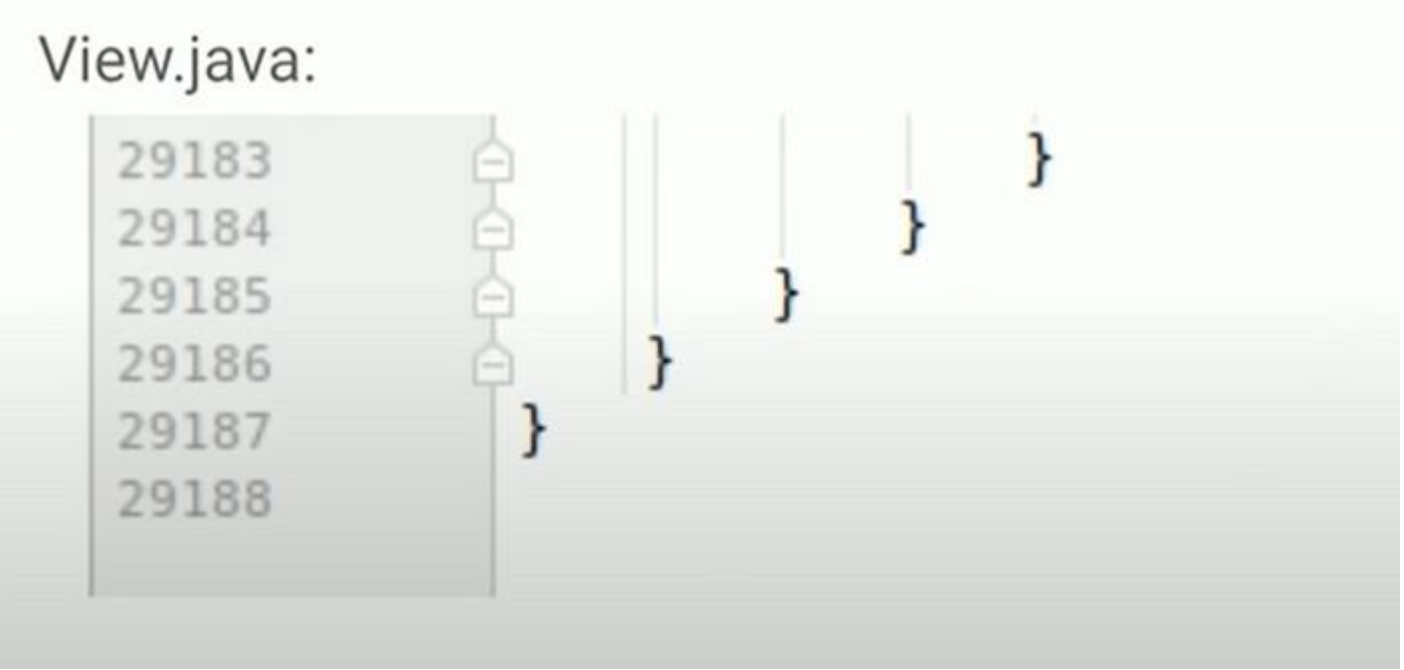
Unit-6 Jet Pack Compose

By, Prof. Hiten Sadani


2008
Android 1.0
Android Studio
2013
2014
RecyclerView
ConstraintLayout
Support Design Library
2017
2018
Material Design
Components
Jetpack Compose
2019

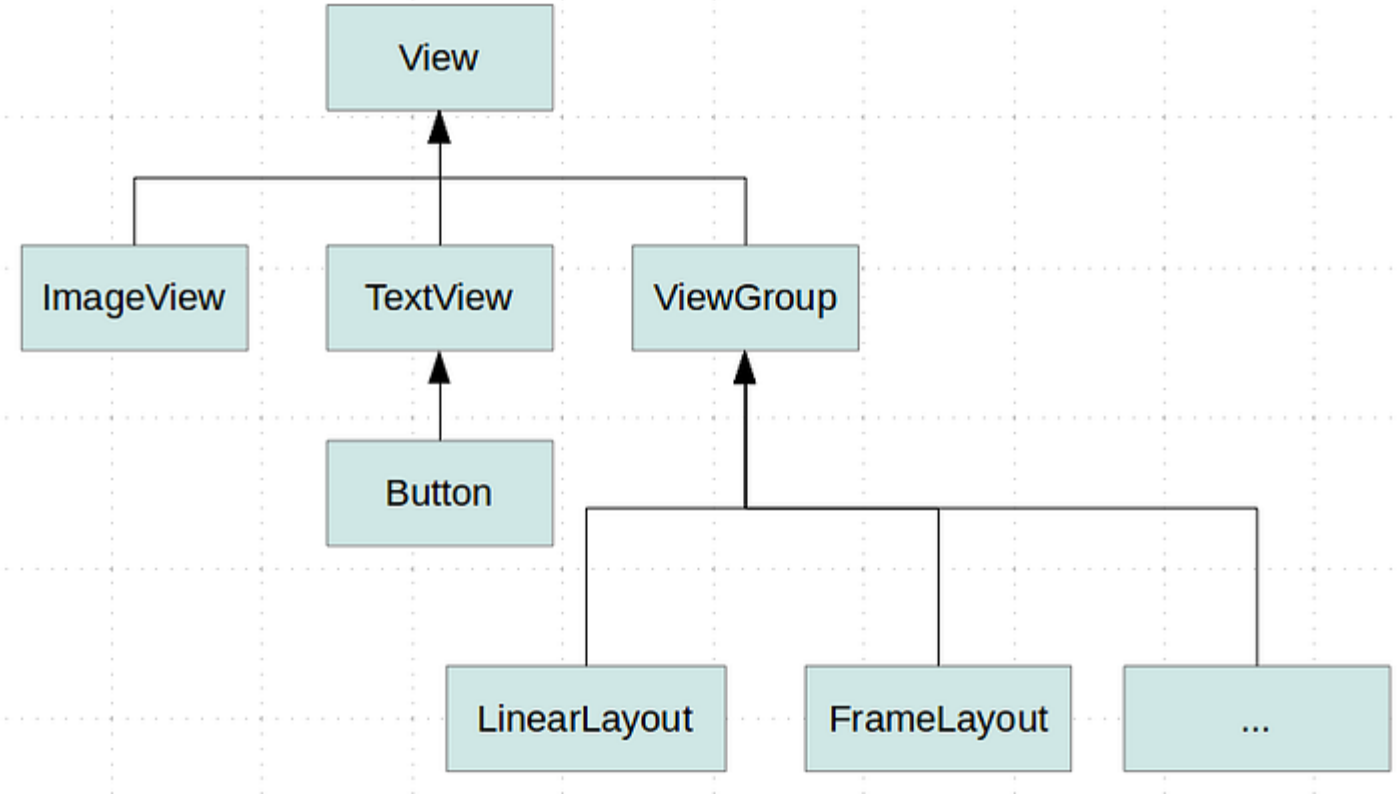
Why need tool like Jetpack Compose?

- problems that the view based design system (the approach of creating UI with xml) has caused to us android developers while designing the UI.
- the View.java class approaches a terrible figure of 30,000 rows.

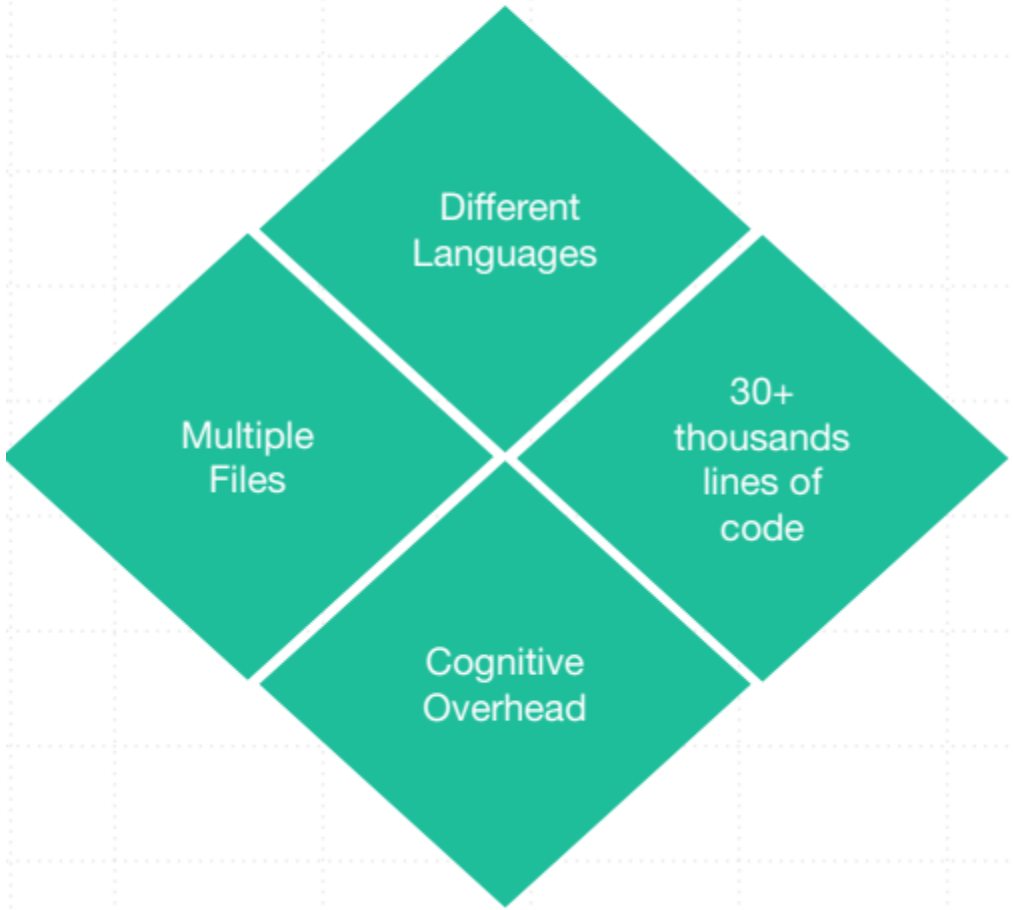


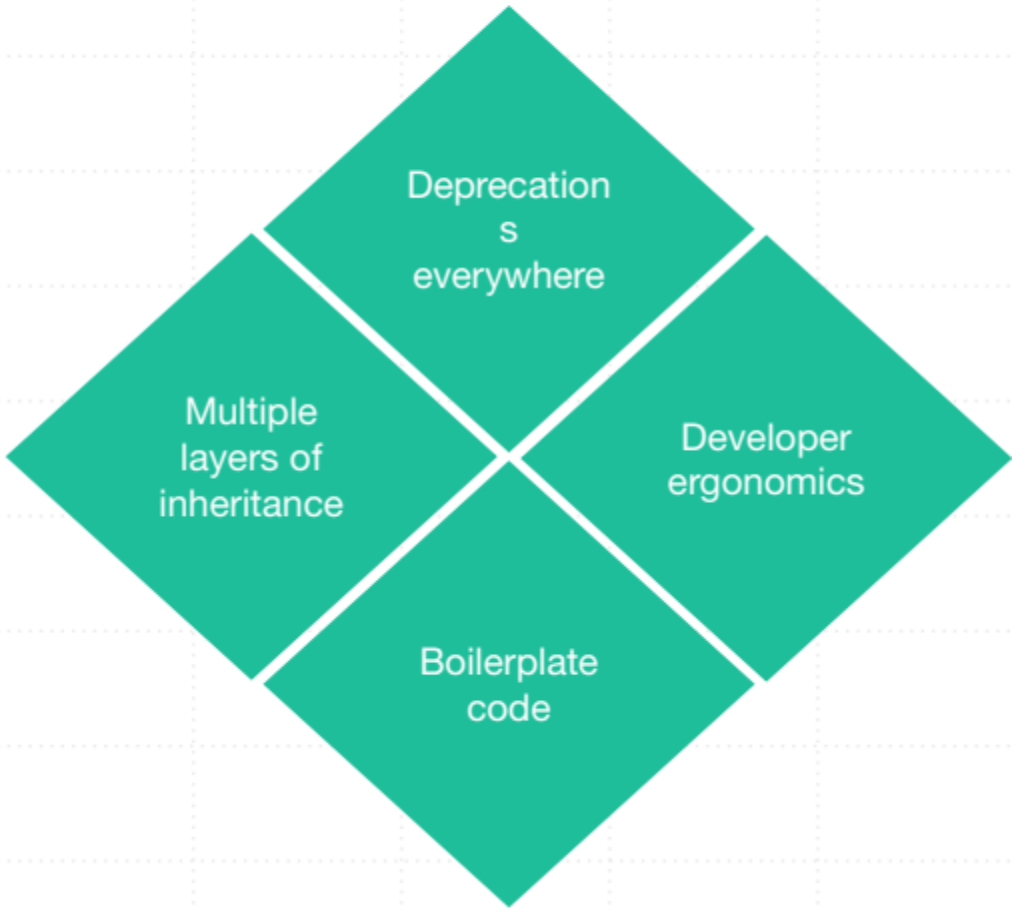
Why need tool like Jetpack Compose?

