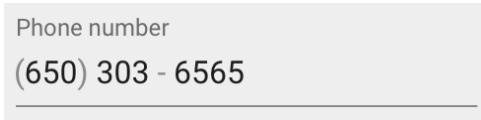
Button



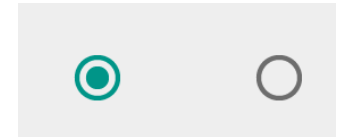
CheckBox



TextField



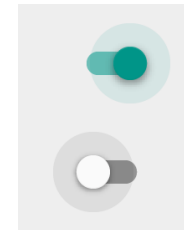
RadioButton



Slider



Switch



**See more details in slides  
'05 UI Components-Layouts'**

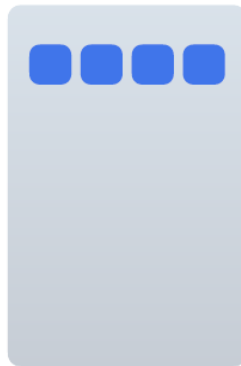
**Full list available at these [link1](#) and [link2](#)**

# Layouts

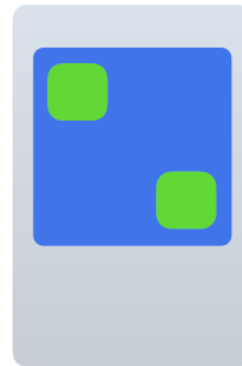
- Use a Layout to size & **position** UI elements on the screen
- **Row** - position elements horizontally
- **Column** - position elements vertically
- **Box** - position elements in the corners of the screen or stack them on top of each other
- Use [Constraint Layout](#) (self-study) for complex layouts



Column



Row



Box



Constraint  
Layout

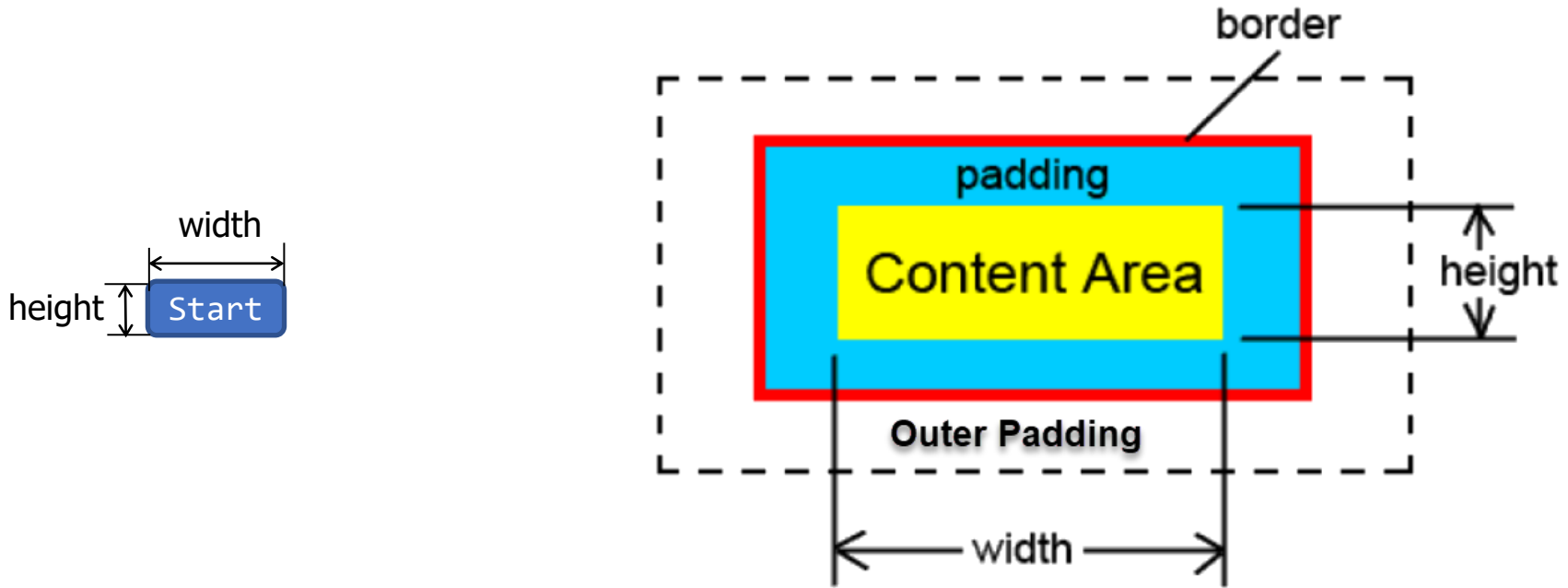
See more details in slides '05 UI Components-Layouts' & this [link](#)

