CS 473 - MDP
Mobile Device Programming

Maharishi International University

CS 473 - MDP
Mobile Device Programming

MS.CS Program

Department of Computer Science

Renuka Mohanraj, Ph.D.

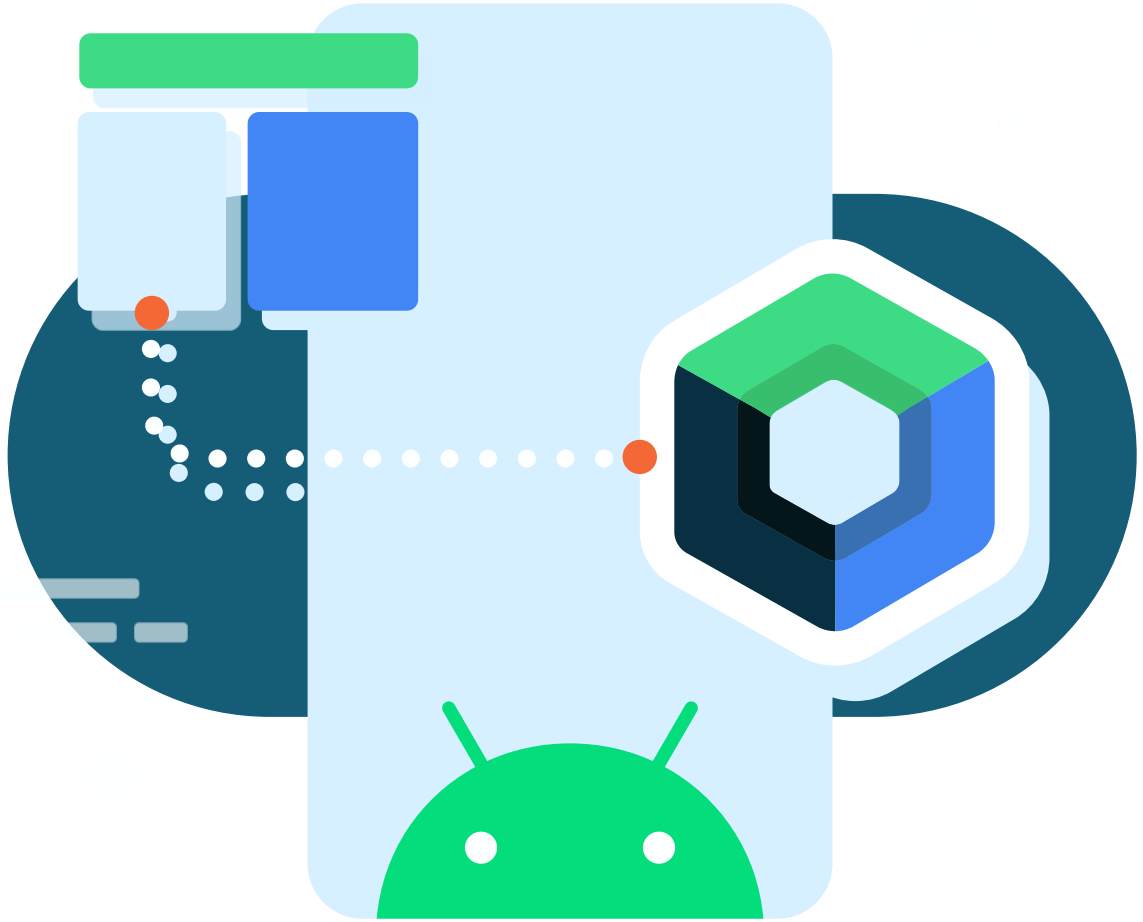
Maharishi International University

Course Overview Chart

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Week 1	Course Overview; **Lesson 1(a):** Kotlin Fundamentals	**Lesson 1(b) & 2:** More Kotlin Fundamentals & Introduction to Android	**Lesson 3:** Jetpack Compose Basics 	**Lesson 4:** Lists & Grid	**Lesson 5:** App Architecture and Dependency Injection	**Lesson 5 contd:** App Architecture and Dependency Injection
	Lesson 1(a): cont'd	Lab	Lab	Lab	Lab	
Week 2	Revision for Midterm	Midterm Exam	**Lesson 6:** Coroutine and Testing ViewModel with Coroutine	**Lesson 7:** Navigating Android Apps: Activities and Jetpack Navigation	**Lesson 7 contd:** Navigating Android Apps: Activities and Jetpack Navigation	**Lesson 8:** Full Stack App Development
			Lab/Project	Lab/Project	Lab/Project	
Week 3	**Lesson 8 cont'd:** Full Stack App Development	**Lesson 9:** Data Persistence	**Lesson 9 cont'd:** Data Persistence	**Lesson 10:** Work Manager	**Lesson 11:** Generative AI on Android	**Lesson 11 cont'd:** Generative AI on Android
	Lab/Project	Lab/Project	Lab/Project	Lab/Project	Lab/Project	Project
Week 4	Revision for Final	Final Exam	Project	Project Presentation	Project Presentation	Project Presentation
			Early Bird Project Presentation	Project Presentation	Project Presentation	Project Presentation

Lesson-3 Jetpack Compose Basics

Outer depends on inner



Wholeness

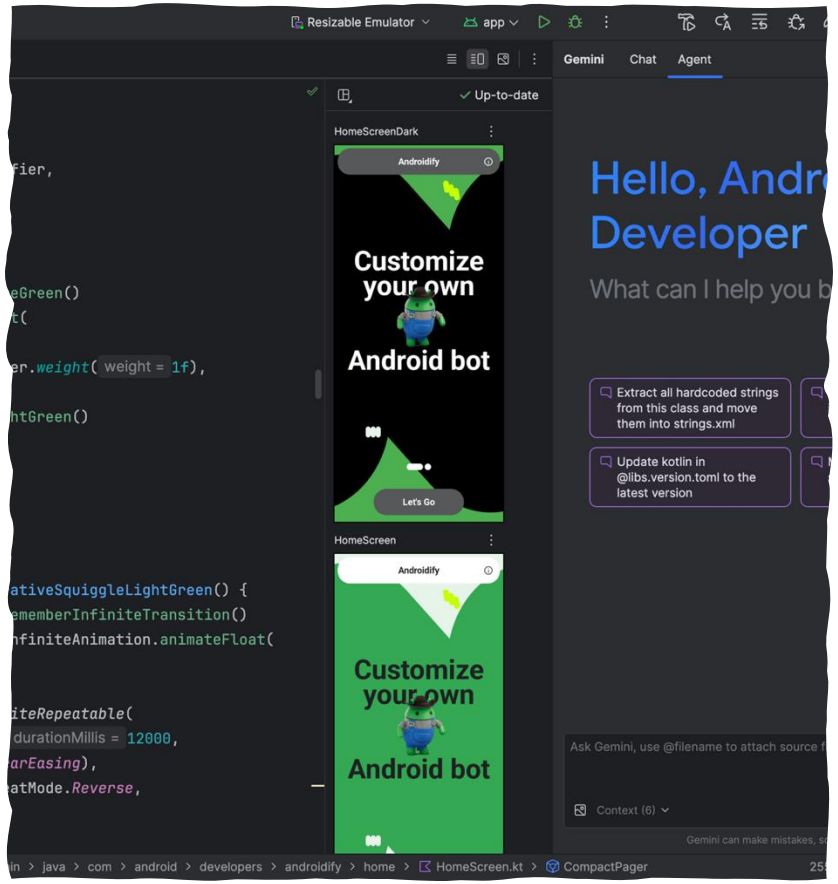
Android Jetpack Compose revolutionizes UI development in Android Studio, offering a seamless approach to building dynamic user interfaces. Through simple UIs and powerful modifiers, developers can create elegant and responsive apps with ease. Jetpack Compose simplifies UI code, allowing for faster development cycles and easier maintenance. Its declarative nature promotes code clarity and facilitates efficient collaboration among developers. Overall, Jetpack Compose empowers developers to craft engaging Android applications by streamlining UI design and development. *Science and Technology of Consciousness:* The principle "Outer depends on inner" is reflected in Jetpack Compose, where the visible user interface (outer) arises directly from the underlying application state (inner).

Agenda

- Introduction to Android Studio
- Jetpack Compose and its basics
- Android Project Structure, Folders and First Project Tour on IDE
- Composable functions
- AVD Manager
- Basic Layouts and simple UIs
- Birthday Wish APP Demo

Android Studio

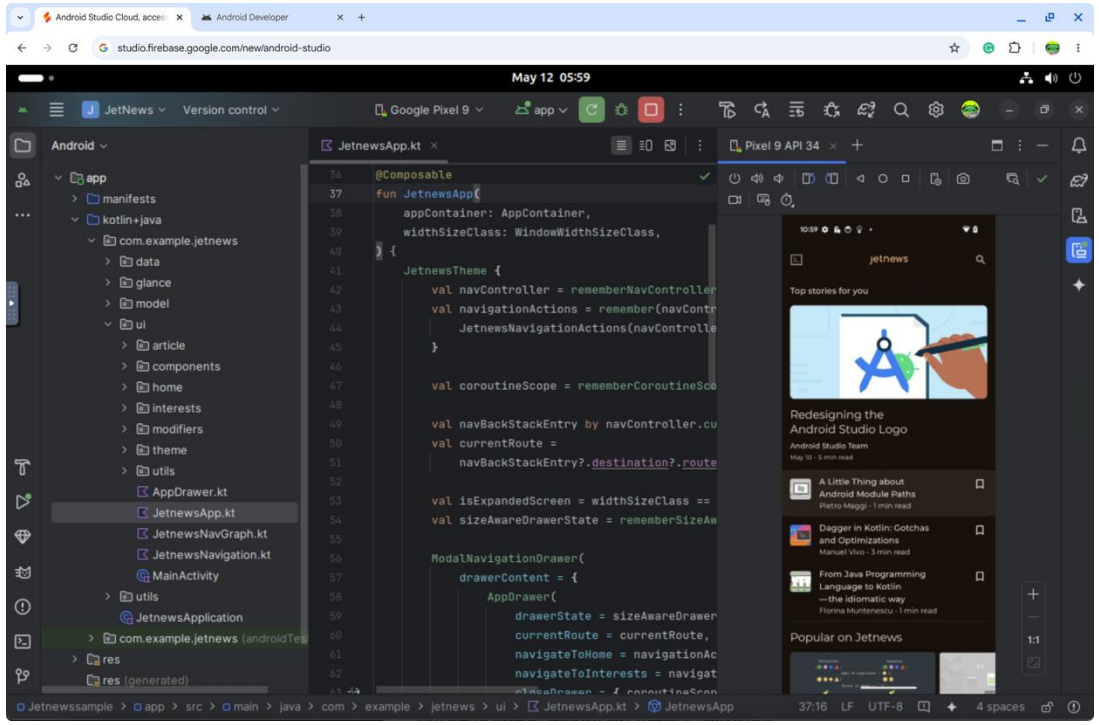
https://developer.android.com/studio



- Android Studio is the official Integrated Development Environment (IDE) for Android app development. Based on the powerful code editor and developer tools from IntelliJ IDEA.
- Android Studio offers even more features that enhance your productivity
  - A flexible Gradle-based build system
  - A fast and feature-rich emulator
  - A unified environment where you can develop for all Android devices
  - Live Edit to update composables in emulators and physical devices in real time
  - Extensive testing tools and frameworks

Android Studio Cloud

https://developer.android.com/studio/preview/android-studio-cloud



Activity

An activity is a single, standalone module of application functionality that either correlates directly to a single user interface screen and its corresponding functionality or acts as a container for a collection of related screens.

An appointments application might, for example, contain an activity screen that displays appointments set up for the current day.