- The Android toolkit team states that the number of codes reached by this class has become unmanageable, and they want to seek a solution to this as the first problem.



Challenges with Views System





Why need tool like Jetpack Compose?

- strange inheritance structure as another problem.
- the Button class inherits from the TextView class, there are cases where the Button text takes on features such as selectable and editable with this abstraction. It would not be very correct behavior for the button to carry these properties. They state that they considered the situation created by the strange inheritance structure here during the development process of Jetpack Compose.

Why need tool like Jetpack Compose?

- When coding a page with the view based design approach, we usually code an xml layout, collect various common behaviors for views around styles.xml, and when coding structures such as custom views, we write our attributes in the attrs.xml class and finally we set our ui logic in an activity and or fragment (.kt or .java) file. This approach inevitably causes us to write many lines of code. We also use two different languages while doing this coding (java|kotlin & xml). This situation has become one of the problems that the android toolkit team wants to change and improve.

Why need tool like Jetpack Compose?

- In another case, the Source Of Truth event arising from the distress caused by the imperative ui (we will talk about this in detail later) approach. In the Imperative UI approach; We inflate a layout, then we access the components in it with various methods (findViewById, viewbindig, databindig .. etc.) and by changing the internal states of the accessed views, we provide the state that appears on the screen. If we give an example of state change events that are meant here; Examples can be given such as changing visibility (like gone, visible), setting text to TextView ..etc.

UI logic and definition separate

Layout Definition

XML

Static

Completely separate, can't use syntactic sugar and language features. (unless you are databinding fanatic)

Generate tons of boilerplate codes, especially when using bindings.

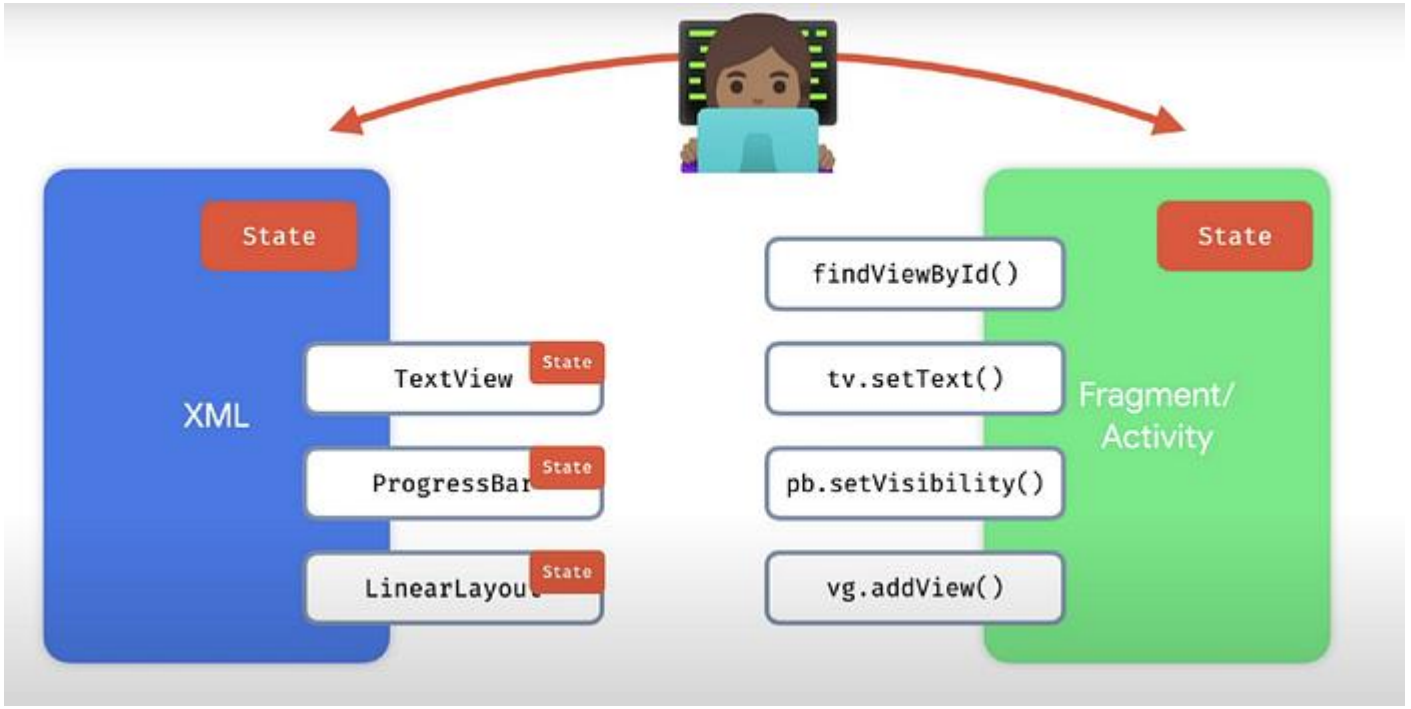
Logic & Behaviour

Kotlin/java

Dynamic

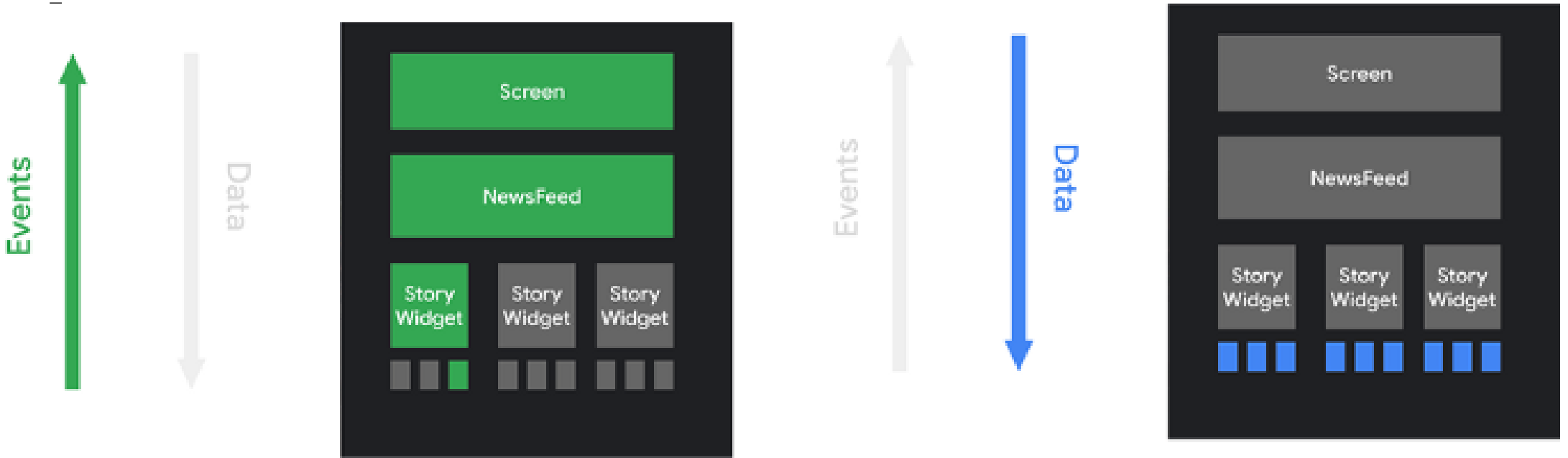
Imperative UI

- The Imperative UI approach is the view based approach we currently use while developing android.



Declarative Ui

- Declarative UI approach is the approach used in the technologies used in both mobile and web development in the software world. Moreover, the team that developed Jetpack Compose states that they are also inspired by these technologies. Examples include Flutter, React Native, Litho, Vue



Imperative UI

vs

Declarative UI

Focuses on ***How***

Inheritance is prefered

View and ***logic*** are seperate

Tons of ***Boilerplate Code, slow*** development

Focuses on ***What***

Composition is prefered

Uses ***one*** language to build entire application.

Less ***Code, faster*** development

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, Jetpack Compose!"
        android:textColor="@android:color/black"
        android:textSize="24sp" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is a simple example of using Jetpack Compose to build a user interface."
        android:textColor="@android:color/darker_gray"
        android:textSize="16sp" />