# Modifiers

# Modifiers

- Modifiers are used to configure and customize the style (i.e., the look) or behavior of UI components
  - **Style** UI element such as setting the size, color, border, padding, and **layout parameters** to control spacing and positioning
  - **Add behavior** to UI elements such as making the element clickable or scrollable
- Several modifiers can be **chained**
  - Each modifier **modifies** the composable and **prepares** it for the next modifier in the chain
  - The **order of modifiers** in the chain matters

# Size and Spacing



- Composable size and spacing properties can be set using **Modifiers**:
  - **Outer padding** (aka margin) - the space that separates composables
  - **Border** - the line around each edge of the composable
  - **Padding** - the space between the border and the content

# Size and Spacing - Example

```
Text(  
    text = "Width and Height",  
    color = Color.White,  
    modifier = Modifier  
        .padding(10.dp) // Outer padding (margin)  
        .background(Color.Blue)  
        .width(200.dp)  
        .height(150.dp)  
    // .size(width = 250.dp, height = 100.dp) // Alternative way  
)
```



```
Text(  
    text = "Padding and margin!",  
    modifier =  
        Modifier.padding(16.dp) // Outer padding (margin)  
            .background(color = Color.Yellow) // background color  
            .border(  
                width = 2.dp,  
                color = Color.Gray  
            ) // Add a border  
            .padding(8.dp) // Inner padding  
)
```

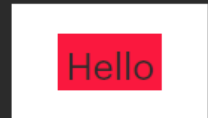




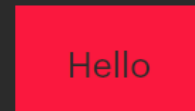
# Modifiers Chain

- Modifiers can be chained and the order matters!
  - Applied in a sequential way and the order impacts the behavior

```
Text(  
  text = "Hello",  
  modifier = Modifier.padding(16.dp)  
    .background(color = Color.Red)  
)
```



```
Text(  
  text = "Hello",  
  modifier = Modifier.background(color = Color.Red)  
    .padding(16.dp)  
)
```



# Another Modifier Example

```
@Composable
fun Greeting(name: String) {
    Column(
        modifier = Modifier.fillMaxSize()
                           .background(Color.Green)
                           .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ){
        Text(text = "Hello $name!")
        Text(text = "Number of coffees: 0")
    }
}
```