Jetpack Compose: Modern UI toolkit

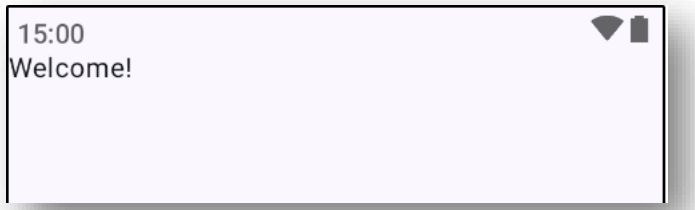
What is Compose?

- Composable functions (also referred to as composables or components) are special Kotlin functions that are used to create user interfaces when working with Compose.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {...}
```

- Compose offers a modern, declarative approach to building Activity UIs.

```
Text(text = "Welcome!")
```



Jetpack Compose: Modern UI toolkit

- Benefits:
  - Rapid development
  - Less code
  - Powerful, built-in support for Material Design, Dark theme, animations, and more
  - Declarative and composable syntax
    - Instead of dealing with XML layout files and Views, Compose allows you to create UI components in a declarative manner using Kotlin. This means you can define your UI by simply writing code, making UI development more intuitive, efficient, and less error-prone.
  - Enhanced performance

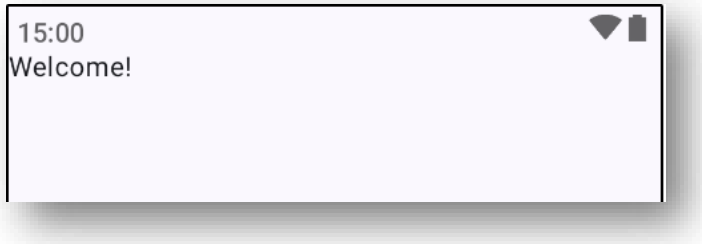
Jetpack Compose: Modern UI toolkit

Design using XML (Old Style)

```
<TextView
        android:id="@+id/text_view_id"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:text="Welcome!" />
```

Less code in Jetpack Compose