</LinearLayout>
```

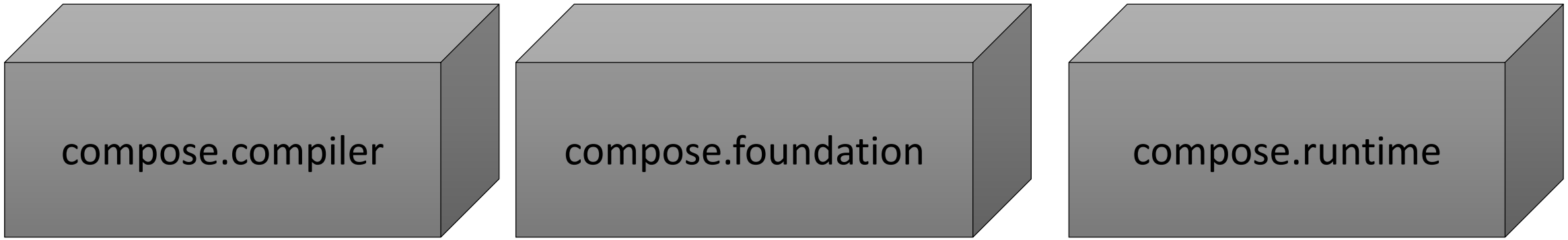
```kotlin
@Composable
fun SimpleUI() {
    MaterialTheme {
        Column(modifier = Modifier.padding(16.dp)) {
            Text(
                text = "Hello, Jetpack Compose!",
                style = MaterialTheme.typography.h1,
                color = Color.Black
            )
            Text(
                text = "This is a simple example of using Jetpack Compose to build a user interface.",
                style = MaterialTheme.typography.body1,
                color = Color.Gray
            )
        }
    }
}
```

- Continuous updates (like most Jetpack components)
- Compose is added to module-level *build.gradle* file
- Devs can pick or skip updates (based on what has changed)

Transform `@Composable` functions and enable optimizations with a Kotlin compiler plugin

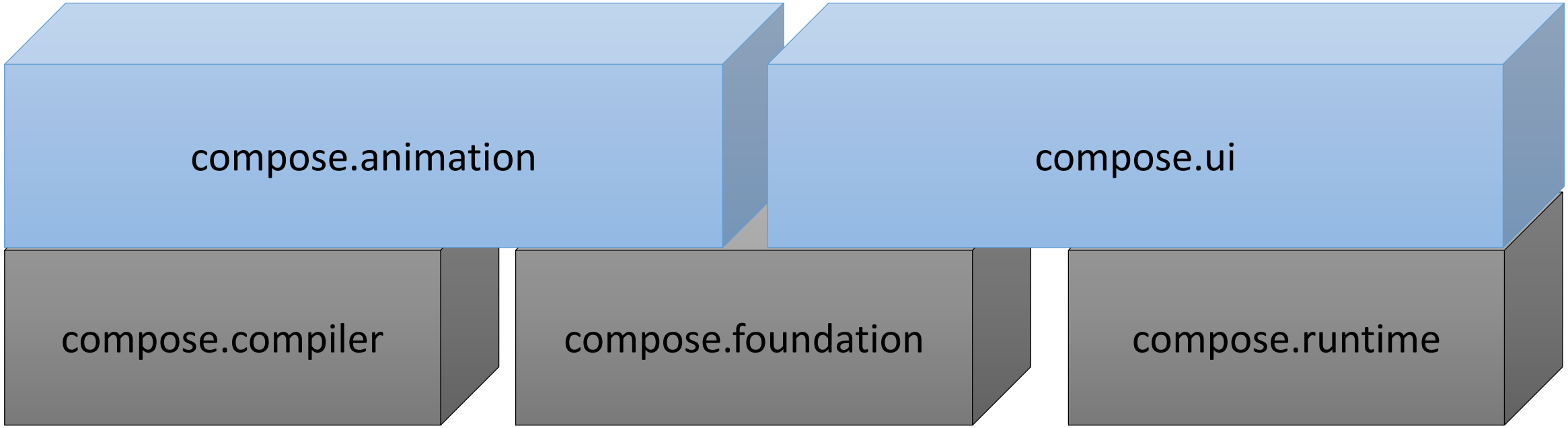
Basic functionality like text and drawing primitives

Fundamental building blocks of Compose's programming model and state management; core runtime targeted by the Compose Compiler Plugin

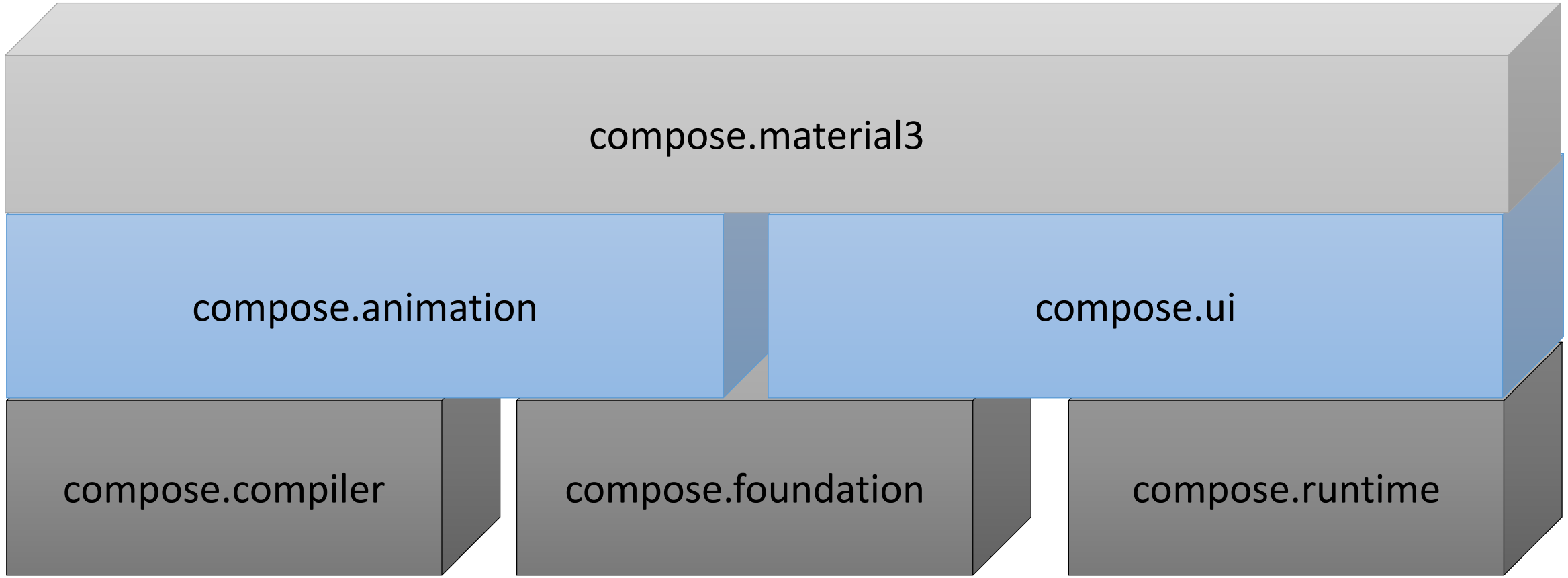


Build animations in your Compose apps to enrich the user experience

Fundamental components of the Compose UI needed to interact with the device, for example layout, drawing, and input



Material Design Components. Higher-level entry point of Compose, designed to provide components that match those described at *www.material.io*.



- The user interface is declared by nesting composable functions
- Compose hierarchies can be shown inside activities and fragments
- Composables and traditional views can be mixed

- Jetpack Compose is only loosely coupled with the underlying platform
- Compose Multiplatform by JetBrains makes Jetpack Compose available iOS, the desktop, and the web

Jetpack Compose Overview

- Jetpack Compose is a modern toolkit for building native Android UI
- Jetpack Compose simplifies and accelerates UI development on
- Android with less code, powerful tools, and intuitive Kotlin API
- Much easier, no more jumping between code and xml
- Some UI elements a lot easier to implement (Lists!)

Jetpack Compose Essentials

- Composables
- State
- Coroutines
- LaunchedEffect
- rememberCoroutineScope

Composable

- Composables are functions in Jetpack Compose that define UI components.
- Annotated with @Composable.
- Can contain other composables, creating a hierarchy.
- Automatically update UI with state changes.
- Executed by the Compose runtime, not directly by the developer.
- Can accept parameters for configuration and dynamic content.
- Follow a declarative programming model.
- Designed for Kotlin and integrate seamlessly with Android's UI toolkit.

Feature	React Components	Jetpack Compose Composables	SwiftUI Views
Paradigm	Declarative	Declarative	Declarative
UI Definition	JSX	Kotlin code with @Composable	Swift code
State Management	**useState**, useContext, etc.	**remember**, **mutableStateOf**, etc.	@State, @Binding, etc.
Reusability	Components can be reused	Composables can be reused	Views can be reused
Hierarchical Structure	Component tree	Composable tree	View hierarchy
Rendering Mechanism	Virtual DOM, diffing algorithm	Slot table, smart recomposition	Declarative UI, diffing algorithm
Lifecycle Management	**useEffect**, useMemo, etc.	DisposableEffect, **LaunchedEffect**	onAppear, onDisappear
Environment Adaptation	Responsive design with CSS or libs	Modifiers, MaterialTheme, etc.	EnvironmentValues, @Environment
Platform Integration	Web (DOM)	Android	iOS, macOS, watchOS, tvOS
Development Language	JavaScript, TypeScript	Kotlin	Swift
Community and Ecosystem	Large, with extensive libraries	Growing, Android-centric	Growing, Apple platforms-focused
UI Updates	Explicit re-renders	Automatic UI updates	Automatic UI updates
Data Flow	Props down, events up	Parameters, state hoisting	Binding, state hoisting
Debugging Tools	React Developer Tools	Layout Inspector, Compose Preview	Xcode Previews, Instruments
Learning Curve	Moderate	Moderate to high	Moderate to high

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Greeting("Android")
        }
    }
}

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

No XML, give composable function

Annotates the function to be composable, part of the UI

Text is another composable, notice now return!

```
dependencies {
    implementation("androidx.core:core-ktx:1.12.0")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.6.2")
    implementation("com.google.android.material:material:1.10.0")

    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")

    ////////////////////
    // Jetpack Compose
    ////////////////////

    implementation(platform("androidx.compose:compose-bom:2023.10.00"))
    // basic APIs
    implementation("androidx.compose.ui:ui")
    // Android Studio Preview support
    implementation("androidx.compose.ui:ui-tooling-preview")
    // Material Design 3
    implementation("androidx.compose.material3:material3")

    androidTestImplementation(platform("androidx.compose:compose-bom:2023.10.00"))
    // Android Studio Preview support
    debugImplementation("androidx.compose.ui:ui-tooling")
    // UI Tests
    androidTestImplementation("androidx.compose.ui:ui-test-junit4")
    debugImplementation("androidx.compose.ui:ui-test-manifest")

    // Integration with activities
    implementation("androidx.activity:activity-compose:1.8.0")
    // Integration with ViewModels
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.6.2")
}
```

```
plugins {
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
}

android {
    ...
    buildFeatures {
        compose = true
    }
    composeOptions {
        kotlinCompilerExtensionVersion = "1.5.3"
    }
    ...
}

```

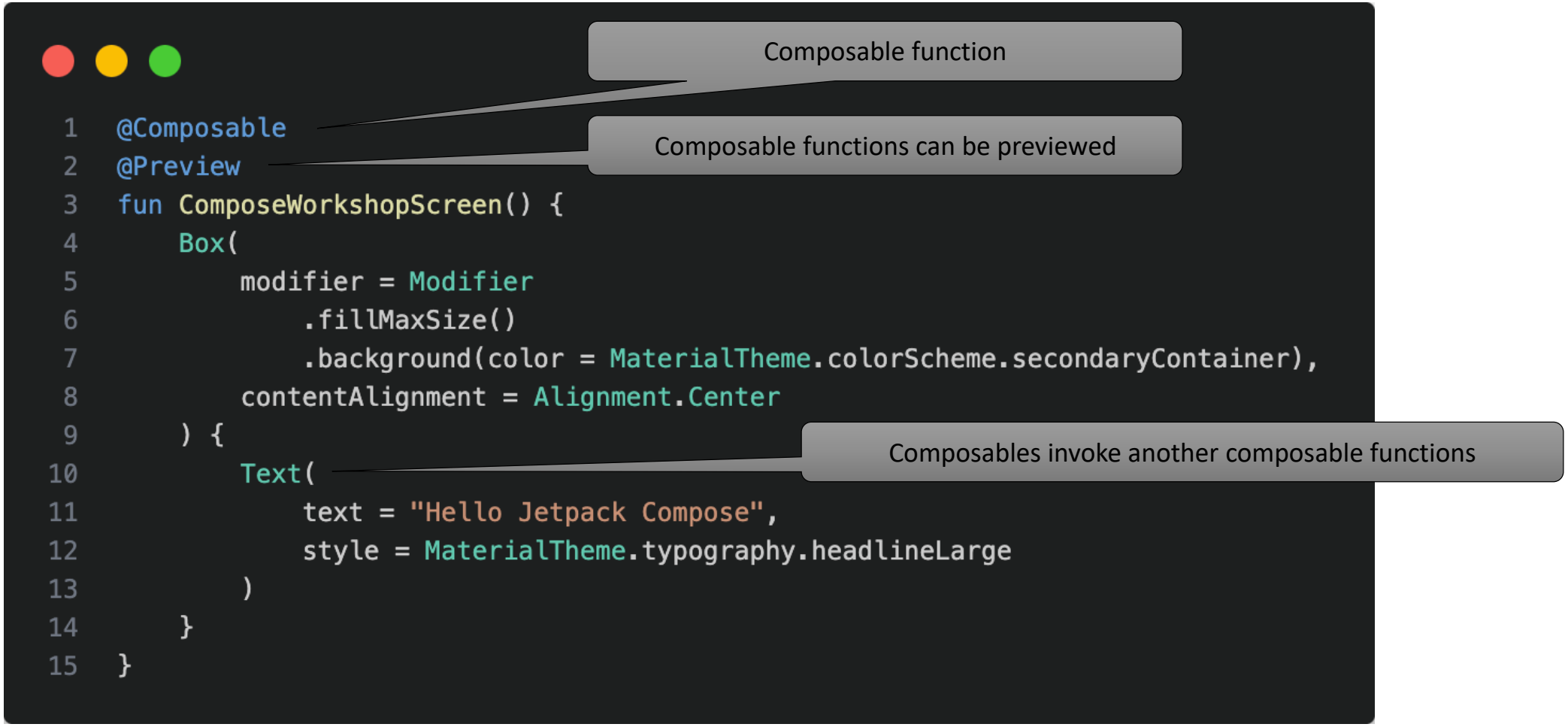
```
plugins {
    id("com.android.application") version "8.1.2" apply false
    id("org.jetbrains.kotlin.android") version "1.9.10" apply false
}

```


class MainActivity : ComponentActivity() {
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContent {
MaterialTheme {
ComposeWorkshopScreen()
}
}
}
}

Modifiers

- Modifiers augment composables, allowing you to modify their appearance, size, padding, and behavior. For example:

- Text("Welcome")
- .padding(16.dp)
- .clickable { /* Handle click event */ }


Composable function
Composable functions can be previewed
Composables invoke another composable functions
@Composable
@Preview
fun ComposeWorkshopScreen() {
Box(
modifier = Modifier
.fillMaxSize()
.background(color = MaterialTheme.colorScheme.secondaryContainer),
contentAlignment = Alignment.Center
) {
Text(
text = "Hello Jetpack Compose",
style = MaterialTheme.typography.headlineLarge
)
}
}

- Composables can receive parameters
- Usually, their names are noun phrases
- They usually do not return a result
- Invoking a composable adds it to internal data structures
- This usually makes the composable visible at some point

- Visual appearance and behavior can be defined through
  - parameters
  - modifiers
- Modifiers are component properties on steroids