Kt



**fillMaxSize** makes the composable fill the maximum width & height given to it from its parent

# Surah Card



110. النصر - An-Nasr

Aya count: 3

@Composable

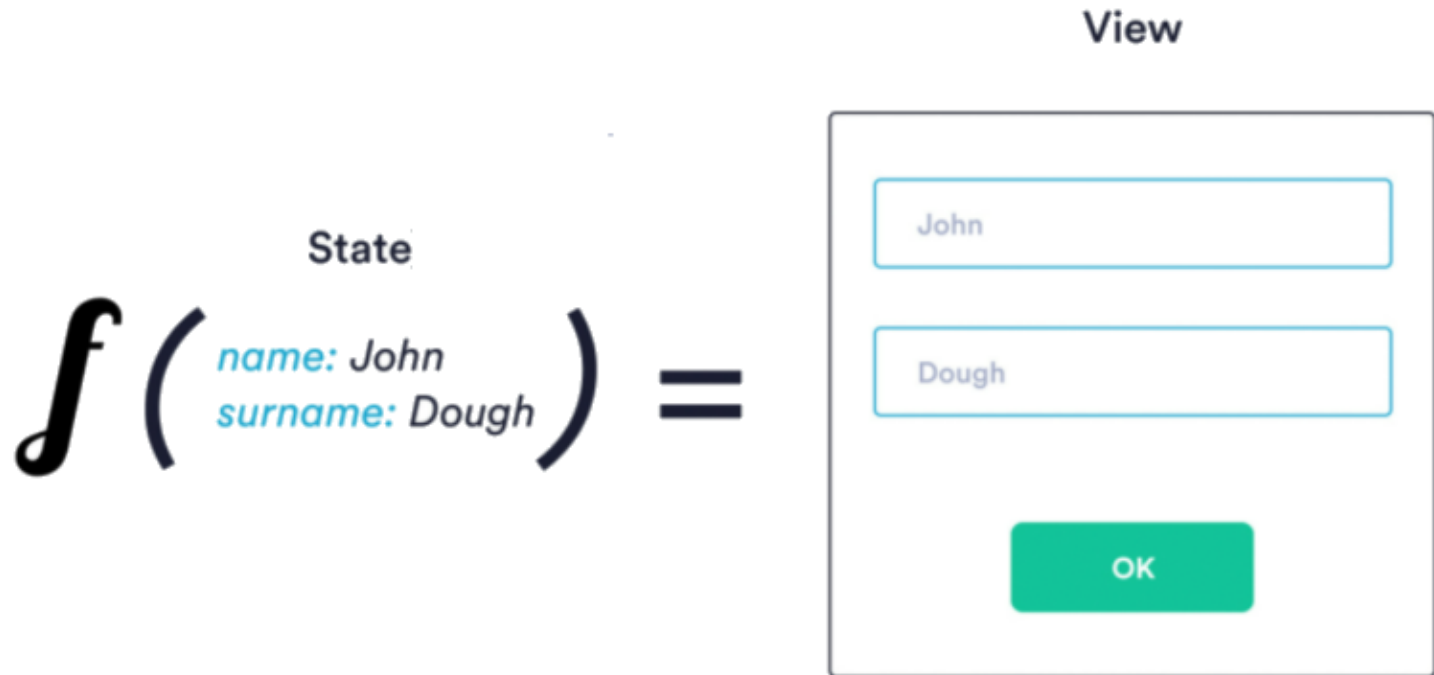
```
fun SurahCard(surah: Surah) {  
    Card (elevation = 10.dp,  
        backgroundColor = if (surah.type == "Medinan") lightGreen else lightYellow,  
        modifier = Modifier  
            .fillMaxWidth()  
            .padding(horizontal = 5.dp)  
            .border(width = 2.dp, color = Color.LightGray, shape = RoundedCornerShape(8.dp))  
    ) {  
        Row (verticalAlignment = Alignment.CenterVertically,  
            horizontalArrangement = Arrangement.spacedBy(4.dp),  
            modifier = Modifier.padding(5.dp)  
        ) {  
            val imgResourceId = if (surah.type == "Medinan") R.drawable.ic_madina  
                                else R.drawable.ic_mecca  
            Image(painter = painterResource(id = imgResourceId),  
                contentDescription = "Surah Type",  
                Modifier.height(50.dp)  
            )  
            Column(verticalArrangement = Arrangement.spacedBy(2.dp)) {  
                Text(text = "${surah.id}. ${surah.name} - ${surah.englishName}")  
                Text(text = "Aya count: ${surah.ayaCount}")  
            }  
        }  
    }  
}
```

# Modifier.*clickable*

```
Text(  
    text = "+",  
    modifier = Modifier  
        .border(2.dp, Color.Gray)  
        .padding(10.dp)  
        .clickable {  
            count += 1  
        },  
    style = MaterialTheme.typography.titleLarge  
)
```



# State



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

# State

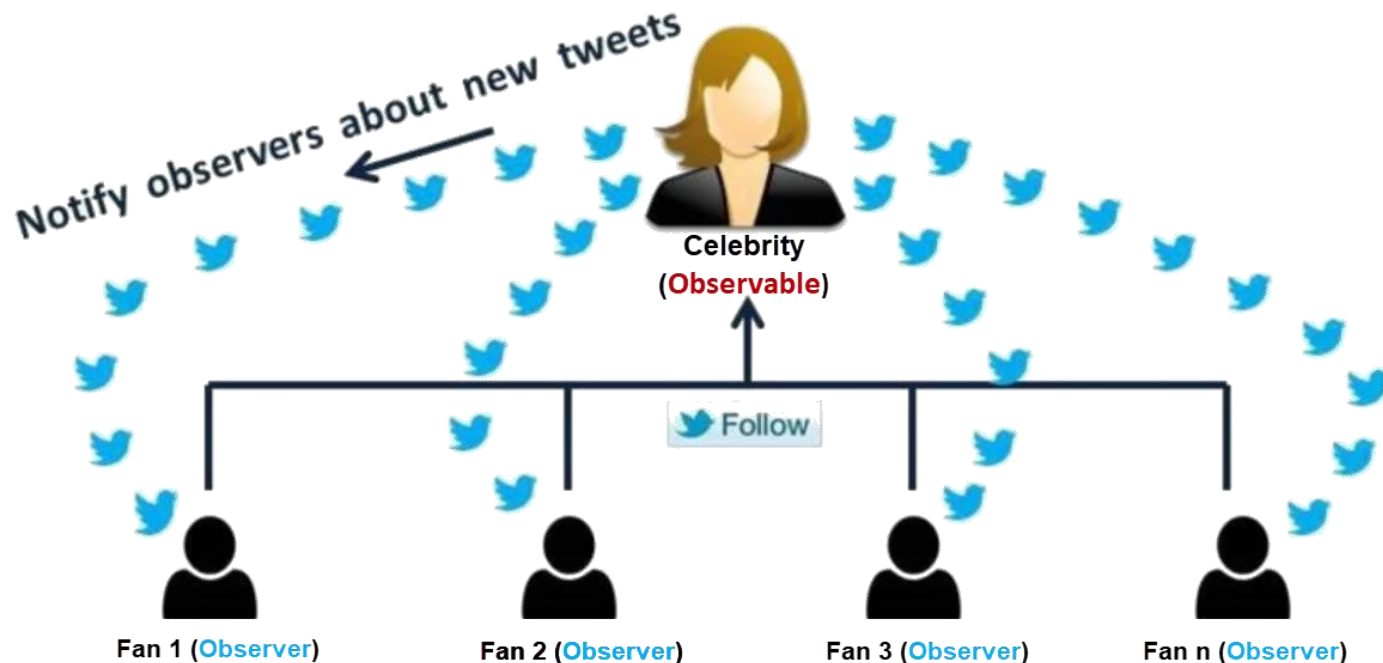
- State = any value that can change overtime
  - State variable must be declared as **Mutable State** variables to act as **Change Notifiers**
    - They **are observed** by the Jetpack compose runtime
    - 👍 ○ Any change of a state variable will trigger the **recomposition** of any composable functions that **reads** the state variable
- => UI is **auto-updated** to reflect the updated app state

```
var stateVar by remember { mutableStateOf(defaultVal) }
```