```
Text(text = "Welcome!")
```

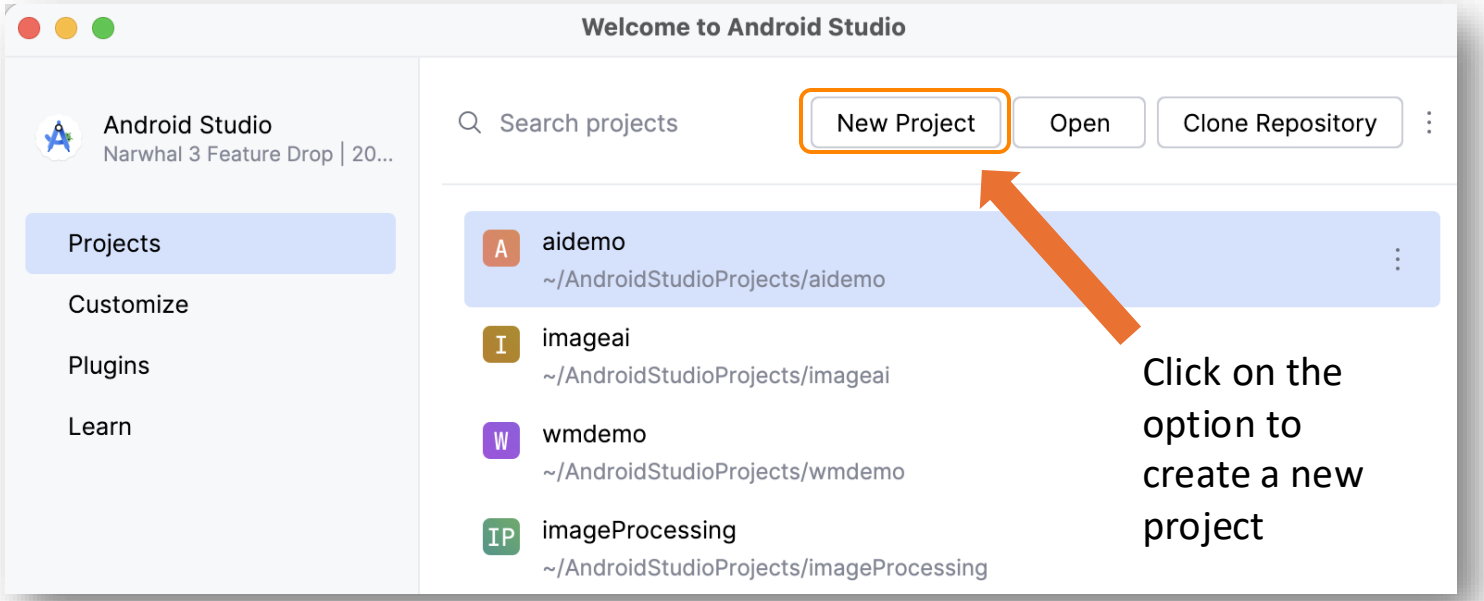


Create a new project

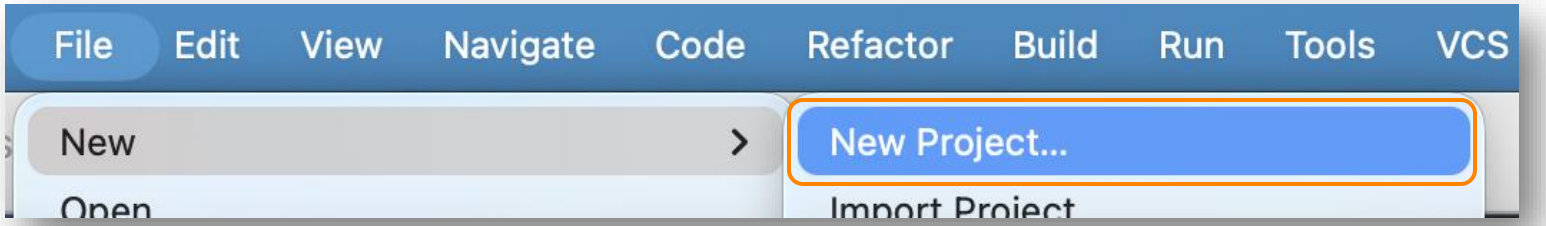
The Android Studio welcome screen gives you several options for what you want to do. We want to create a new project, so click on the option for “New Project”.

(or)

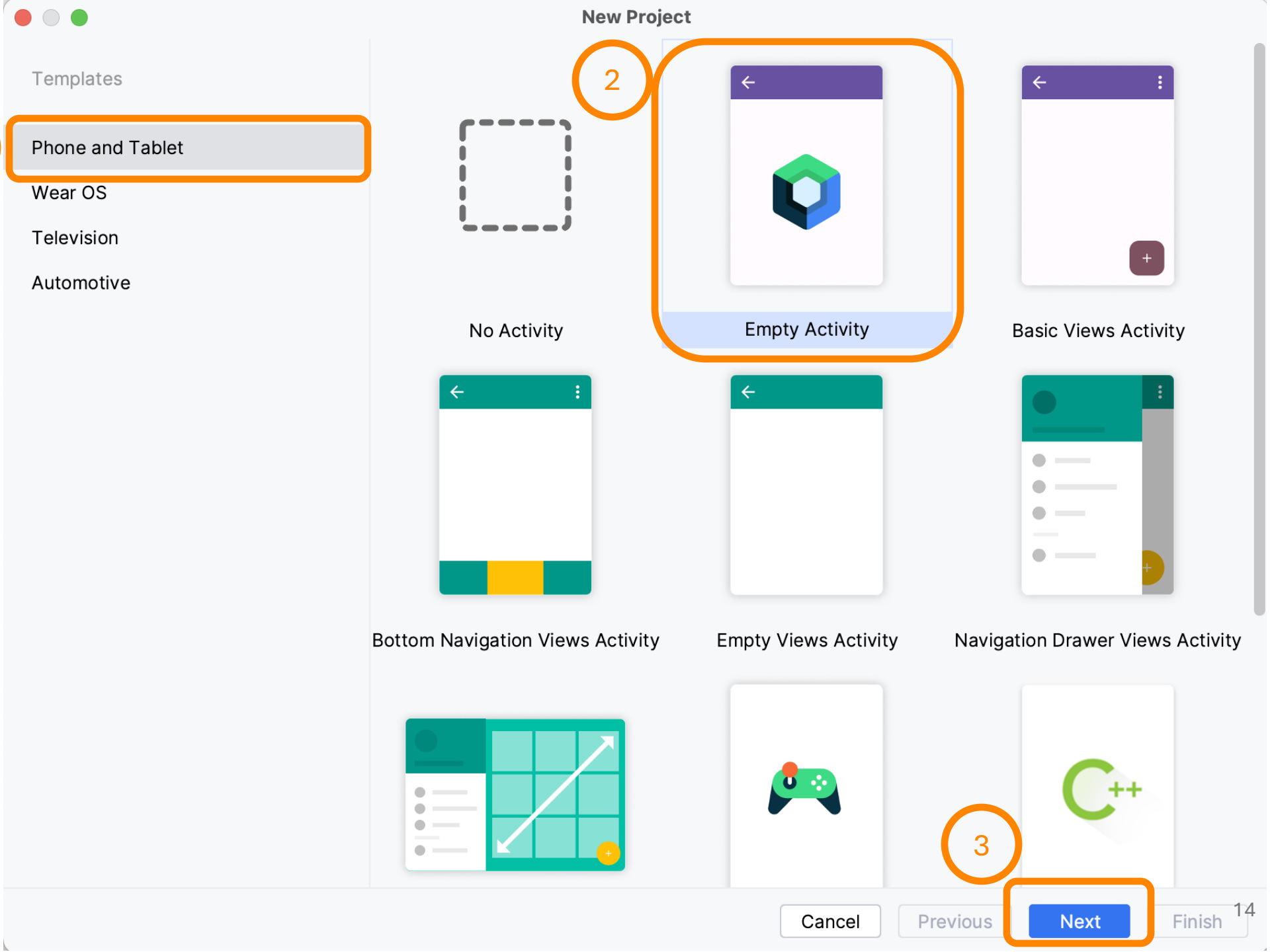
File -> New->New Project



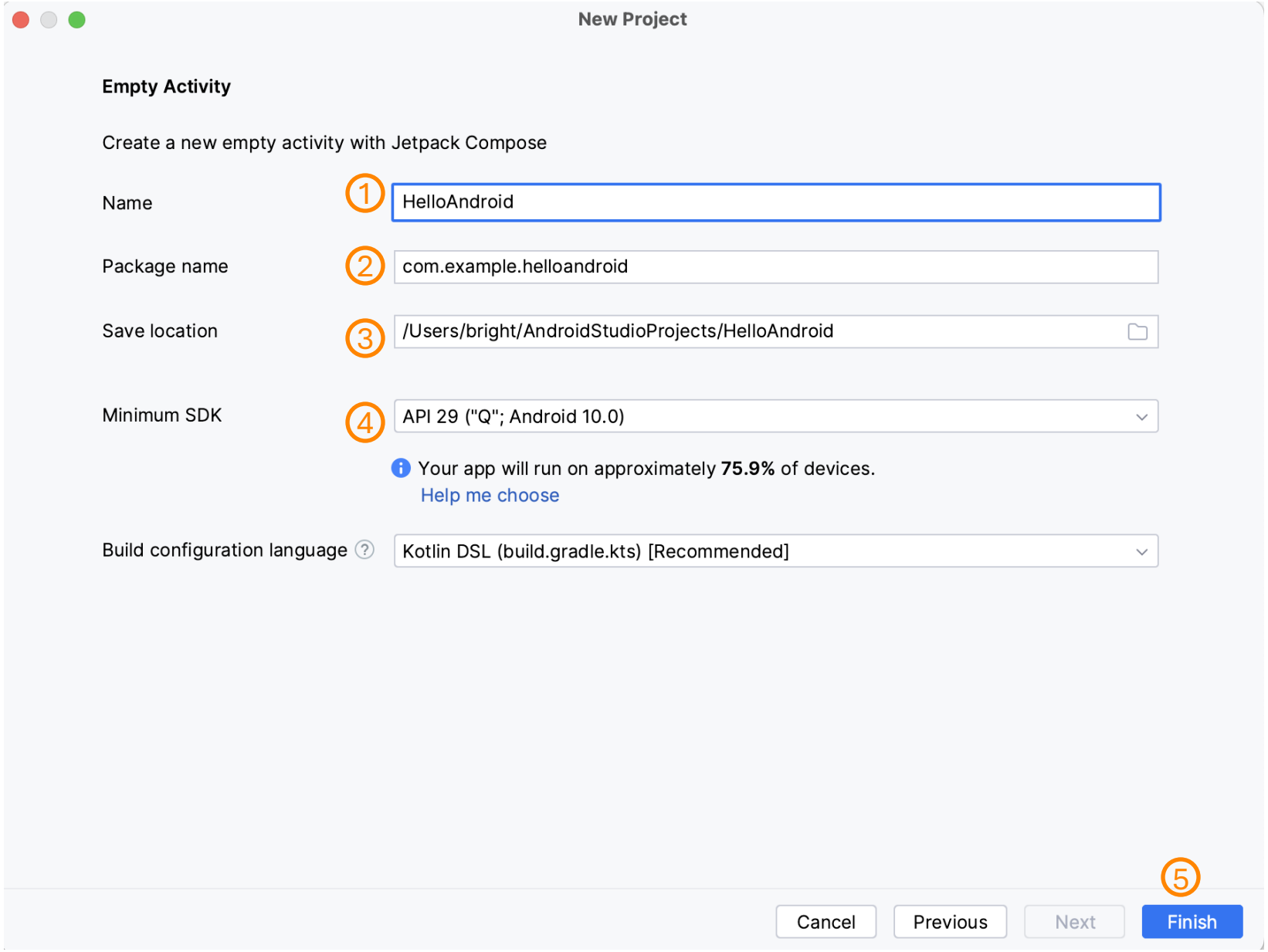
OR



1. Choose Phone and Tablet
2. Select Empty Activity
3. Click Next



Create a new project



1. Name of your Application shown in Google play store and other places.

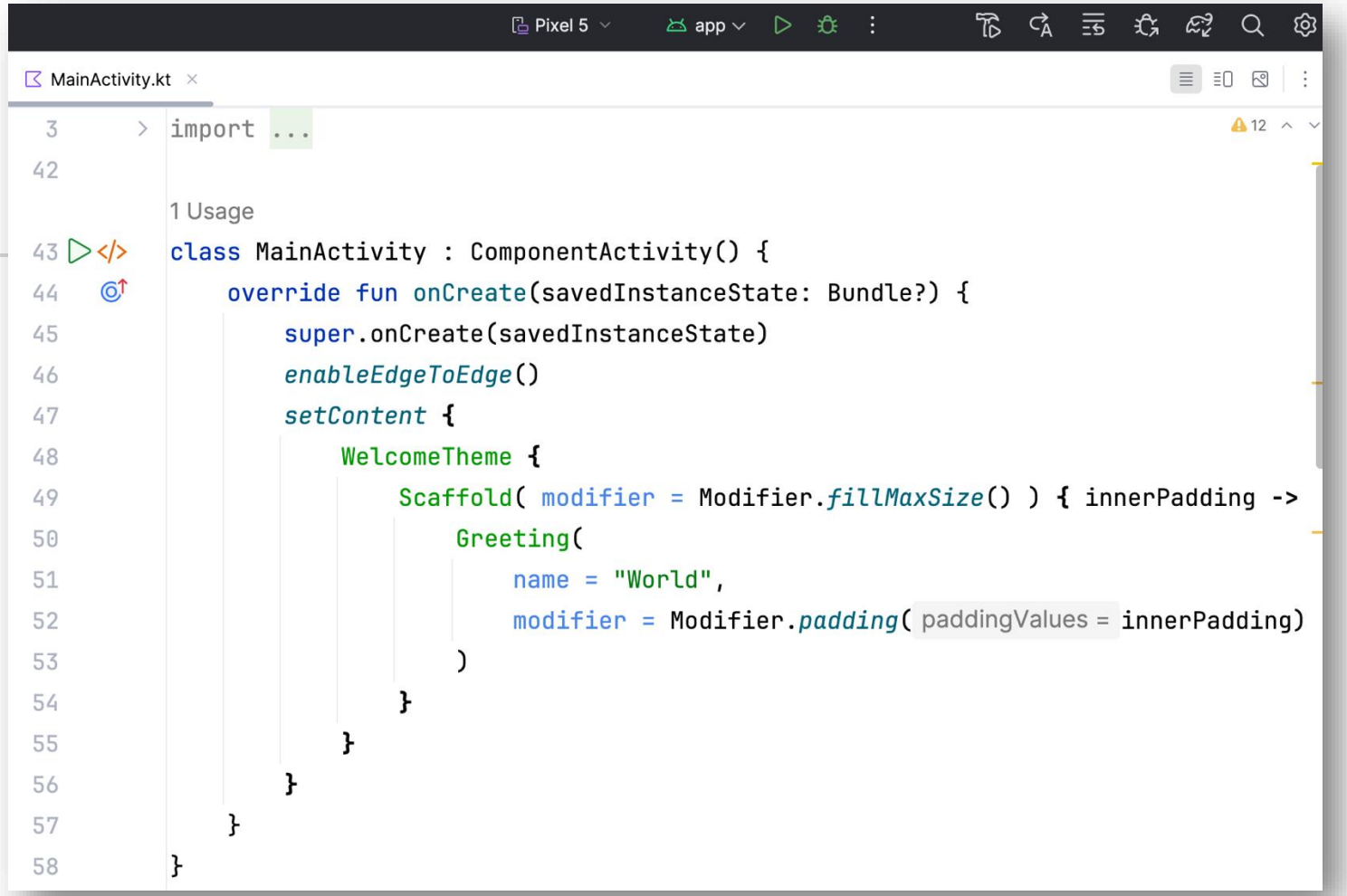
2. Forms the package name by combining application name and company name

3. All your files for your project will be stored here.

4. Choose Minimum API support of SDK.

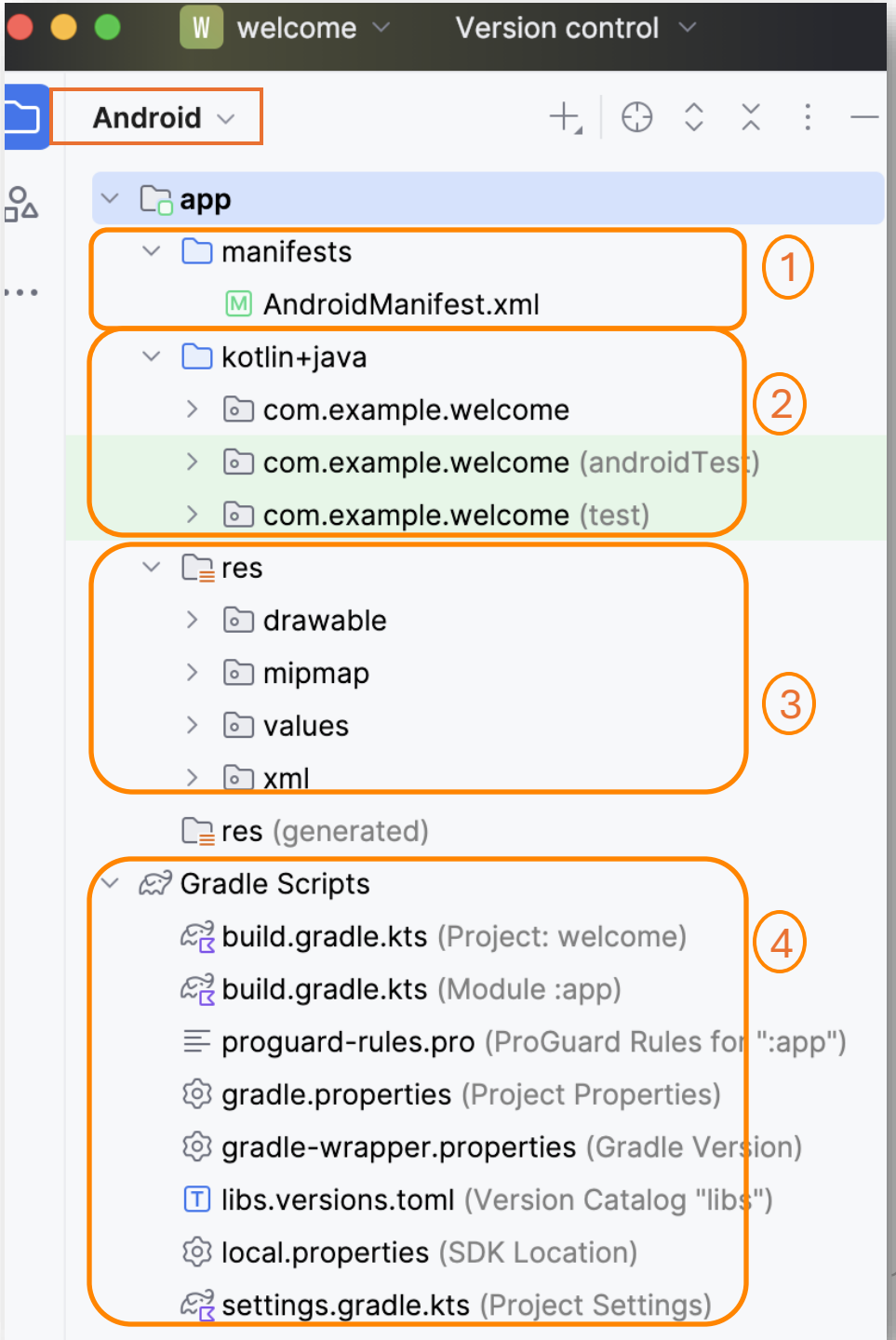
5. Click Finish

Here’s what our project looks like (do not worry if it looks complicated right now, we’ll break it down over the next few slides):

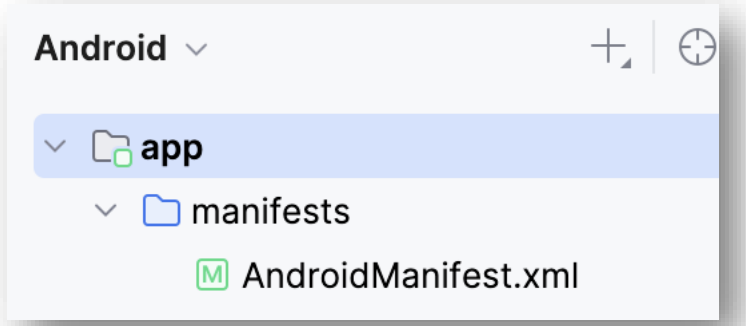


Project Folders

1. **manifests**
   - This folder contains AndroidManifest.xml. This file describes all the components of your Android app and is read by the Android runtime system when your app is executed.
2. **kotlin+java**
   - All your Kotlin and Java language files are organized here.
3. **res**
   - This folder contains all the resources for your app, including images, strings, icons, and styling.
4. **Gradle Scripts**
   - Shows all the project's build-related configuration files.



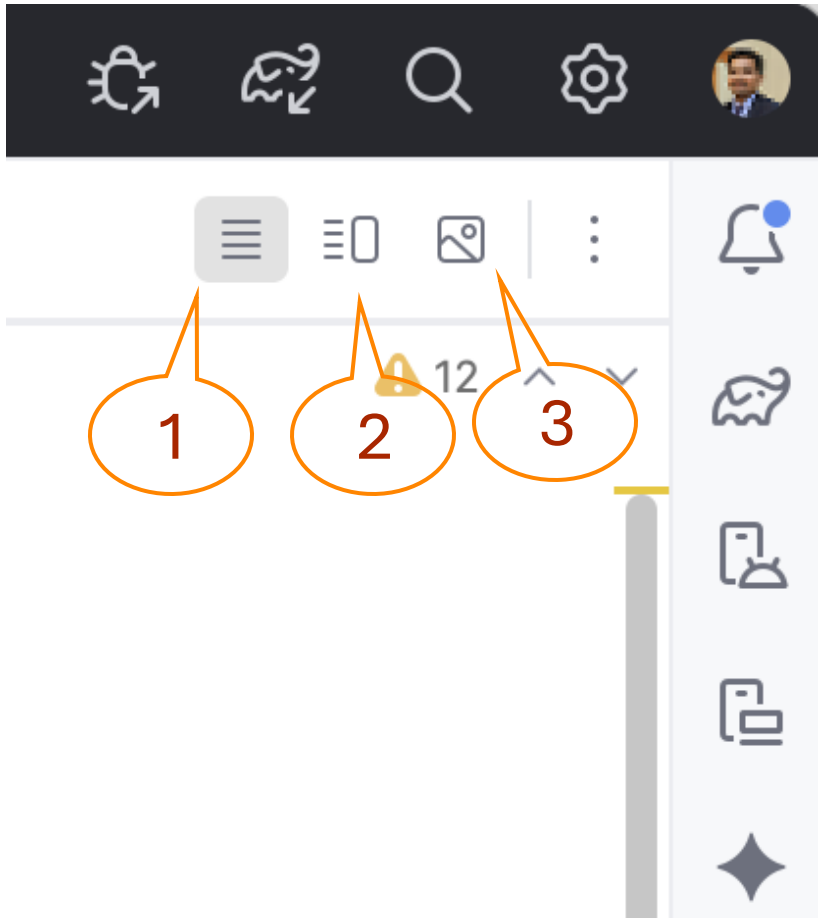
AndroidManifest.xml



- It is an XML file located at the root of your project and acts as a blueprint for the Android system, providing essential information about your app.
- Components:
  - Activities
    - Represent individual screens your app can display. The manifest lists all activities and their launch points.
  - Services
    - Run in the background to perform long-running tasks or handle system events. The manifest defines how they operate and interact with other components.
  - Broadcast Receivers
    - Respond to system events or broadcasts from other apps. The manifest specifies what events they listen to and how they react.
  - Content Providers
    - Manage and share app data with other apps. The manifest defines their access permissions and data handling rules.

Different Modes in Code Editor

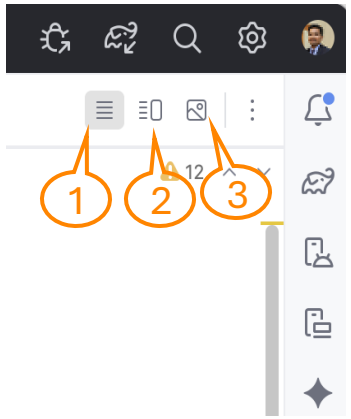
1. Code editor
2. Code editor with preview panel
3. Design mode



1. Code Mode in Activity

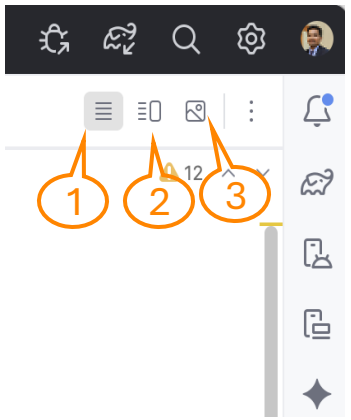
MainActivity.kt