```
@Composable
@Preview
fun ComposeWorkshopScreen() {
    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(color = MaterialTheme.colorScheme.secondaryContainer),
        contentAlignment = Alignment.Center
    ) {
        Text(
            text = "Hello Jetpack Compose",
            style = MaterialTheme.typography.headlineLarge
        )
    }
}
```

Modifiers

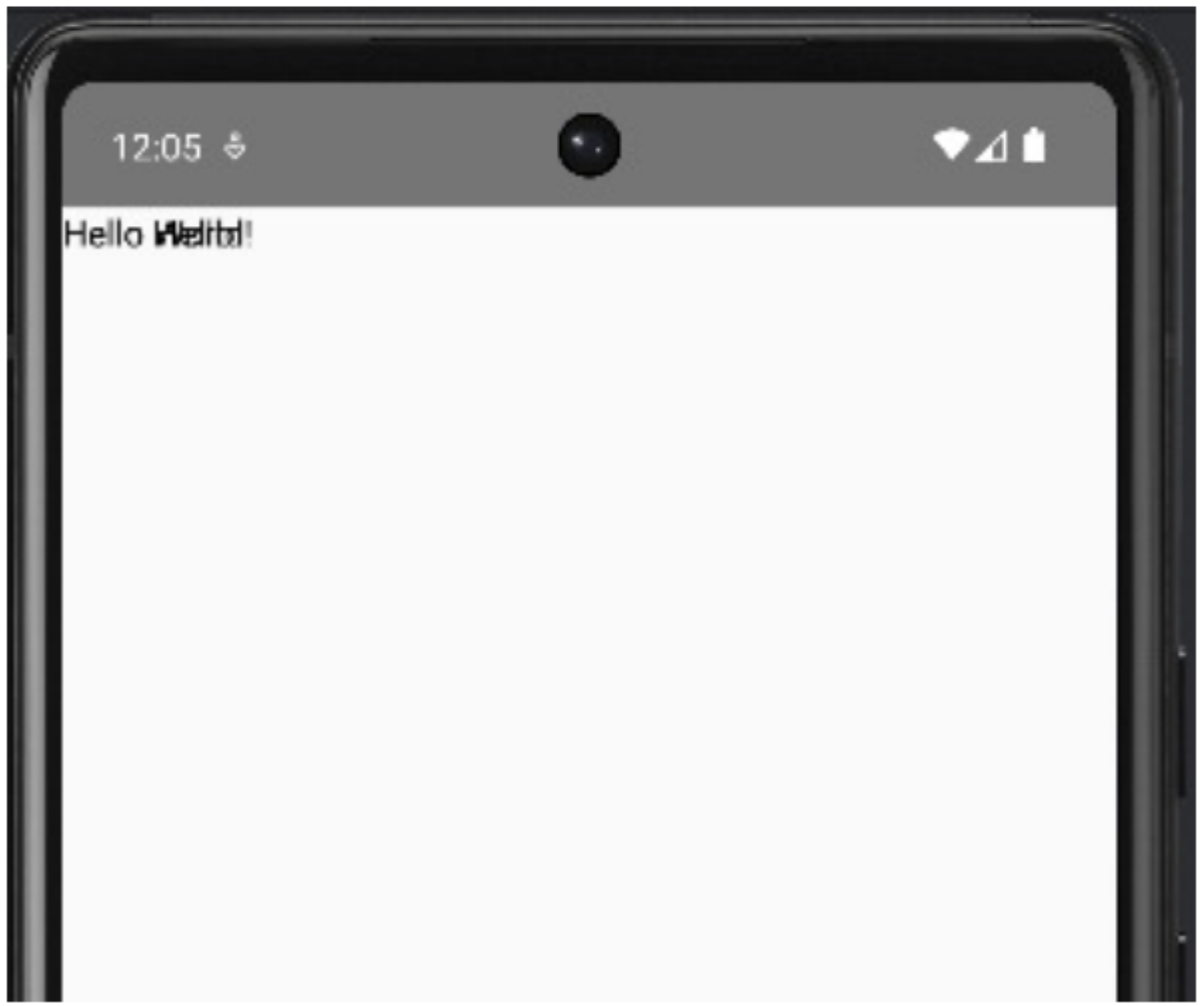
- An ordered, immutable collection of modifier elements
- Decorates or Add behaviour to Compose UI elements
- Modifiers can be chained
- Chaining order is important

- Properties in class-based frameworks are available only in the class that defines them, and its children
- Modifier can be applied to any composable that can receive modifiers

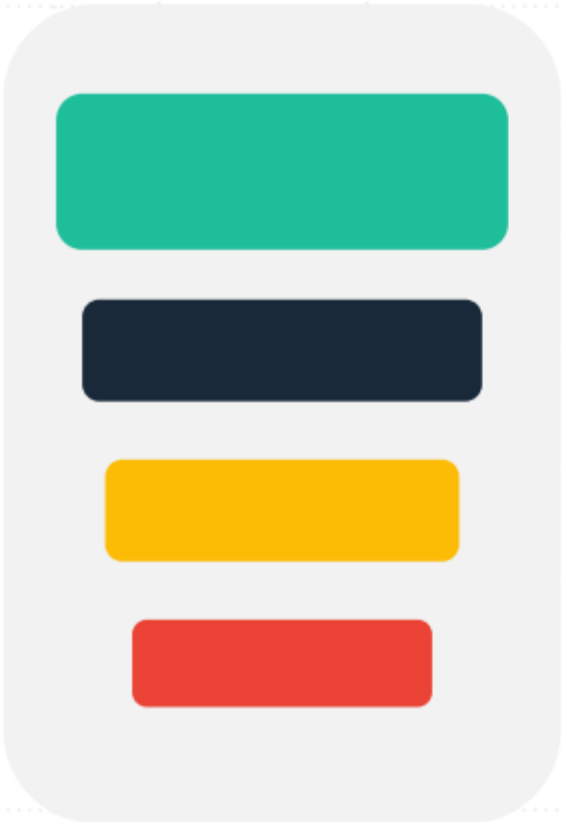
- **Modifier** amend composables with visual properties and behavior
- They form chains, which start with `Modifier`
- The order in the code defines when a modifier is applied

- `.fillMaxSize()` maximizes a composable
- `.width()` and `.size()` declare the preferred width or size
- `.padding()` creates an inward-facing space
- `.background(color = ...)` creates a colored background

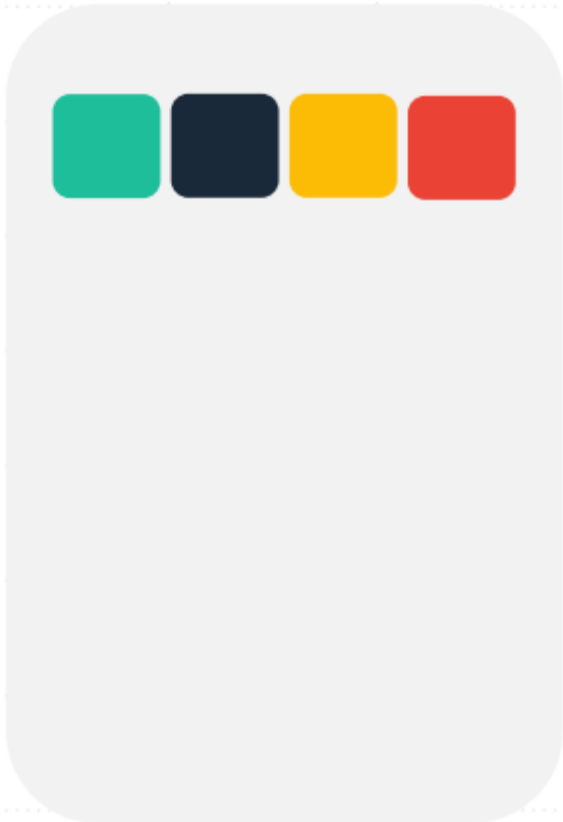
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            App()
        }
    }
}

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



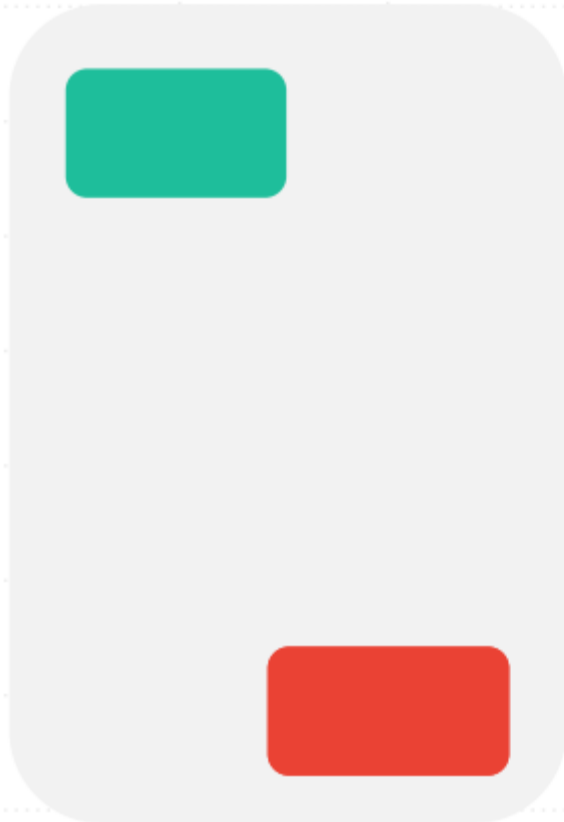
Layouts



Column



Row



Box

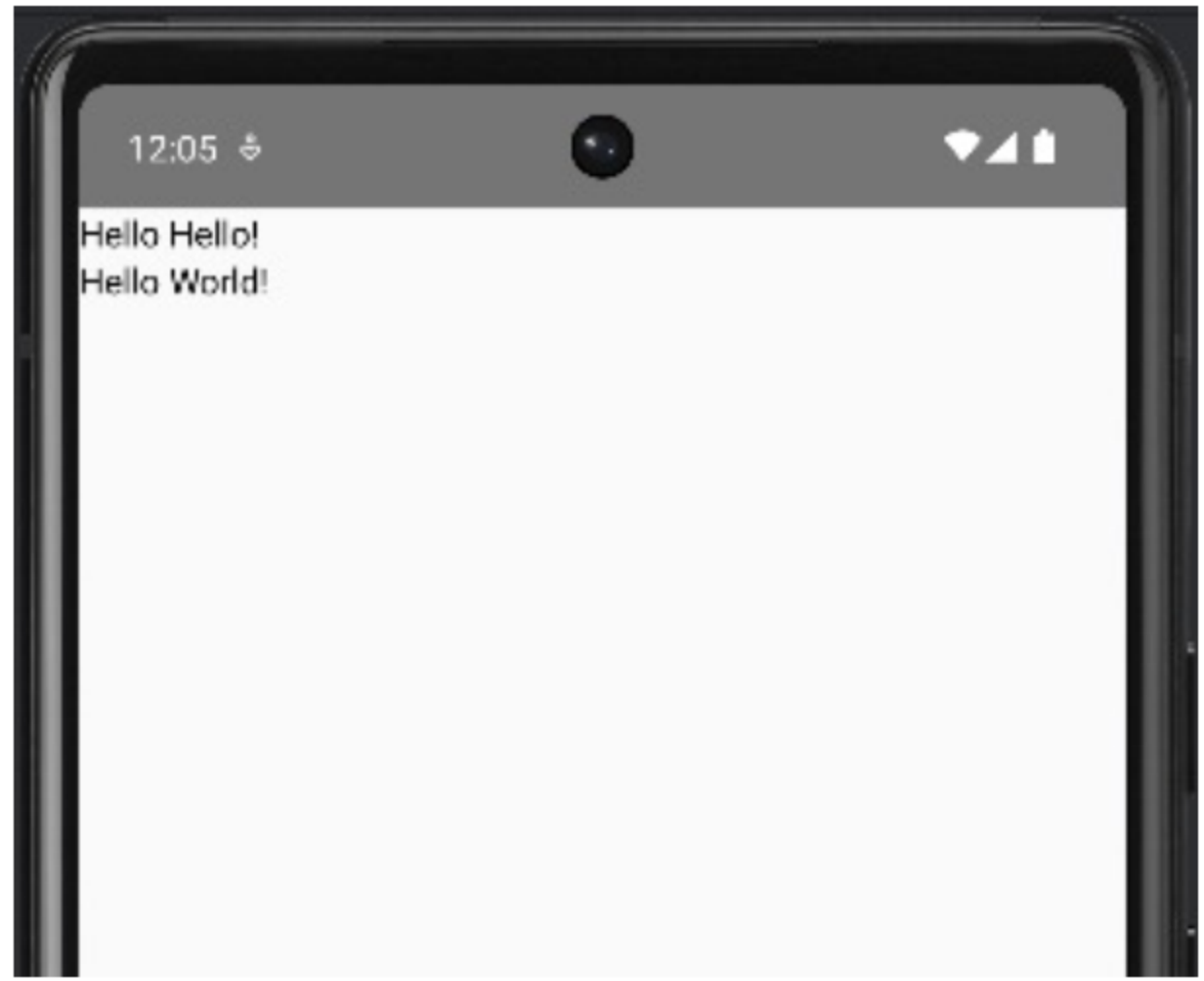
Constraint

Layouts

- Layouts Jetpack Compose provides a collection of layout composables to arrange your UI elements in various ways:
- Column: Arranges elements in a vertical list.
- Row: Arranges elements in a horizontal row.
- Box: Overlays elements on top of each other.
- LazyColumn and LazyRow: Efficiently render lists, rendering only the visible items.

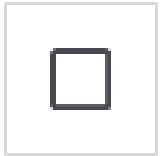
```
@Composable
fun App() {
    Column {
      Greeting("Hello")
      Greeting("World")
    }
}
@Composable
fun Greeting(name: String) {
    Text(
        text = "Hello $name!",
    )
}
```

Column will put the composables in one column



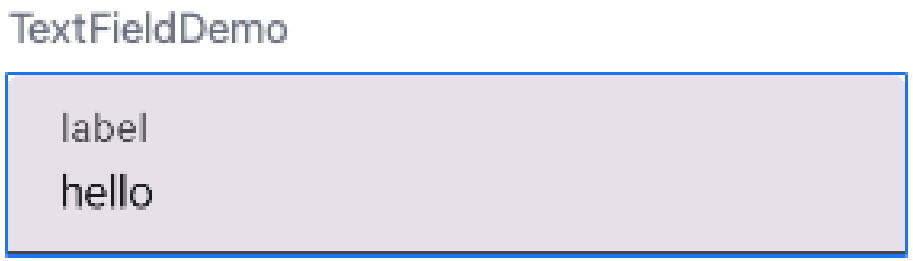
```kotlin
@Composable
@Preview
fun CheckboxDemo() {
    var state by remember { mutableStateOf(false) }
    Checkbox(
        checked = state,
        onCheckedChange = {
            state = it
        }
    )
}
```

CheckboxDemo



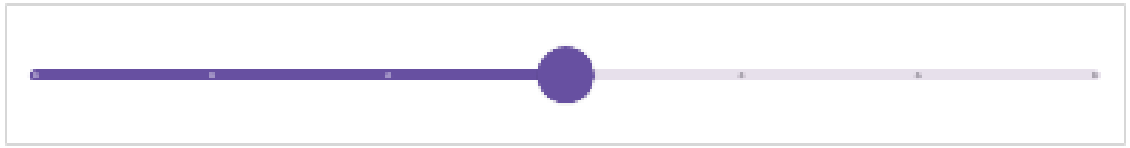
```kotlin
@Composable
@Preview
fun ButtonDemo() {
    var counter by remember { mutableStateOf(1) }
    Button(
        onClick = {
            counter += 1
        }
    ) {
        Text(
            text = counter.toString()
        )
    }
}
```