- **remember** is used to **store** values of state variable in the **composition tree** (to preserve the values and avoid reinitialization to the default value during the recomposition)
  - the stored value is returned during recomposition

# Observer Pattern at the heart of Jetpack Compose

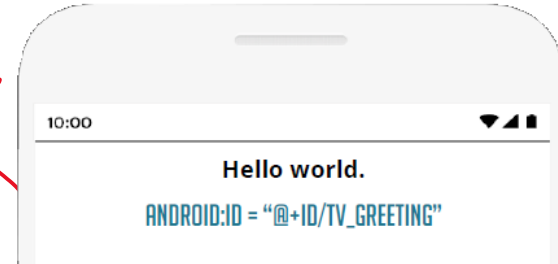
- [Observer Pattern](#) Real-Life Example: A celebrity who has many fans on Tweeter
  - Fans want to get all the latest updates (posts and photos)
  - Here fans are **Observers** and celebrity is an **Observable** (analogous **Mutable State** in Jetpack Compose)
  - **Mutable State** is an **observable data holder**: Jetpack Compose runtime **observes its changes** and updates the UI accordingly



# Imperative UI vs. Declarative UI

- Imperative UI – call a setter on the view to change its internal state

```
TextView greetings = (TextView) findViewById(R.id.tv_greeting)  
greetings.text = "Hello world."
```



- UI in Compose is immutable
  - In compose 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 ***greeting text*** can only be changed by updating the ***name*** state variable

```
@Composable  
fun WelcomeScreen() {  
    var name by remember { mutableStateOf("Android") }  
    Greeting(name)  
}
```

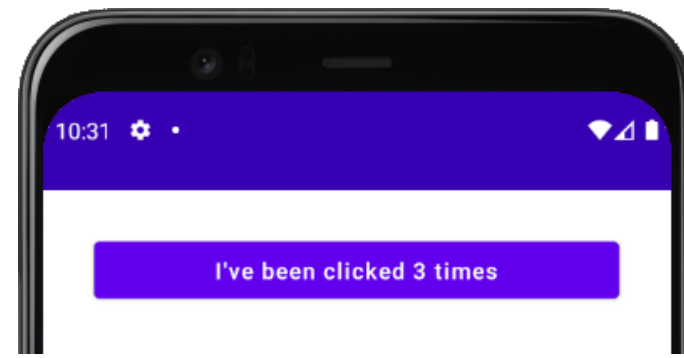
```
@Composable  
fun Greeting(name: String) {  
    Text(text = "Hello $name!")  
}
```



# Recomposition

- When the user interacts with the UI, the UI raises events such as onClick
  - Those events should notify the app logic, which can then change the app's state
  - When the state changes it causes the composable functions to be automatically called again with the new data => this causes the UI elements to be redrawn
  - This process is called **recomposition**
- The Compose framework can intelligently recompose only the components that changed