```
3   import ...
42
    1 Usage
43  class MainActivity : ComponentActivity() {
44      override fun onCreate(savedInstanceState: Bundle?) {
45          super.onCreate(savedInstanceState)
46          enableEdgeToEdge()
47          setContent {
48              WelcomeTheme {
49                  Scaffold( modifier = Modifier.fillMaxSize() ) { innerPadding ->
50                      Greeting(
51                          name = "World",
52                          modifier = Modifier.padding( paddingValues = innerPadding)
53                      )
54                  }
55              }
56          }
57      }
58  }
```



1. Code editor
2. Code editor with preview panel
3. Design mode

2. Split Mode in Activity



Version control | Pixel 5 | app

MainActivity.kt

```
1   package com.example.welcome
2
3   import ...
15
16  class MainActivity : ComponentActivity() {
17      override fun onCreate(savedInstanceState: Bundle?) {
18          super.onCreate(savedInstanceState)
19          enableEdgeToEdge()
20          setContent {
21              WelcomeTheme {
22                  Scaffold(modifier = Modifier.fillMaxSize())
23                      Greeting(
24                          name = "Android",
25                          modifier = Modifier.padding( padding\
26                      )
27                  }
28              }
29          }
30      }
31  }
```

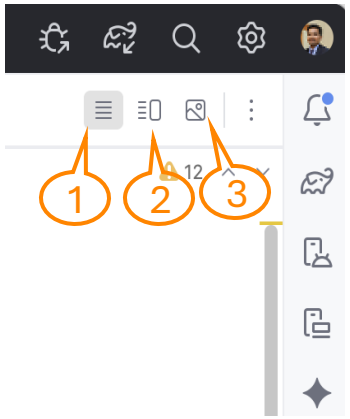
A successful build is needed before the prev

Build & Refresh... (⌥⇧⌘R)

1. Code editor
2. Code editor with preview panel
3. Design mode

If you see this notification, click on the Build & Refresh link to rebuild the project.

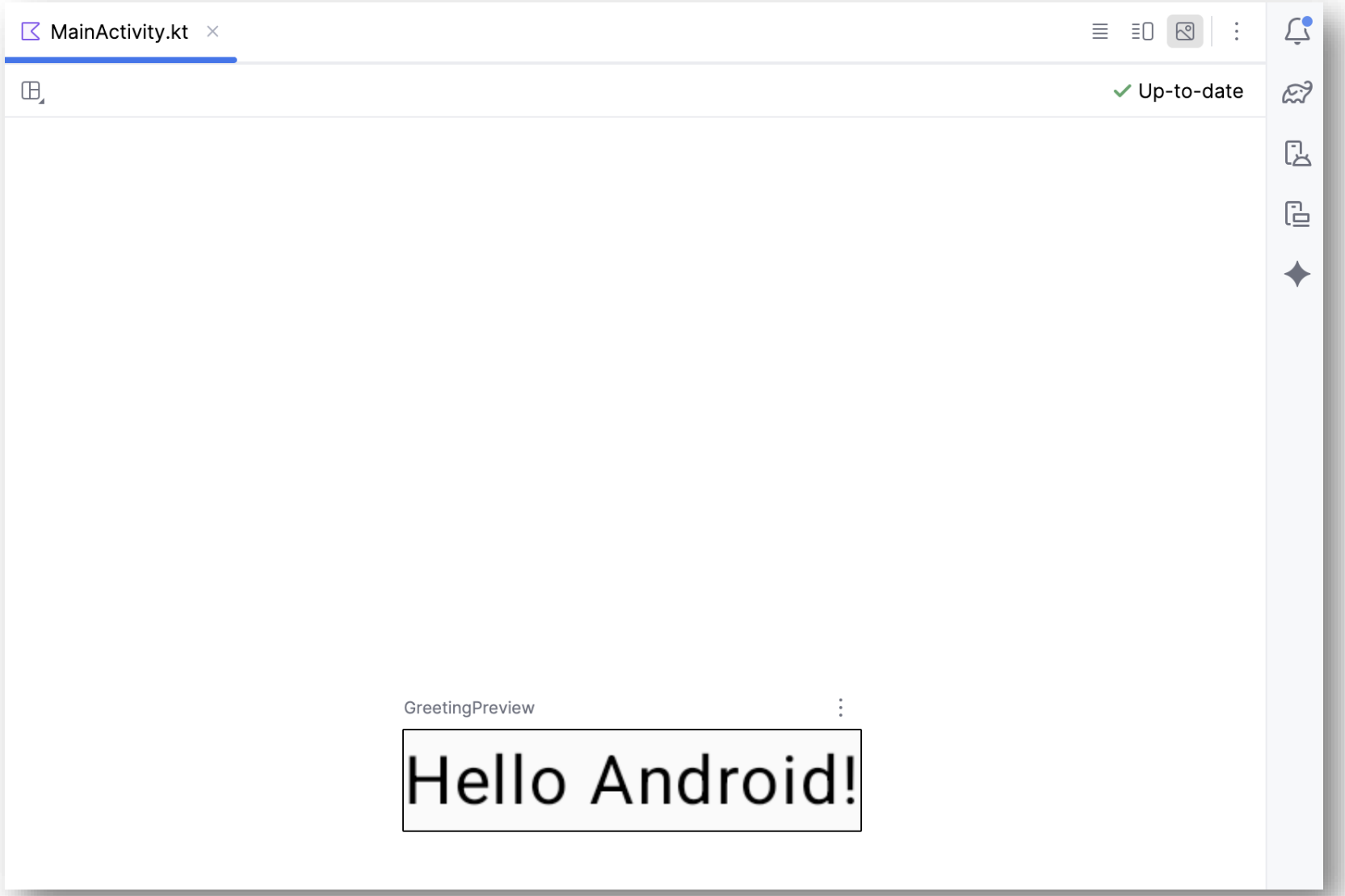
2. Split Mode in Activity

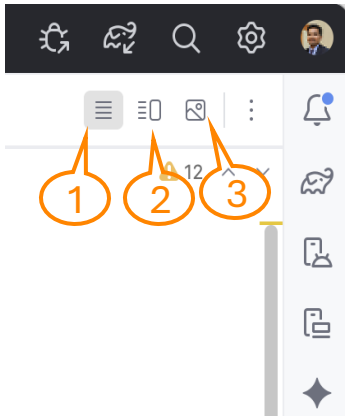


1. Code editor
2. Code editor with preview panel
3. Design mode

After the build is complete, the Preview panel should update to display the user interface defined by the code in the MainActivity.kt file

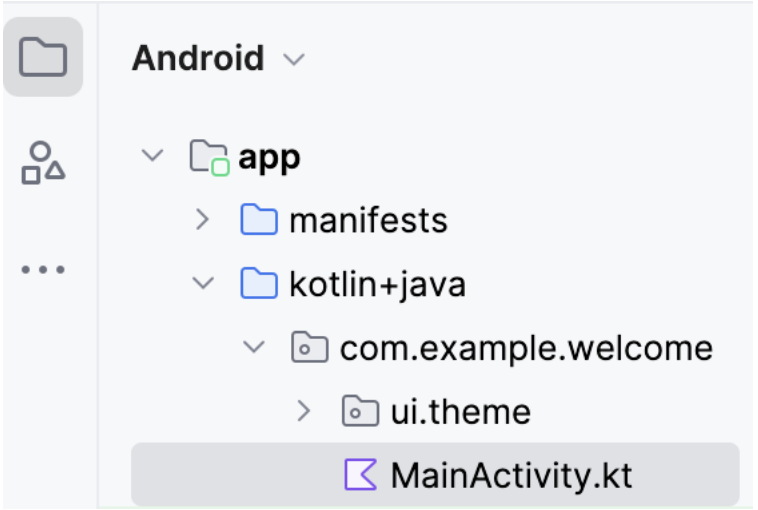
3. Design Mode in Activity





1. Code editor
2. Code editor with preview panel
3. Design mode

Reviewing the MainActivity



- Android applications are created by combining one or more components known as Activities.
- An activity is a single, standalone module of application functionality that either correlates directly to a single user interface screen and its corresponding functionality or acts as a container for a collection of related screens.
- Android Studio created a single initial activity for our app, named it MainActivity, and generated some code for it in the MainActivity.kt file.
- This activity contains the first screen that will be displayed when the app is run on a device.

Reviewing the MainActivity

It is a series of import directives from libraries provided by Android SDK.

MainActivity.kt