```
@Composable
@Preview
fun TextFieldDemo() {
    var text by remember { mutableStateOf("hello") }
    TextField(
        value = text,
        onValueChange = {
            text = it
        },
        label = { Text("label") },
        placeholder = { Text("placeholder") }
    )
}
```

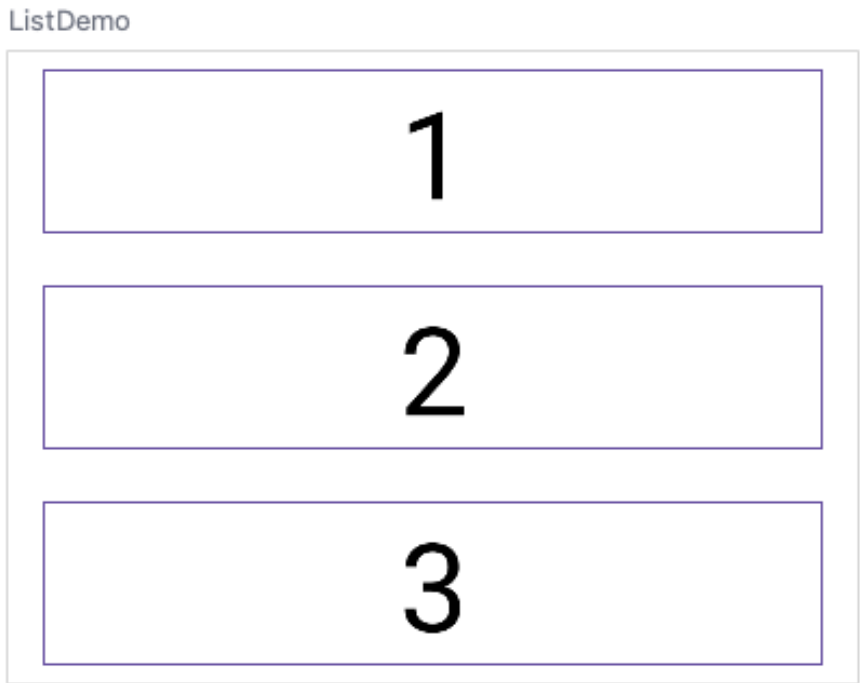


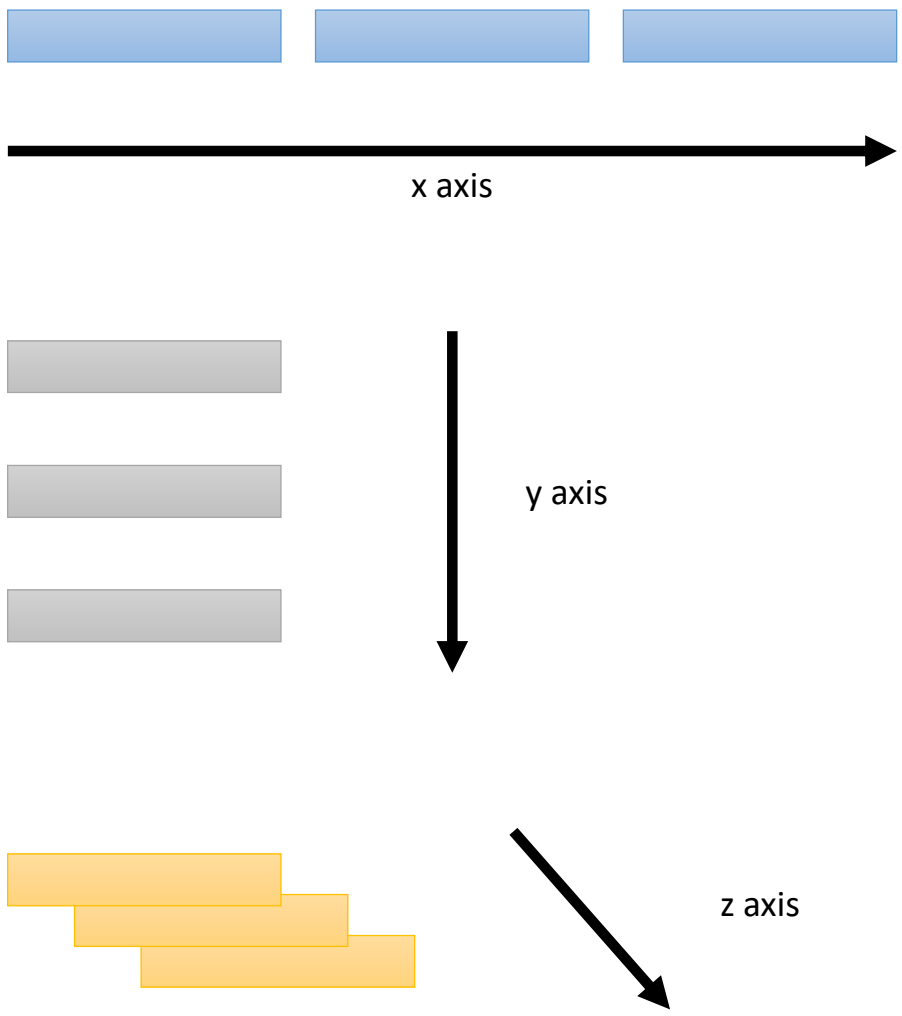
```
@Composable
@Preview
fun SliderDemo() {
    var value by remember { mutableStateOf(0.5F) }
    Slider(
        value = value,
        onValueChange = {
            value = it
        },
        steps = 5
    )
}
```

SliderDemo