# Recomposition Example



- Every time the button is clicked, the UI raises **onClick** event to notify the app logic, which increments **clicksCount** state variable
- This causes a **recomposition** to take place, i.e., the **ClickCounter** function is automatically called again to redraw the Button

@Composable

```
fun MainScreen() {  
    var clicksCount by remember { mutableStateOf(0) }  
    ClickCounter(clicks = clicksCount, onClick = { clicksCount += 1 })  
}
```

@Composable

```
fun ClickCounter(clicks: Int, onClick: () -> Unit) {  
    Button(onClick = onClick) {  
        Text("I've been clicked $clicks times")  
    }  
}
```

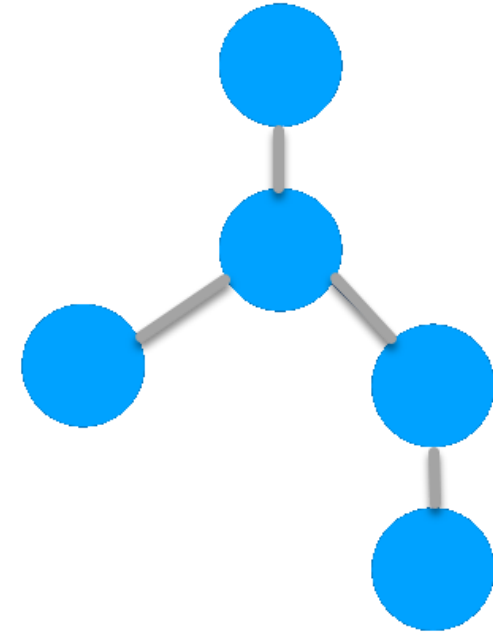
# Tip Calculator Example

- 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" and two icons on the right: a fork and knife, and a vertical ellipsis. Below the header is a white input field labeled "Bill Amount". Underneath this 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

- A stateful composable uses **remember** to store an object in the composition tree
  - However, stateful composable tend to be less reusable and harder to test
- A stateless composable that doesn't hold any state
  - The caller controls and manages the state
  - An easy way to achieve stateless is by using **state hoisting**

# State Hoisting

- To make a composable stateless, **extract** its state and **move it to the caller** of the composable
- Then **pass the state** to the composable as an immutable parameter, along with a callback function that the composable 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 composables

# State Hoisting - Example

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

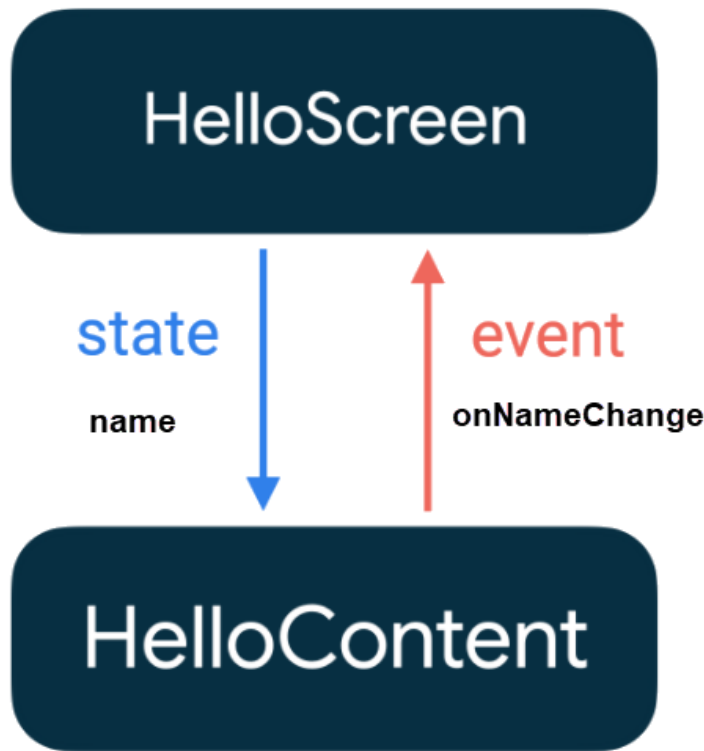
    HelloContent(name = name, onNameChange = { name = it })
}

@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  
function 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 components
- UI Component = Composable function
  - just a function annotated with **@Composable**
- Layouts are used to position UI elements on the screen
- UI in Compose is **immutable**
  - It only accepts state & exposes events
  - **Unidirectional Data Flow** pattern:
    - State flows down via parameters
    - Events flow up via callbacks
- .. mastering Compose 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>

- 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>