```
1   package com.example.welcome
2
3   import android.os.Bundle
4   import androidx.activity.ComponentActivity
5   import androidx.activity.compose.setContent
6   import androidx.activity.enableEdgeToEdge
7   import androidx.compose.foundation.layout.fillMaxSize
8   import androidx.compose.foundation.layout.padding
9   import androidx.compose.material3.Scaffold
10  import androidx.compose.material3.Text
11  import androidx.compose.runtime.Composable
12  import androidx.compose.ui.Modifier
13  import androidx.compose.ui.tooling.preview.Preview
14  import com.example.welcome.ui.theme.WelcomeTheme
15
```

Reviewing the MainActivity

The MainActivity class is declared as a subclass of the Android ComponentActivity class.

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

ComponentActivity
- Base class for activities that enables composition of higher-level components.

onCreate( )
- MainActivity class implements a single method in the form of onCreate().
- This is the first method that is called when an activity is launched by the Android runtime system.

savedInstanceState: Bundle
- It is to save the current state of the activity to deal with configuration changes. (Rotate Device, changing the language settings).

setContent { }
- It defines the activity's layout where composable functions are called.

Composable functions

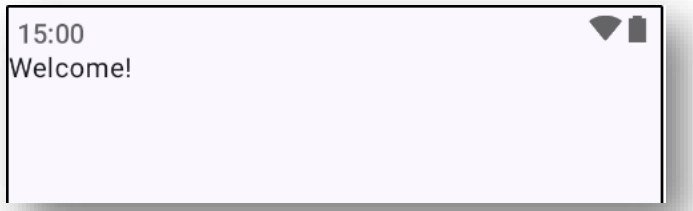
A composable function is differentiated from regular Kotlin functions in code using the **@Composable** annotation.

```
@Composable
fun MyFunction( ) {
   //…
}
```

We can also call other composables from within the function:

```
@Composable
fun MyFunction( ) {
   Text(text = "Welcome")
}
```

It's a composable function that is defined by the Compose UI library displays a text label on the screen



Composable functions

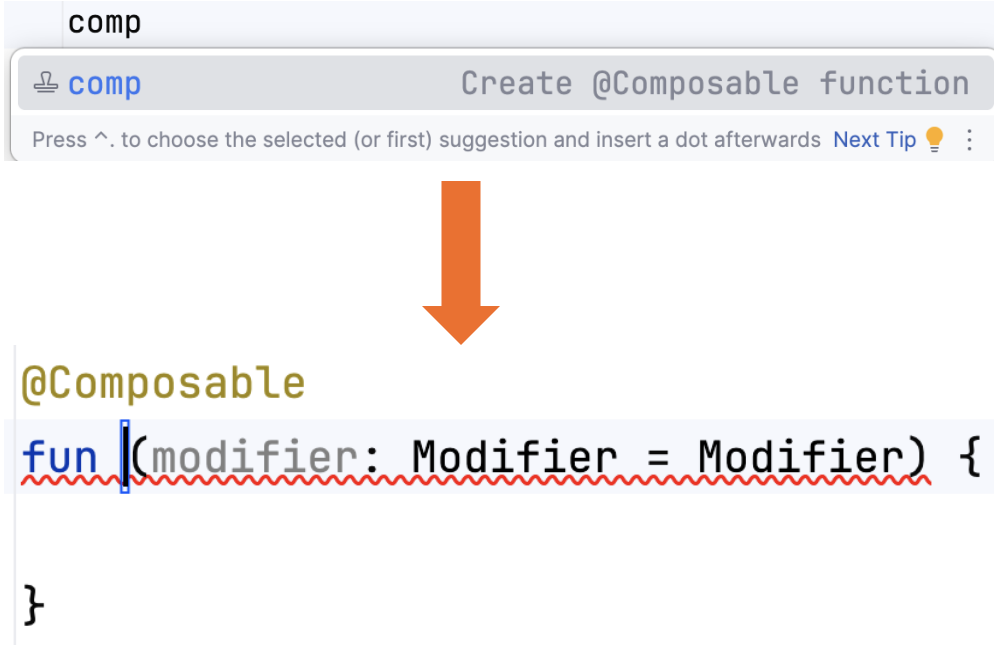
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text("Hello world!")
        }
    }
}
```

To create a composable function, just add the @Composable annotation to the function name.

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

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

Composable functions

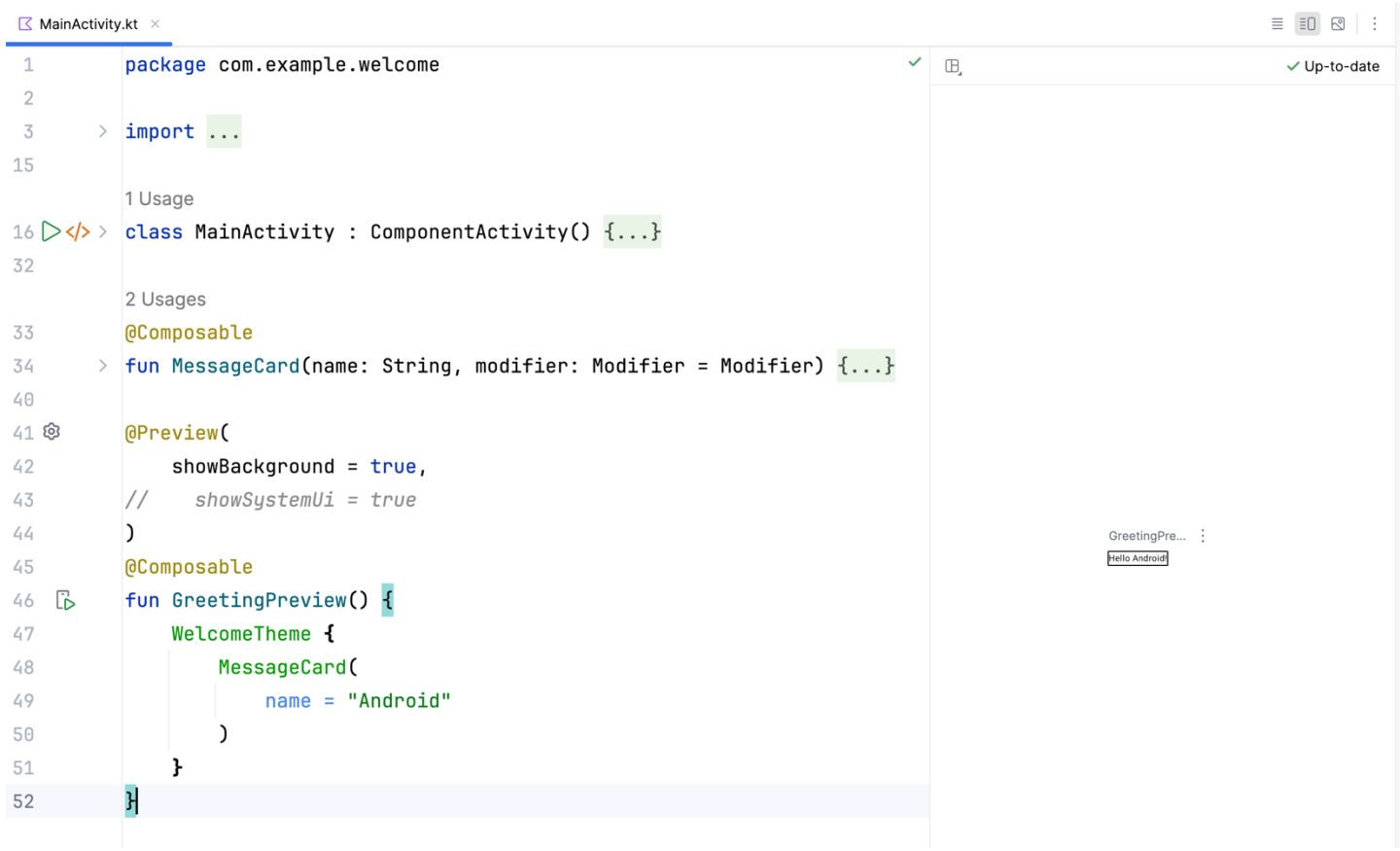
Shortcut: Type **comp** to get the below code automatically



Preview your function

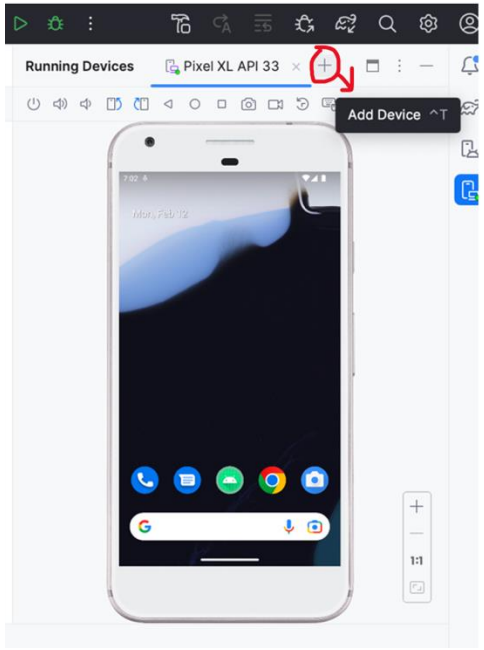
The @Preview annotation lets you preview your composable functions within Android Studio without having to build and install the app to an Android device or emulator.

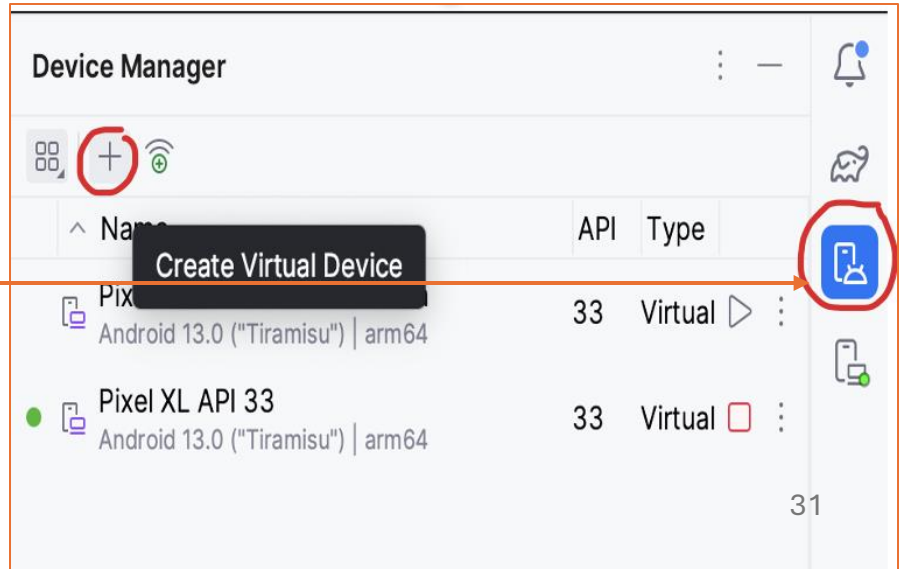
```
@Preview
@Composable
fun PreviewMessageCard() {
    MessageCard("Android")

}
```