```kotlin
@Composable
@Preview
fun ListDemo() {
    val list: List<Int> = (1..3).toList()
    LazyColumn(
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(
            items = list
        ) {
            Text(
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(vertical = 8.dp, horizontal = 16.dp)
                    .border(width = 1.dp, color = MaterialTheme.colorScheme.primary),
                text = it.toString(),
                style = MaterialTheme.typography.displayLarge,
                textAlign = TextAlign.Center
            )
        }
    }
}
```





Row()

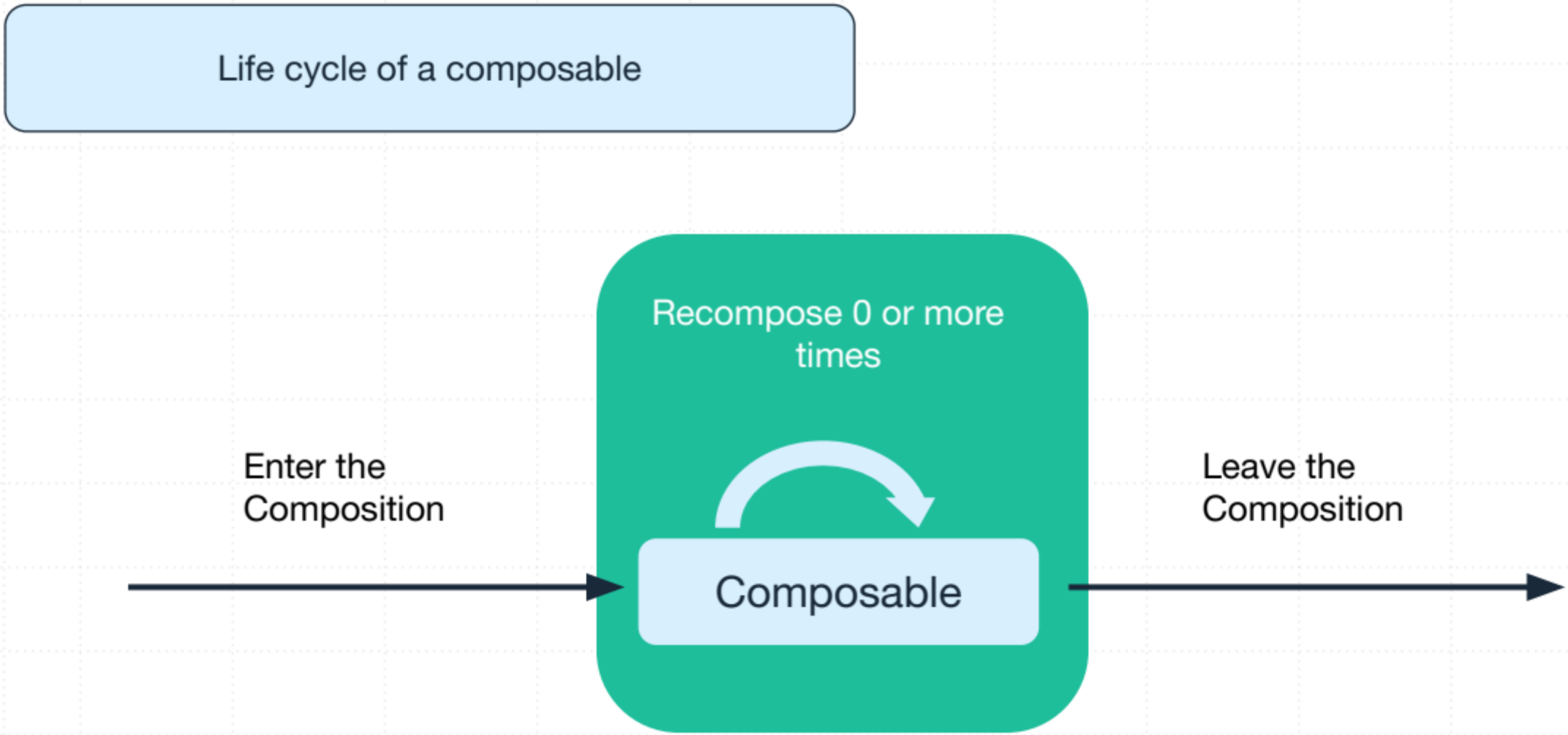
Column()

Box()

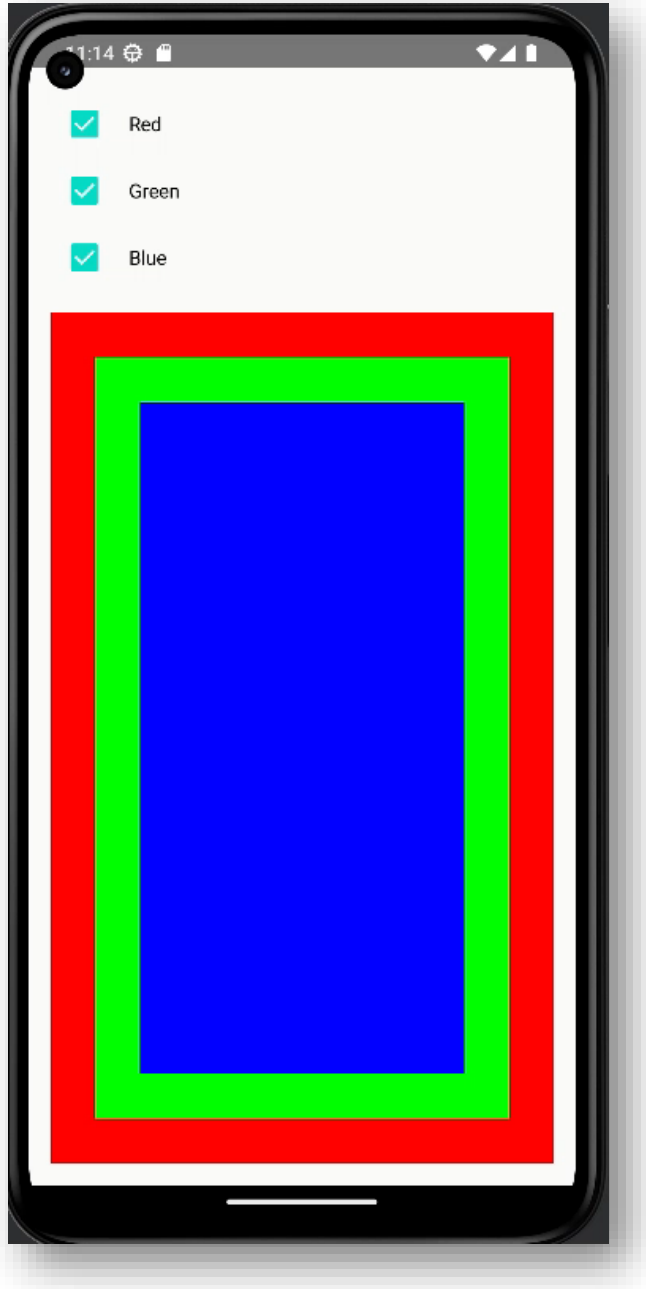
```kotlin
@Composable
fun CheckboxWithLabel(
    label: String,
    checked: Boolean,
    onClicked: (Boolean) -> Unit
) {
    Row(
        modifier = Modifier.clickable {
            onClicked(!checked)
        }, verticalAlignment = Alignment.CenterVertically
    ) {
        Checkbox(
            checked = checked,
            onCheckedChange = {
                onClicked(it)
            }
        )
        Text(
            text = label,
            modifier = Modifier.padding(start = 8.dp)
        )
    }
}
```

```
Column(
    modifier = Modifier
        .fillMaxSize()
        .padding(16.dp)
) {
    CheckboxWithLabel(
        label = stringResource(id = R.string.red),
        checked = red,
        onClicked = { red = it }
    )
    CheckboxWithLabel(
        label = stringResource(id = R.string.green),
        checked = green,
        onClicked = { green = it }
    )
    CheckboxWithLabel(
        label = stringResource(id = R.string.blue),
        checked = blue,
        onClicked = { blue = it }
    )
    ...
}
```

```
Box(
    modifier = Modifier
        .fillMaxSize()
        .padding(top = 16.dp)
) {
    if (red) {
        Box(
            modifier = Modifier.fillMaxSize()
                .background(Color.Red)
        )
    }
    if (green) {
        Box(
            modifier = Modifier.fillMaxSize()
                .padding(32.dp).background(Color.Green)
        )
    }
    if (blue) {
        Box(
            modifier = Modifier.fillMaxSize()
                .padding(64.dp).background(Color.Blue)
        )
    }
}
```


Life cycle of a composable
Recompose 0 or more times
Enter the Composition
Composable
Leave the Composition

- Write an app with checkboxes that show and hide colored boxes
- Use `Row()` to combine `Checkbox()` and `Text()`
- `Column()` arranges the three labeled checkboxes and one `Box()` containing the colored boxes



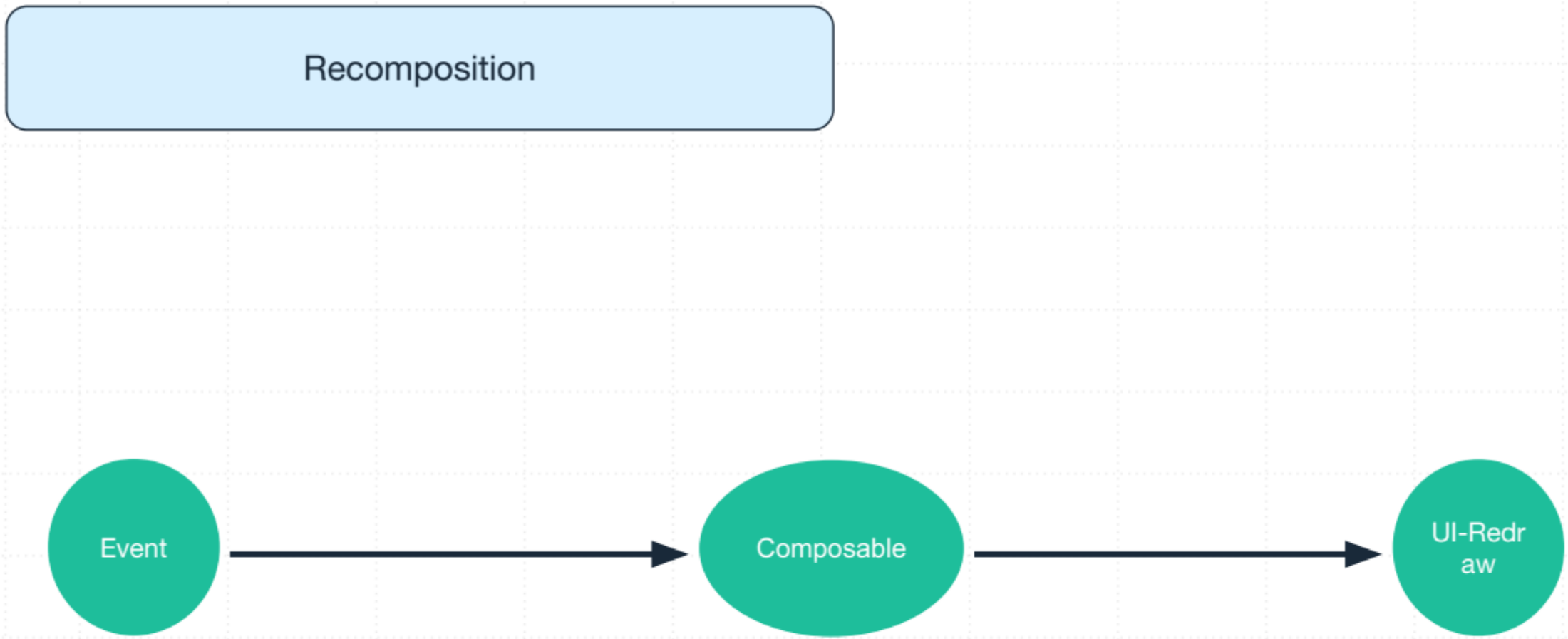
What's the State?

State

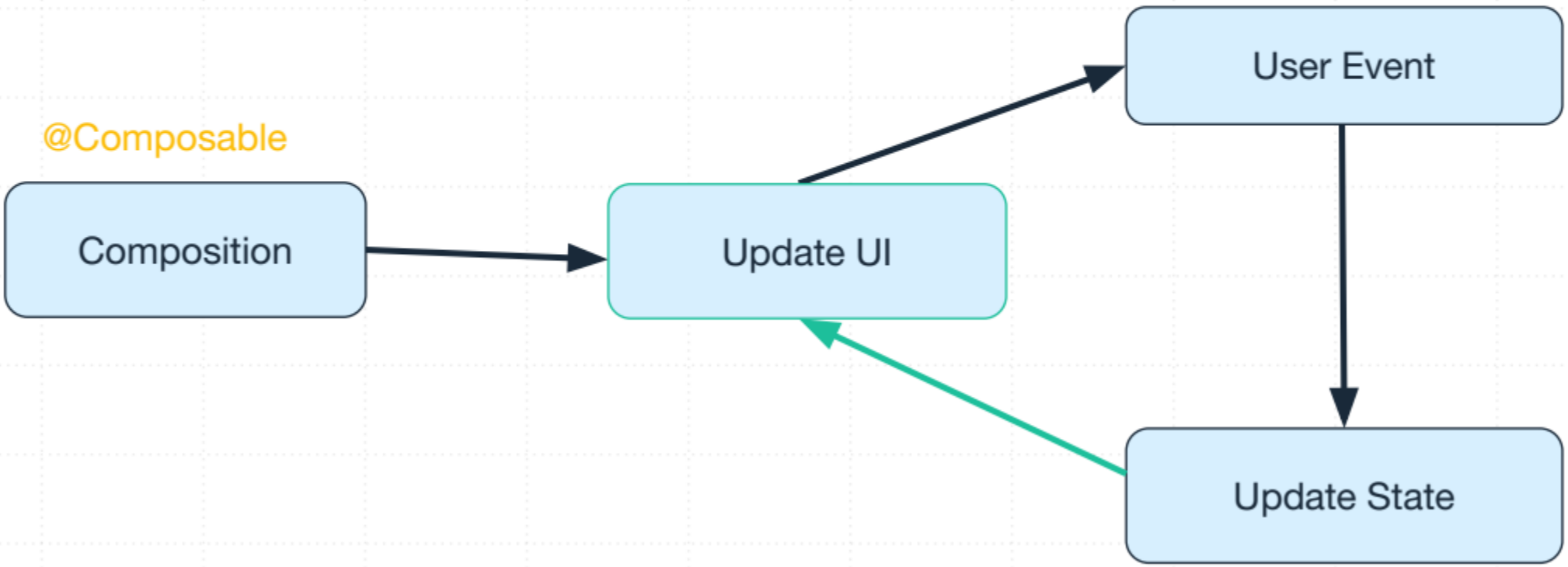
An interface that has a value property during the execution of a Composable function.

MutableState

A mutable value holder where reads to the value property during the execution of a Composable function. Any changes to value will schedule recomposition of any composable functions that read value.


Recomposition
Event
Composable
UI-Redr
aw

State & Recomposition



What's the State?

Value Holder

Observable

Recomposition Trigger

What's the State?

Remember

Remember Saveable

What’s the State?

StateFull

State less

Why state less?

Single source of truth

Encapsulated

Decoupled

Side effects

A **side-effect** is a change to the state of the app that happens outside the scope of a composable function.

Used to trigger one of events

Should be called with the scope of composable lifecycle

Side effects

- LaunchedEffect
- DisposableEffect
- DerivedStateOf

```
@Composable
fun TodoList(highPriorityKeywords: List<String> = listOf("Review", "Unblock", "Compose")) {

    val todoTasks = remember { mutableStateListOf<String>() }

    // Calculate high priority tasks only when the todoTasks or highPriorityKeywords
    // change, not on every recomposition
    val highPriorityTasks by remember(highPriorityKeywords) {
        derivedStateOf { todoTasks.filter { it.containsWord(highPriorityKeywords) } }
    }

    Box(Modifier.fillMaxSize()) {
        LazyColumn {
            items(highPriorityTasks) { /* ... */ }
            items(todoTasks) { /* ... */ }
        }
        /* Rest of the UI where users can add elements to the list */
    }
}
```

- **State** is data that may change over time
- A Compose user interface hierarchy is declared based on state
- When the data changes, the UI needs to be updated
- State can be passed to a composable function as a parameter, or it can be defined inside the composable

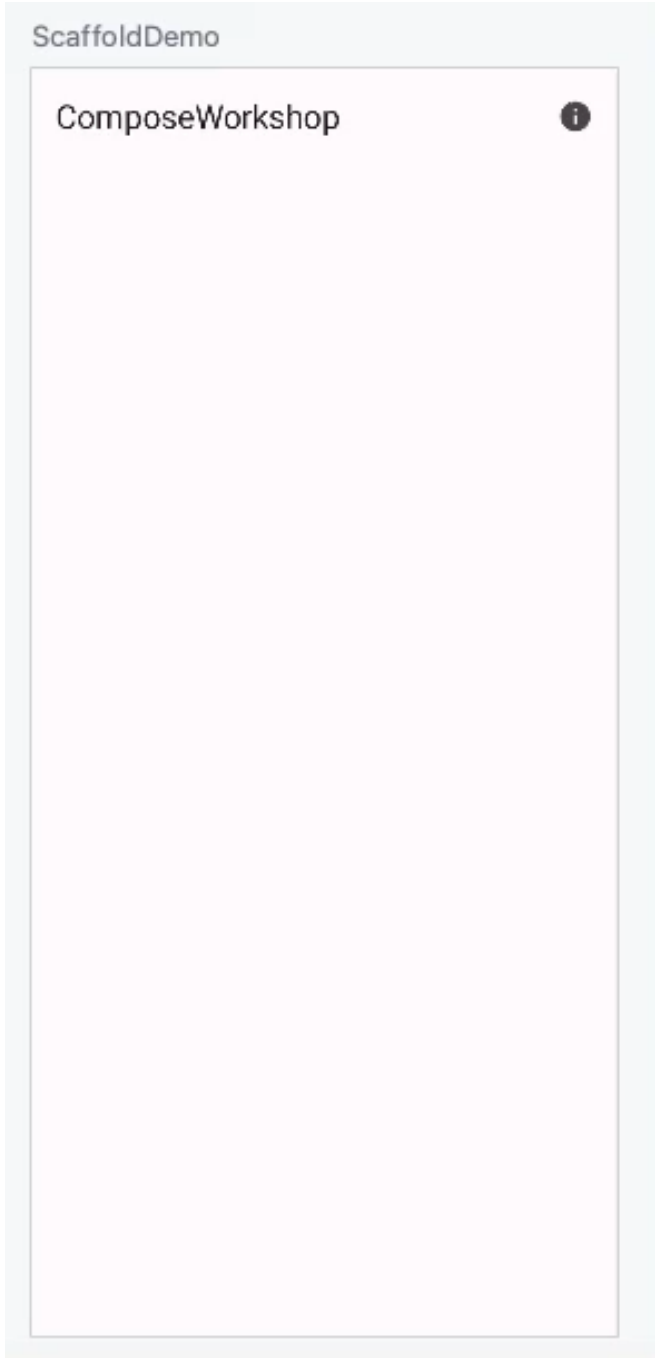
- Two types of composables
  - **stateful**
  - **stateless** (preferred)
- Jetpack Compose favors a unidirectional data flow and a concept called **state hoisting**

- State is defined as late as possible (depending on which composables will be using it)
- Data flows from top to bottom
- If possible, behavior is passed to a composable via callbacks

- Building the UI for the first time is called **composition**
- **State changes** trigger a **recomposition** of the UI
- Jetpack Compose makes sure only affected UI elements are redrawn

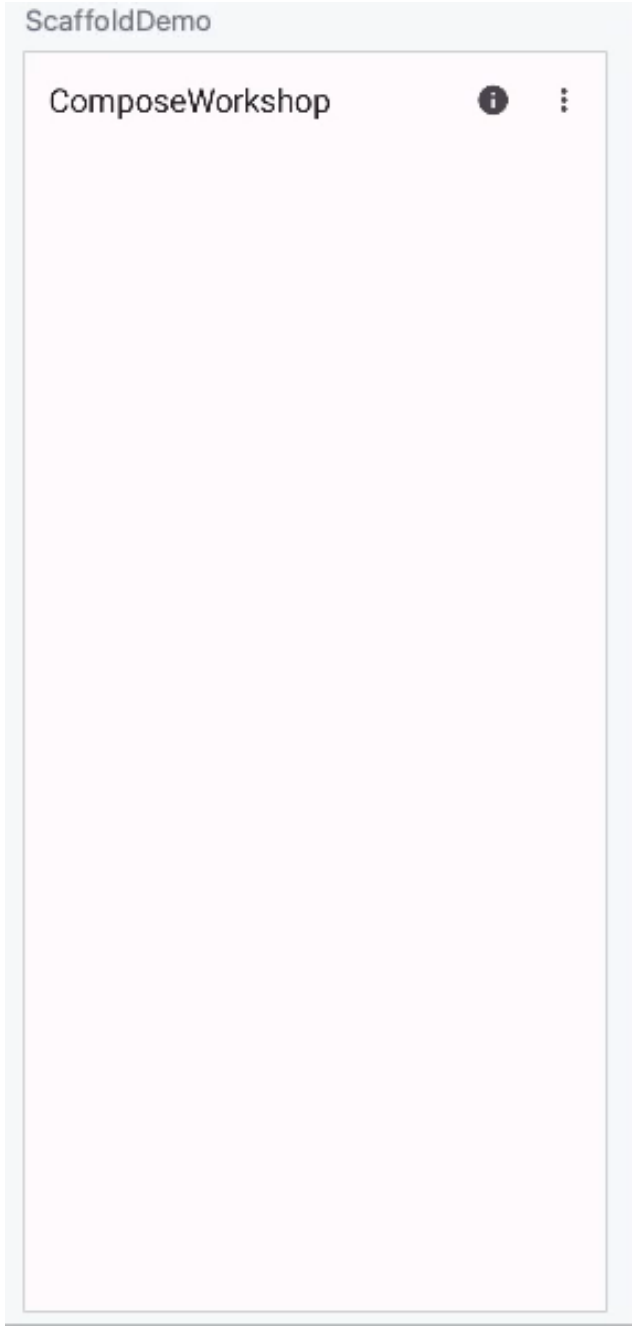
- We create state with `mutableStateOf()`
- We `remember` state
- We assign state to a variable to read and change it
- We pass behavior through callbacks
- We use state to declare the UI

```
@Composable
@Preview
fun CounterDemo() {
    var counter by remember { mutableStateOf(0) }
    Column(
        horizontalAlignment = CenterHorizontally,
        modifier = Modifier.fillMaxSize().padding(16.dp),
        verticalArrangement = Arrangement.Center
    ) {
        Box(
            contentAlignment = Center,
            modifier = Modifier.height(100.dp)
        ) {
            if (counter == 0) {
                Text(
                    text = "Noch nicht geklickt",
                    softWrap = true,
                    textAlign = TextAlign.Center,
                    style = MaterialTheme.typography.headlineMedium
                )
            } else {
                Text(
                    text = "$counter",
                    textAlign = TextAlign.Center,
                    style = MaterialTheme.typography.headlineLarge
                )
            }
        }
        Button(onClick = { counter += 1 }) {
            Text(text = "Klick")
        }
    }
}
```




```kotlin
@OptIn(ExperimentalMaterial3Api::class)
@Composable
@Preview
fun ScaffoldDemo(
    content: @Composable (PaddingValues) -> Unit = {}
) {
    Scaffold(topBar = {
        TopAppBar(
            title = {
                Text(
                    text = stringResource(id = R.string.app_name)
                )
            },
            actions = {
                IconButtonWithTooltip(
                    onClick = {},
                    painter = painterResource(id = R.drawable.baseline_info_24),
                    contentDescription = stringResource(id = R.string.info)
                )
            }
        )
    }) {
        content(it)
    }
}
```

```kotlin
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun IconButtonWithTooltip(
    onClick: () -> Unit,
    painter: Painter,
    contentDescription: String
) {
    PlainTooltipBox(tooltip = {
        Text(text = contentDescription)
    }) {
        IconButton(
            onClick = onClick,
            modifier = Modifier.tooltipAnchor()
        ) {
            Icon(
                painter = painter,
                contentDescription = contentDescription
            )
        }
    }
}
```



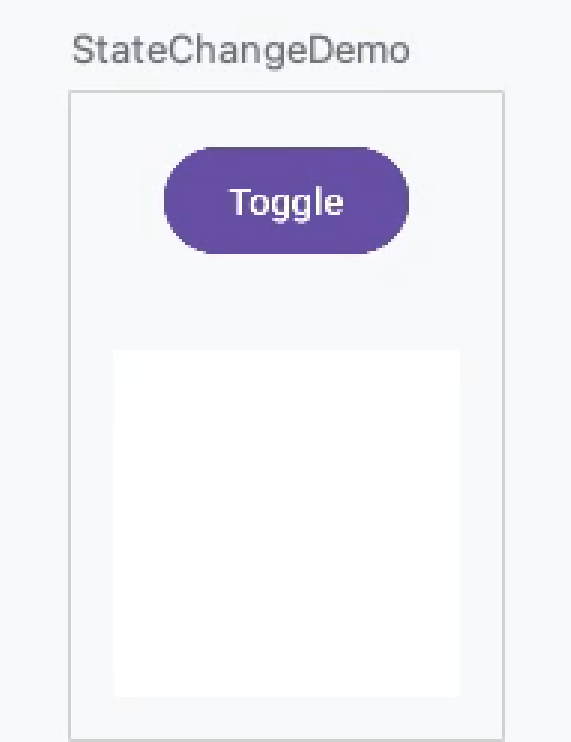

```kotlin
@Composable
fun MenuDemo(
    menuItems: List<String>,
    onClick: (Int) -> Unit
) {
    var expanded by remember { mutableStateOf(false) }
    Box {
        IconButton(onClick = {
            expanded = true
        }) {
            Icon(Icons.Default.MoreVert, stringResource(id = R.string.options_menu))
        }
        if (menuItems.isNotEmpty()) DropdownMenu(
            expanded = expanded,
            onDismissRequest = {
                expanded = false
            }
        ) {
            menuItems.forEachIndexed { index, s ->
                DropdownMenuItem(
                    onClick = {
                        expanded = false
                        onClick(index)
                    },
                    text = {
                        Text(s)
                    })
            }
        }
    }
}

```