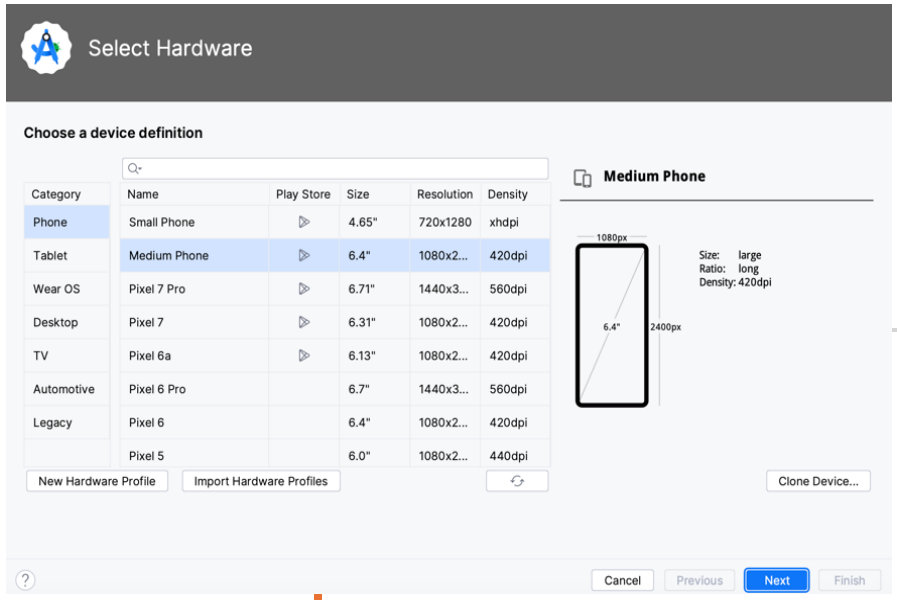
Android Virtual Device (AVD)

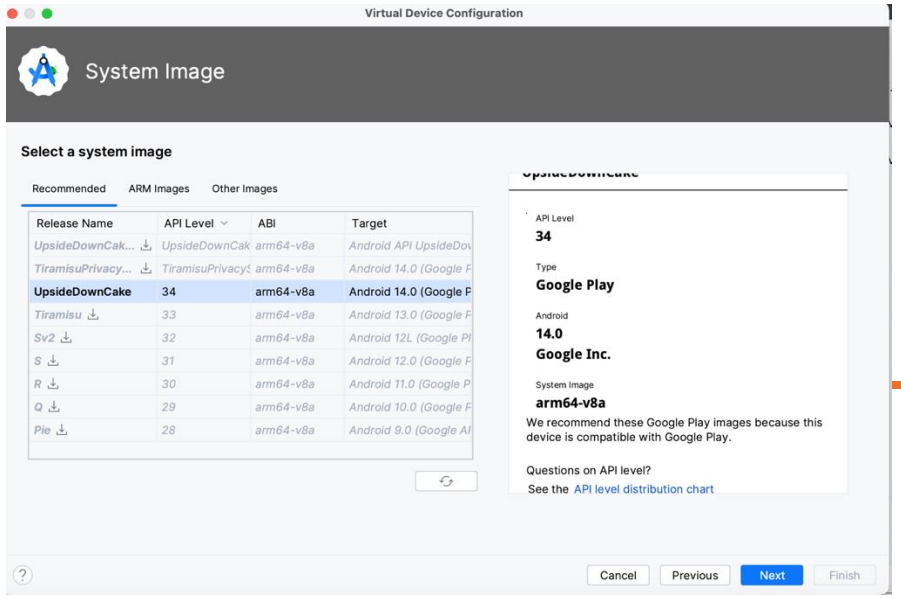
- **Testing Android Applications:**
  - Install and run on a physical device.
  - Use Android Virtual Device (AVD) emulator for testing without physical devices.
- **About AVDs:**
  - Emulators for Android app testing.
  - No need for physical Android devices.
- **Managing AVDs:**
  - Create and manage via Android Virtual Device Manager.
  - Use the "+" icon or Device Manager icon, then click the "+" button to create new AVDs.
  - Refer the step-by-step pictorial view from the next slide

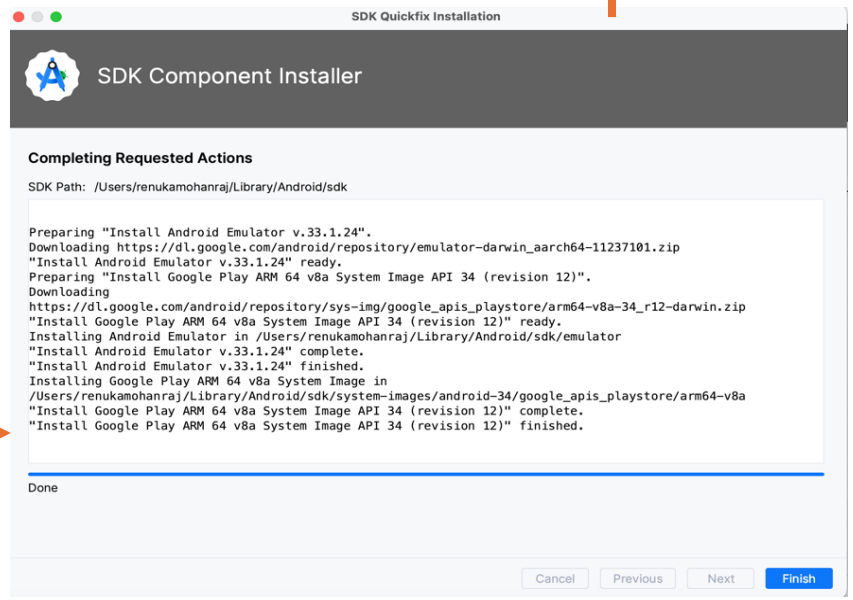


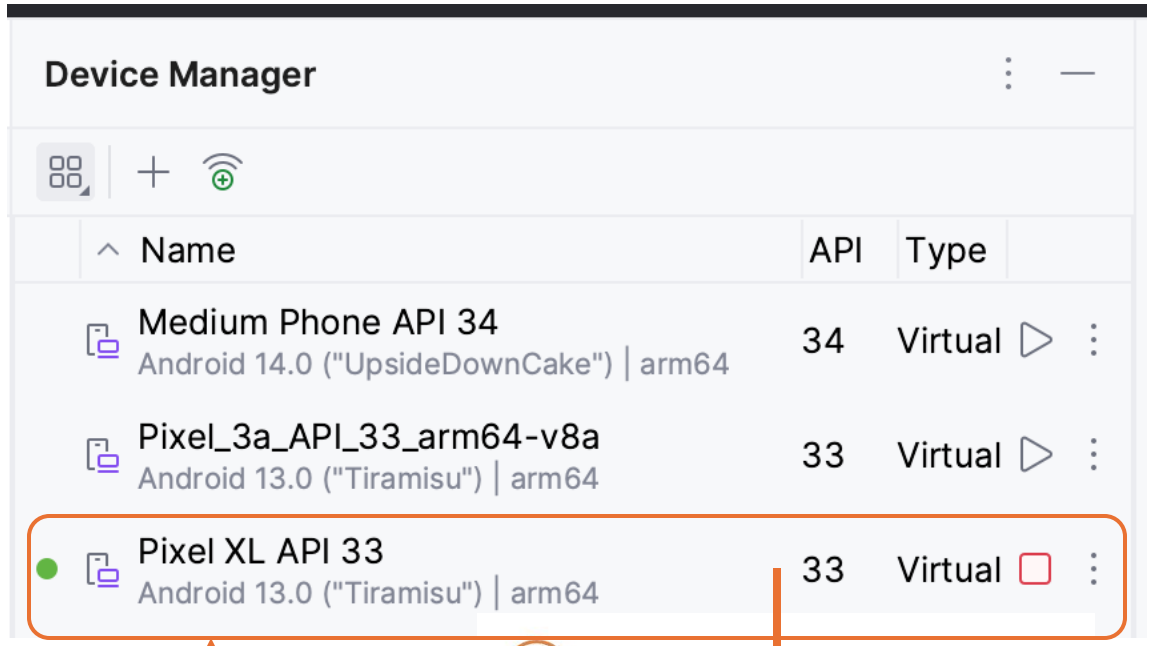


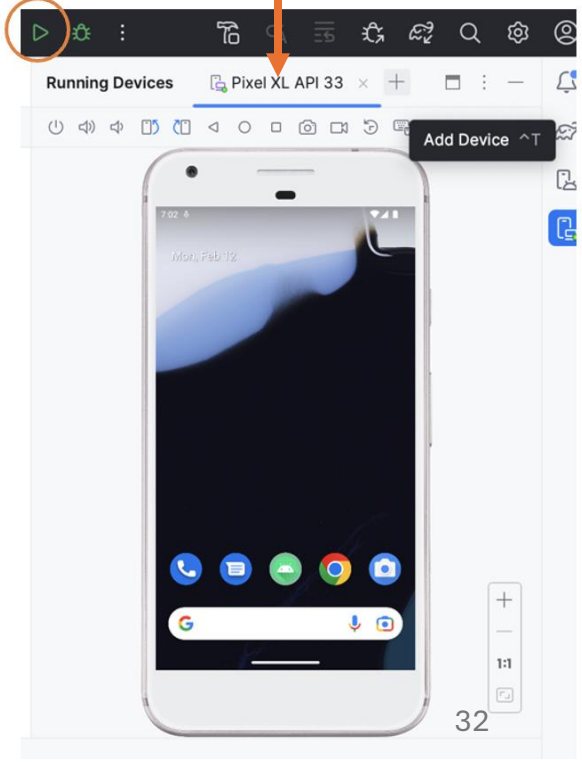
AVD Setup


Select Hardware
Choose a device definition
Category
Name
Play Store
Size
Resolution
Density
Phone
Small Phone
4.65"
720x1280
xhdpi
Tablet
Medium Phone
6.4"
1080x2...
420dpi
Wear OS
Pixel 7 Pro
6.71"
1440x3...
560dpi
Desktop
Pixel 7
6.31"
1080x2...
420dpi
TV
Pixel 6a
6.13"
1080x2...
420dpi
Automotive
Pixel 6 Pro
6.7"
1440x3...
560dpi
Legacy
Pixel 6
6.4"
1080x2...
420dpi
Pixel 5
6.0"
1080x2...
440dpi
Medium Phone
1080px
2400px
6.4"
Size: large
Ratio: long
Density: 420dpi
New Hardware Profile
Import Hardware Profiles
Clone Device...
Cancel
Previous
Next
Finish


Virtual Device Configuration
System Image
Select a system image
Recommended
ARM Images
Other Images
Release Name
API Level
ABI
Target
UpsideDownCake
34
arm64-v8a
Android 14.0 (Google P
Tiramisu
33
arm64-v8a
Android 13.0 (Google P
Sv2
32
arm64-v8a
Android 12L (Google Pl
S
31
arm64-v8a
Android 12.0 (Google P
R
30
arm64-v8a
Android 11.0 (Google P
Q
29
arm64-v8a
Android 10.0 (Google P
Pie
28
arm64-v8a
Android 9.0 (Google Al
API Level
34
Type
Google Play
Android
14.0
Google Inc.
System Image
arm64-v8a
We recommend these Google Play images because this device is compatible with Google Play.
Questions on API level?
See the API level distribution chart
Cancel
Previous
Next
Finish