```kotlin
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ModalBottomSheetDemo(dismissed: () -> Unit) {
    val modalBottomSheetState = rememberModalBottomSheetState()
    ModalBottomSheet(
        onDismissRequest = dismissed,
        sheetState = modalBottomSheetState,
        dragHandle = { BottomSheetDefaults.DragHandle() },
    ) {
        Box(
            modifier = Modifier
                .fillMaxSize()
                .background(color = MaterialTheme.colorScheme.secondaryContainer),
            contentAlignment = Center
        ) {
            Text(text = stringResource(id = R.string.app_name))
        }
    }
}
```

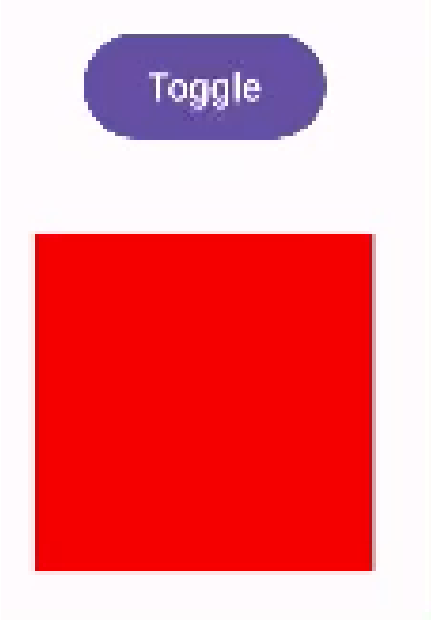
```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
@Preview
fun ScaffoldDemo(
    content: @Composable (PaddingValues) -> Unit = {}
) {
    var modalBottomSheetOpen by remember { mutableStateOf(false) }
    Scaffold( ... )
    }) {
        content(it)
        if (modalBottomSheetOpen) ModalBottomSheetDemo {
            modalBottomSheetOpen = false
        }
    }
}
```

Animations

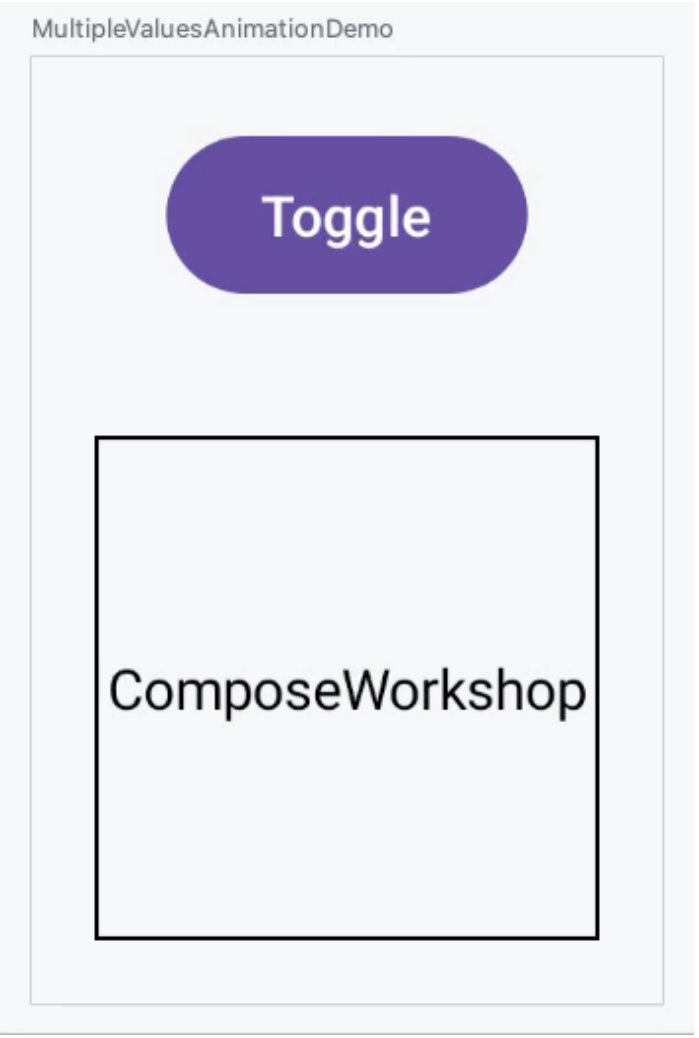
- Have become very important
- Not easy to achieve using `Views`
- But seamlessly integrated into Jetpack Compose



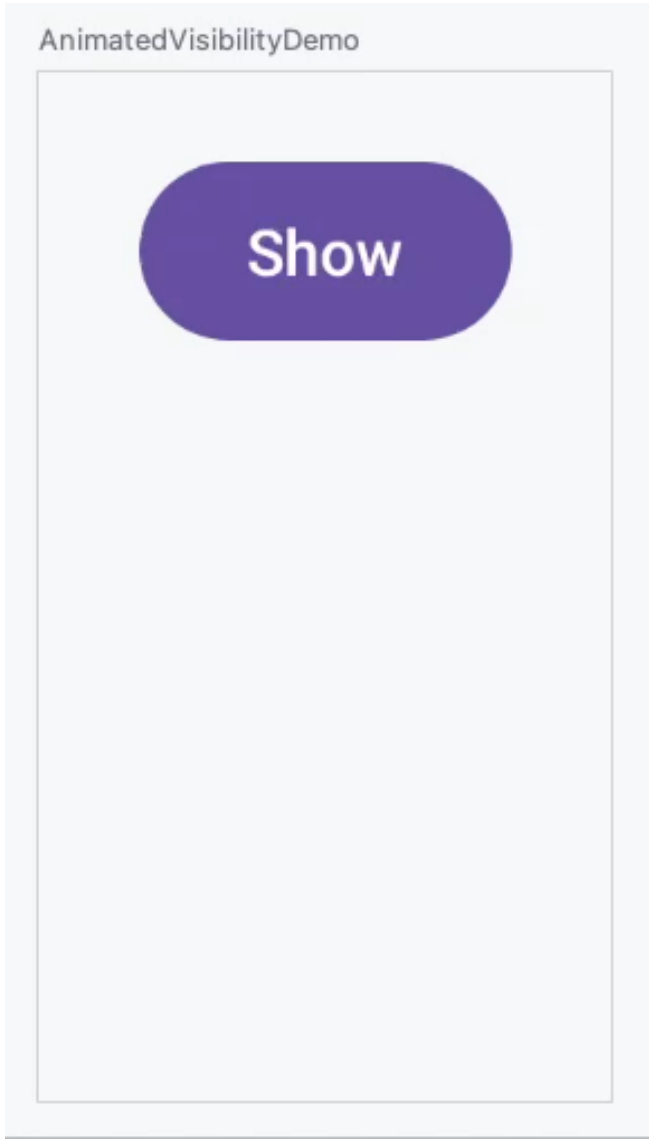

```
@Preview
@Composable
fun StateChangeDemo() {
    var toggled by remember {
        mutableStateOf(false)
    }
    val color = if (toggled) Color.White else Color.Red
    Column(
        modifier = Modifier
            .padding(16.dp),
        horizontalAlignment = CenterHorizontally
    ) {
        Button(onClick = {
            toggled = !toggled
        }) {
            Text(stringResource(R.string.toggle))
        }
        Box(
            modifier = Modifier
                .padding(top = 32.dp)
                .background(color = color)
                .size(128.dp)
        )
    }
}
```



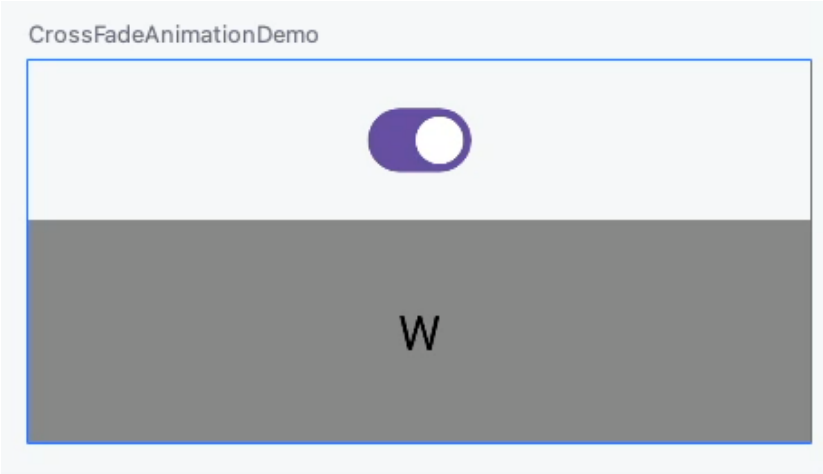

```
@Composable
@Preview
fun SingleValueAnimationDemo() {
    var toggled by remember {
        mutableStateOf(false)
    }
    val color by animateColorAsState(
        targetValue = if (toggled) Color.White else Color.Red,
        animationSpec = spring(stiffness = Spring.StiffnessVeryLow)
    )
    Column(
        modifier = Modifier
            .padding(16.dp),
        horizontalAlignment = CenterHorizontally
    ) {
        Button(onClick = {
            toggled = !toggled
        }) {
            Text(stringResource(R.string.toggle))
        }
        Box(
            modifier = Modifier
                .padding(top = 32.dp)
                .background(color = color)
                .size(128.dp)
        )
    }
}
```




```kotlin
@Composable
@Preview
fun MultipleValuesAnimationDemo() {
    var toggled by remember { mutableStateOf(false) }
    val transition = updateTransition(targetState = toggled)
    val borderWidth by transition.animateDp(label = "borderWidthTransition") { state ->
        if (state) 10.dp else 1.dp
    }
    val degrees by transition.animateFloat(label = "degreesTransition") { state ->
        if (state) -90F else 0F
    }
    Column(
        modifier = Modifier.padding(16.dp),
        horizontalAlignment = CenterHorizontally
    ) {
        Button(onClick = { toggled = !toggled }) {
            Text(stringResource(R.string.toggle))
        }
        Box(
            contentAlignment = Center,
            modifier = Modifier
                .padding(top = 32.dp)
                .border(
                    width = borderWidth,
                    color = Color.Black
                )
                .size(128.dp)
        ) {
            Text(
                text = stringResource(id = R.string.app_name),
                modifier = Modifier.rotate(degrees = degrees)
            )
        }
    }
}
```




```kotlin
@Composable
@Preview(widthDp = 128, heightDp = 230)
fun AnimatedVisibilityDemo() {
    var visible by remember { mutableStateOf(false) }
    Column(
        modifier = Modifier.padding(16.dp),
        horizontalAlignment = CenterHorizontally
    ) {
        Button(onClick = {
            visible = !visible
        }) {
            Text(
                stringResource(
                    id = if (visible) R.string.hide else R.string.show
                )
            )
        }
        AnimatedVisibility(
            visible = visible,
            enter = slideInHorizontally(),
            exit = slideOutVertically()
        ) {
            Box(
                modifier = Modifier
                    .padding(top = 32.dp)
                    .background(color = Color.Red)
                    .size(128.dp)
            )
        }
    }
}
```




```kotlin
@Preview
@Composable
fun CrossFadeAnimationDemo() {
    var isFirstScreen by remember { mutableStateOf(true) }
    Column(
        modifier = Modifier
            .fillMaxWidth()
            .height(192.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Switch(
            checked = isFirstScreen,
            onCheckedChange = {
                isFirstScreen = !isFirstScreen
            },
            modifier = Modifier.padding(top = 16.dp, bottom = 16.dp)
        )
        Crossfade(targetState = isFirstScreen) {
            if (it) {
                Screen(
                    text = stringResource(id = R.string.letter_w),
                    backgroundColor = Color.Gray
                )
            } else {
                Screen(
                    text = stringResource(id = R.string.letter_i),
                    backgroundColor = Color.LightGray
                )
            }
        }
    }
}
```

```
@Composable
fun Screen(
    text: String,
    backgroundColor: Color = Color.White
) {
    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(color = backgroundColor),
        contentAlignment = Alignment.Center
    ) {
        Text(
            text = text,
            style = MaterialTheme.typography.headlineSmall
        )
    }
}
```

Sample Questions

- Write a Jetpack Compose function that displays a simple Image logo in the center of the screen.
- Write a Jetpack Compose function that displays a message “Mobile Application Development” in the center of the screen.
- Write a Jetpack Compose function that displays list with number 1, 2, 3 in the center of the screen.
- Write down Jetpack stateful composable functions that maintain value of a text input field and updates the displayed text as the user types in text input field.
- Write down about Composable functions in Jetpack Compose
- Differentiate Traditional XML Layout of Android UI development vs UI created by Jetpack Compose.
- Differentiate Stateful composable vs Stateless Composable.
- Create two Jetpack composable functions: one is stateful composable function which keep tracks of counter value and counter value will be increment if button is clicked. Second is stateless composable function that displays the counter value as received in parameter.
- Write Jetpack composable functions that create login screen.
- Write Jetpack composable functions that create register screen.
- Write Jetpack composable functions that create book entry screen. (Book Title, Author of book, Price, Publisher).
- Write Jetpack composable functions that create MP3 player UI Screen.
- Write Jetpack Composable functions that implement simple list which displays name of students (“Student 1”, “Student 2”, “Student 3”,…)
- Write Jetpack Composable functions that shows list of types of books in dropdown.
- Write Jetpack Composable functions that creates screen for Vehicle Entry (Vehicle Model, Vehicle Manufacturer, Color, Price).
- Write Jetpack Composable functions that creates screen for restaurant menu Entry (Item name, Category, Price).

Sample Questions

- Design a Main Screen in Jetpack with a Button and a Text displaying either "Table Reserved" or "Reservation Cancelled". When the Button is clicked, toggle the text between the two.
- Create a Main Screen with a list of items and display them using LazyColumn.
- Design a UI to toggle between Light and Dark themes using a Button.
- Implement a TextField that updates a Text composable in real time as the user types.
- Create a checkbox list where each item toggles its selected state when clicked.
- Design a screen with a LazyRow to display horizontal scrolling images.
- Build a form with multiple TextFields (e.g., Name, Email) and validate input on submission.
- Implement a BottomNavigationBar with three tabs and display different content for each.
- Create a counter app with a Button to increase the count and display it in a Text.
- Design a dropdown menu to select an option and display the selected option in a Text.
- Implement a simple rating bar using clickable stars and display the selected rating.