SDK Quickfix Installation
SDK Component Installer
Completing Requested Actions
SDK Path: /Users/renukamohanraj/Library/Android/sdk
Preparing "Install Android Emulator v.33.1.24".
Downloading https://dl.google.com/android/repository/emulator-darwin_aarch64-11237101.zip
"Install Android Emulator v.33.1.24" ready.
Preparing "Install Google Play ARM 64 v8a System Image API 34 (revision 12)".
Downloading
https://dl.google.com/android/repository/sys-img/google_apis_playstore/arm64-v8a-34_r12-darwin.zip
"Install Google Play ARM 64 v8a System Image API 34 (revision 12)" ready.
Installing Android Emulator in /Users/renukamohanraj/Library/Android/sdk/emulator
"Install Android Emulator v.33.1.24" complete.
"Install Android Emulator v.33.1.24" finished.
Installing Google Play ARM 64 v8a System Image in
/Users/renukamohanraj/Library/Android/sdk/system-images/android-34/google_apis_playstore/arm64-v8a
"Install Google Play ARM 64 v8a System Image API 34 (revision 12)" complete.
"Install Google Play ARM 64 v8a System Image API 34 (revision 12)" finished.
Done
Cancel
Previous
Next
Finish


Device Manager
Name
API
Type
Medium Phone API 34
Android 14.0 ("UpsideDownCake") | arm64
34
Virtual
Pixel_3a_API_33_arm64-v8a
Android 13.0 ("Tiramisu") | arm64
33
Virtual
Pixel XL API 33
Android 13.0 ("Tiramisu") | arm64
33
Virtual


Running Devices
Pixel XL API 33
Add Device ^T
1:1

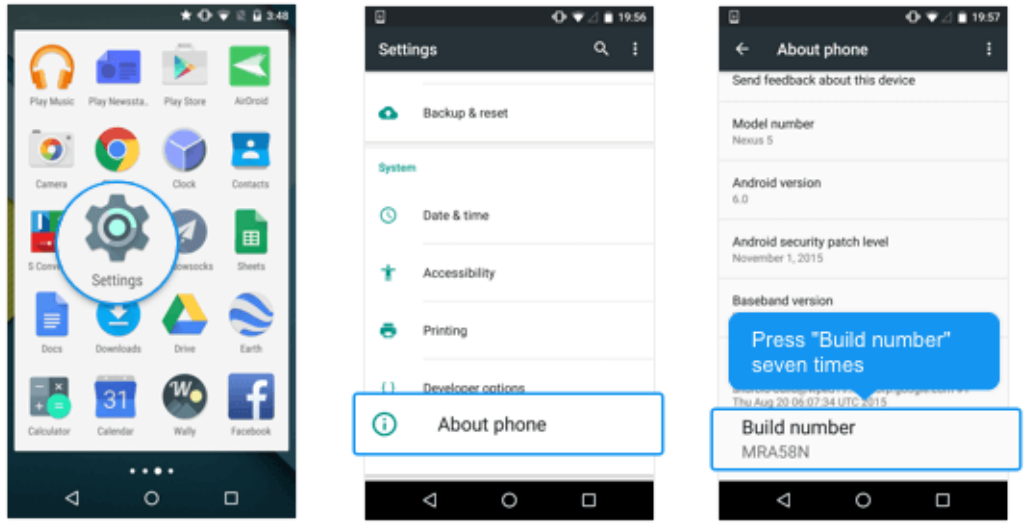
Running your App on Real Device

Set up your device as follows

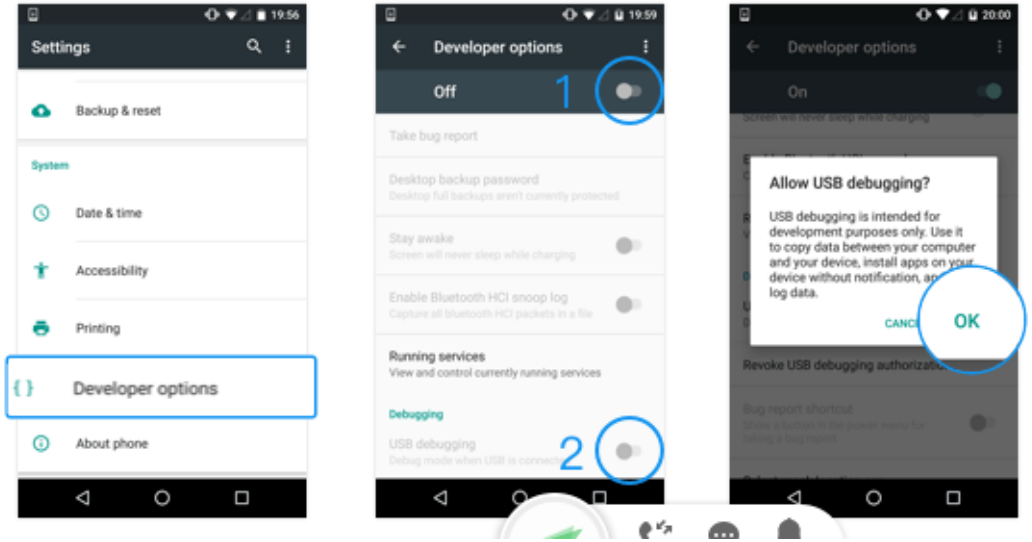
- Connect your device to your development machine with a USB cable.
- Enable **USB debugging** on your device by going to **Settings > Developer options**. **Note:**
  - On Android 4.2 and newer, **Developer options** is hidden by default. To make it available, go to **Settings > About phone/Tablet/Device** and tap **Build number** seven times. Return to the previous screen to find **Developer options**.
- Enable **USB Debugging**.
- See the next slide for Pictorial representation to enable developer options

Running your App on Real Device Setup Screenshots

Ref:
https://developer.android.com/studio/debug/dev-options



2. Go to "Developer options", turn it on, and then enable "USB debugging".



Run the app from Android Studio as follows:

- In Android Studio, select your project and click **Run** from the toolbar.
- In the **Select Deployment Target** window, select your device, and click **OK**.
- Android Studio installs the app on your connected device and starts it.

Hello, World!

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello, $name!",
        modifier = modifier
            .fillMaxWidth(),
        textAlign = TextAlign.Center,
        fontWeight = FontWeight.Bold,
        color = Color.Red
    )
}
```

GreetingPreview

Hello, World!

https://github.com/brightgeevarghese/ComposeHelloWorld.git

GreetingPreview1 - Light Mode

GreetingPreview2 - Dark Mode


```
@Preview(
    showBackground = true,
    showSystemUi = true,
    wallpaper = Wallpapers.NONE,
    uiMode = Configuration.UI_MODE_NIGHT_YES
)
```

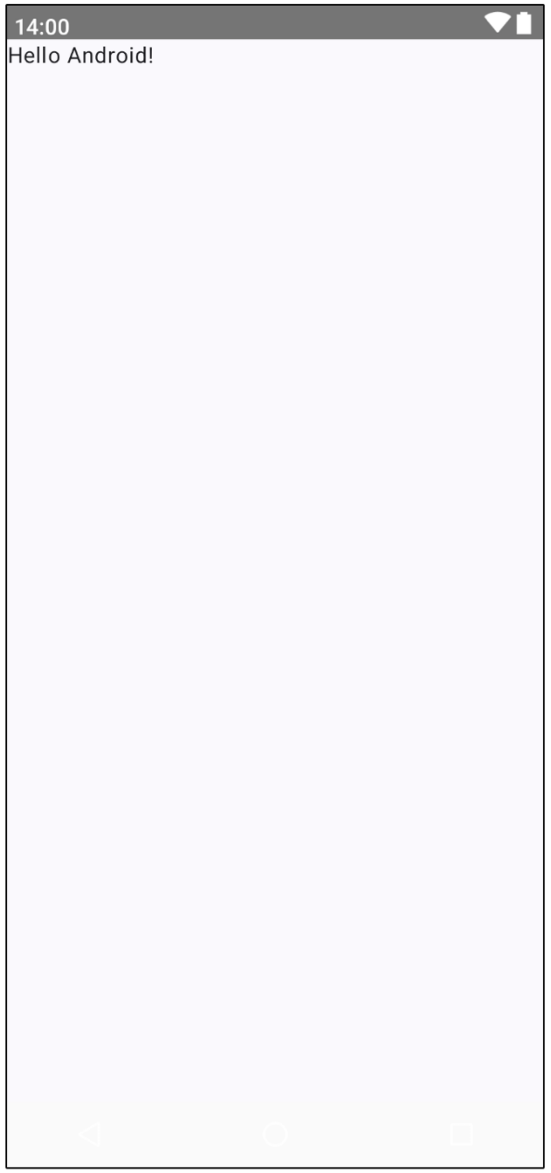
Multiple Previews

https://github.com/brightgeevarghese/MultiplePreviews.git

```
@Preview(
    name = "Light Mode",
    showSystemUi = true,
    showBackground = true
)
@Composable
fun GreetingPreview1() {
    MultiplePreviewsTheme {
        //...
    }
}
```

```
@Composable
@Preview(
    showBackground = true,
    showSystemUi = true,
    name = "Dark Mode",
    uiMode = Configuration.UI_MODE_NIGHT_YES
)
fun GreetingPreview2() {
    MultiplePreviewsTheme {
        //...
    }
}
```

GreetingPreview1 - Light Mode



GreetingPreview2 - Dark Mode

Layouts

- Layout Introduction
  - Compose offers diverse user interface components for app development.
  - Layout composables in Compose facilitate organization of the user interface.
  - UI elements follow a hierarchical structure, nested within one another.
  - Building UI hierarchy involves calling composable functions within others.

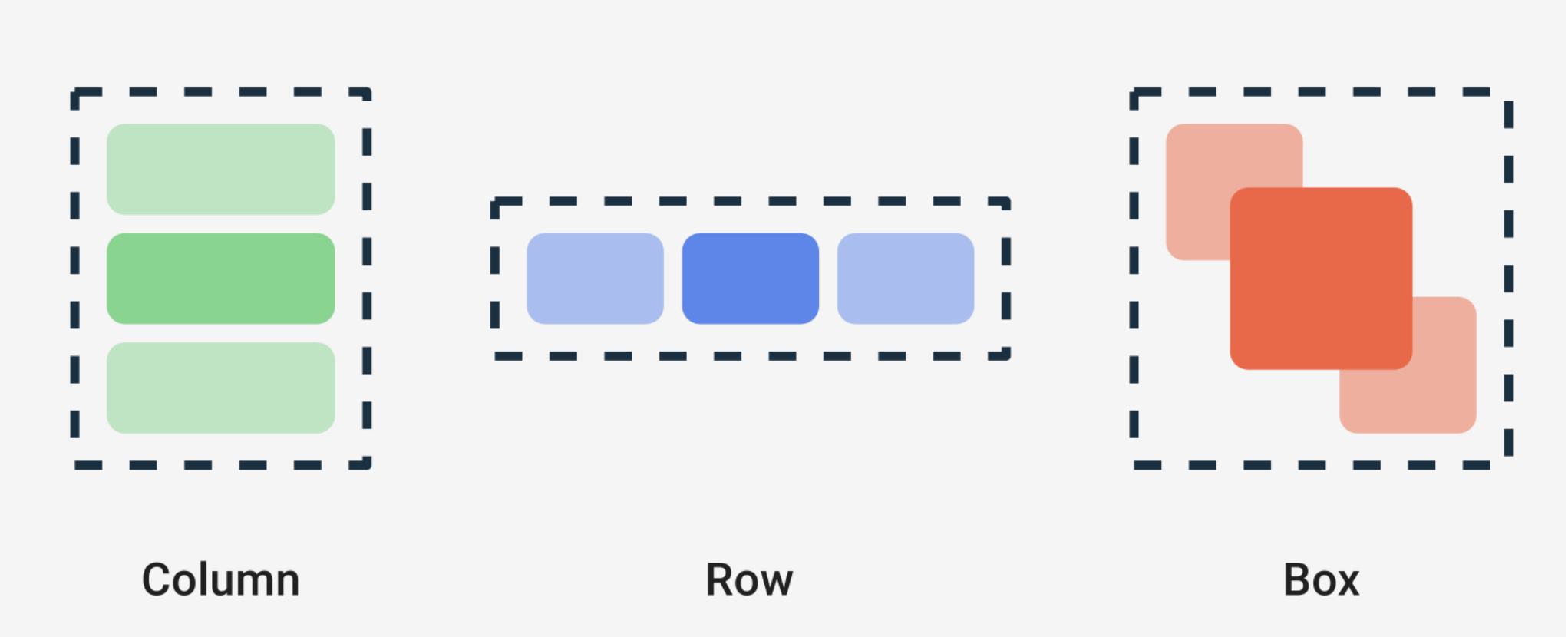
- Without guidance on how you want them arranged, Compose stacks the text elements on top of each other, making them unreadable

```
@Preview
@Composable
fun whyLayouts(){
   Text("      Android     Development")
   Text("Jetpack Compose")
}
```

Preview Result

Standard/Simple Layouts

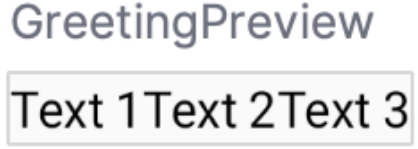
- You can write your own composable function to combine these layouts into a more elaborate layout that suits your app.



Row and Column Composable Layouts

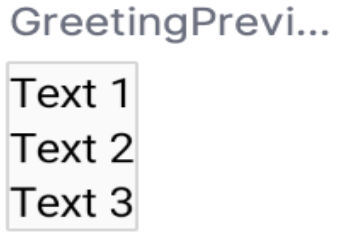
The Row composable, as the name suggests, lays out its children horizontally on the screen.

```
@Composable
fun GreetingPreview(){
   // Layout using Rows for the Horizontal
  //Alignment
   Row {
      Text("Text 1")
      Text("Text 2")
      Text("Text 3")
   }
}
```



The Column composable lets you arrange elements vertically.

```
@Composable
fun MainScreenColumn(){
   // Layout using Rows for the Vertical Alignment
   Column {
      Text("Text 1")
      Text("Text 2")
      Text("Text 3")
   }
}
```



Using Modifiers in Compose

- Modifiers allow you to decorate or enhanced a composable. Modifiers are standard Kotlin objects.
- Modifiers let you do these sorts of things:
  - Change the composable's size, layout, behavior, and appearance
  - Add information, like accessibility labels
  - Process user input
  - Add high-level interactions, like making an element clickable, scrollable, draggable, or zoomable

Creating a Modifier

```
val modifier = Modifier
```

Call methods on the Modifier to apply settings

Example: Padding and border configuration

```
val modifier = Modifier
    .padding(all = 10.dp)
    .border(width = 2.dp, color = Color.Black)
```

Multiple configurations can be applied by chaining method calls.

dp stands for density pixel

Row and Column Child components arrangements

We can decide on how to place child composables on a container's **main axis**

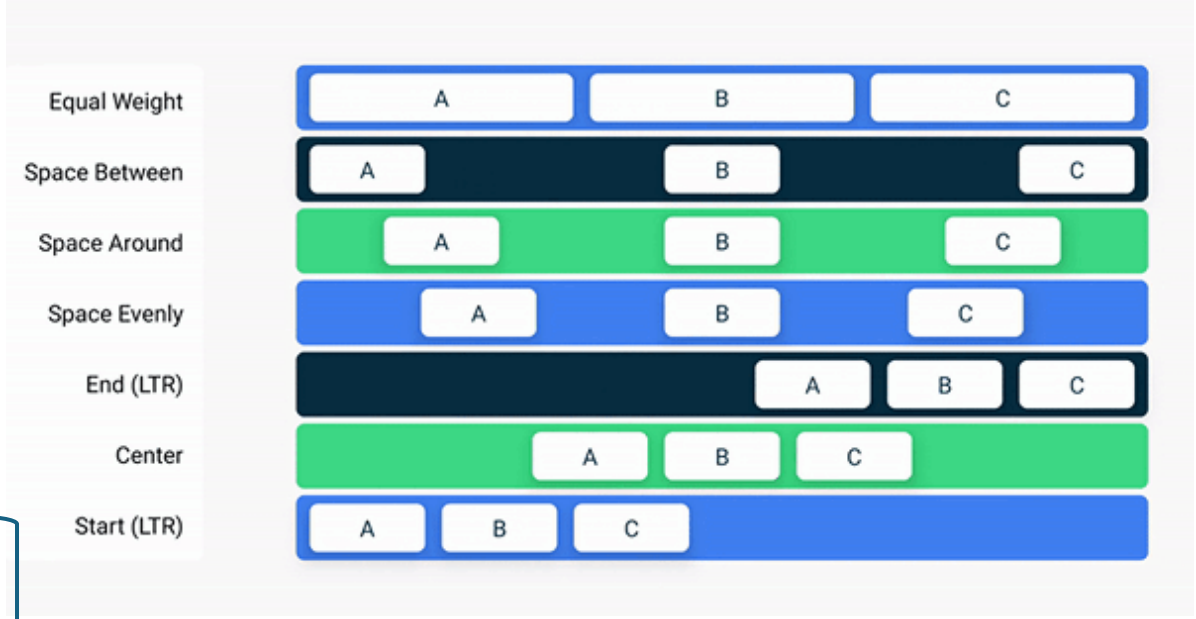
- horizontal for Row
- vertical for Column

Row Child components arrangements

However, we can decide on how to place child composables on a container's **main axis** (horizontal for Row, vertical for Column).

- For a Row, you can choose the following arrangements/alignments:

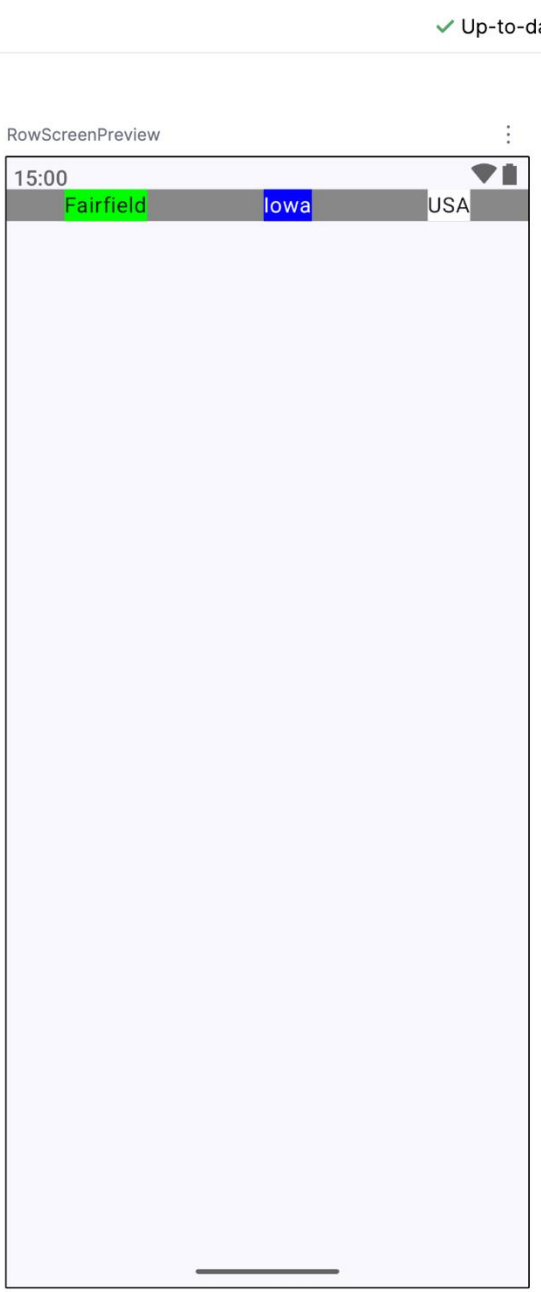
```
Row(
    modifier = Modifier
        .background(Color.LightGray)
        .padding(16.dp)
        .fillMaxSize(),
    horizontalArrangement = Arrangement.SpaceEvenly,
//      horizontalArrangement = Arrangement.SpaceBetween,
//      horizontalArrangement = Arrangement.SpaceAround,
//      horizontalArrangement = Arrangement.spacedBy(16.dp),
//      horizontalArrangement = Arrangement.Start,
//      horizontalArrangement = Arrangement.End,
//      verticalAlignment = Alignment.Top,
//      verticalAlignment = Alignment.Bottom,
    verticalAlignment = Alignment.CenterVertically,
  )
```



Choose any one arrangement

Choose any one alignment

```
@Composable
fun RowScreen(modifier: Modifier = Modifier) {
    Row(
        modifier = modifier
            .fillMaxWidth()
//            .fillMaxHeight()
            .background(Color.Gray),
        horizontalArrangement = Arrangement.SpaceAround,
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text(
            text = "Fairfield",
            modifier = Modifier
                .background(Color.Green)
        )
        Text(
            text = "Iowa",
            color = Color.White,
            modifier = Modifier
                .background(Color.Blue)
        )
        Text(
            text = "USA",
            modifier = Modifier
                .background(Color.White)
        )
    }
}
```



Experiment with Row{...}

MyTextPreview

Fairfield Iowa USA

Row

https://github.com/brightgeevarghese/ComposeRowDemo.git

Column Child components arrangements

Column Child components arrangements

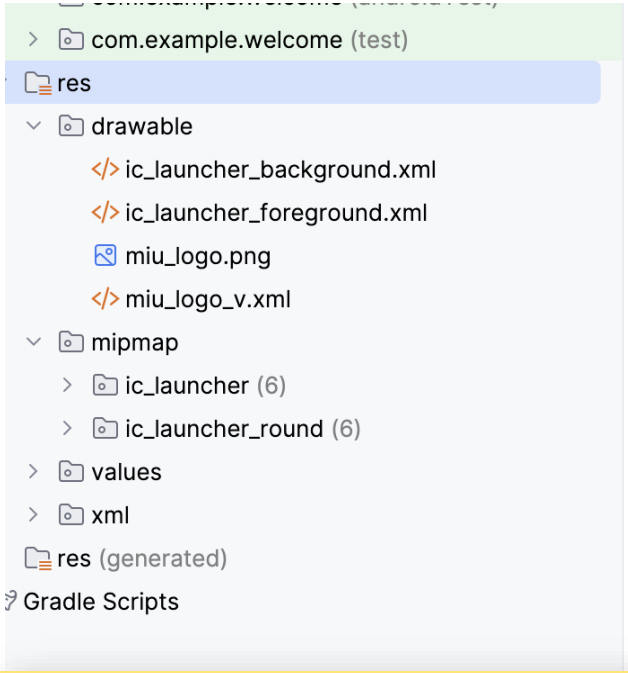
```
@Composable
@Preview(showBackground = true)
fun DefaultPreview() {
    Column(
        modifier = Modifier
            .fillMaxSize(),
        verticalArrangement = Arrangement.Top,
//          verticalArrangement = Arrangement.Bottom,
//          verticalArrangement = Arrangement.Center,
//          verticalArrangement = Arrangement.SpaceEvenly,
//          verticalArrangement = Arrangement.SpaceBetween,
//          verticalArrangement = Arrangement.SpaceAround
        horizontalAlignment = Alignment.CenterHorizontally
//          horizontalAlignment = Alignment.Start
//          horizontalAlignment = Alignment.End
    ) {
```

Code:
https://github.com/brightgeevarghese/ComposeColumnDemo.git

Choose any one arrangement

Choose any one alignment

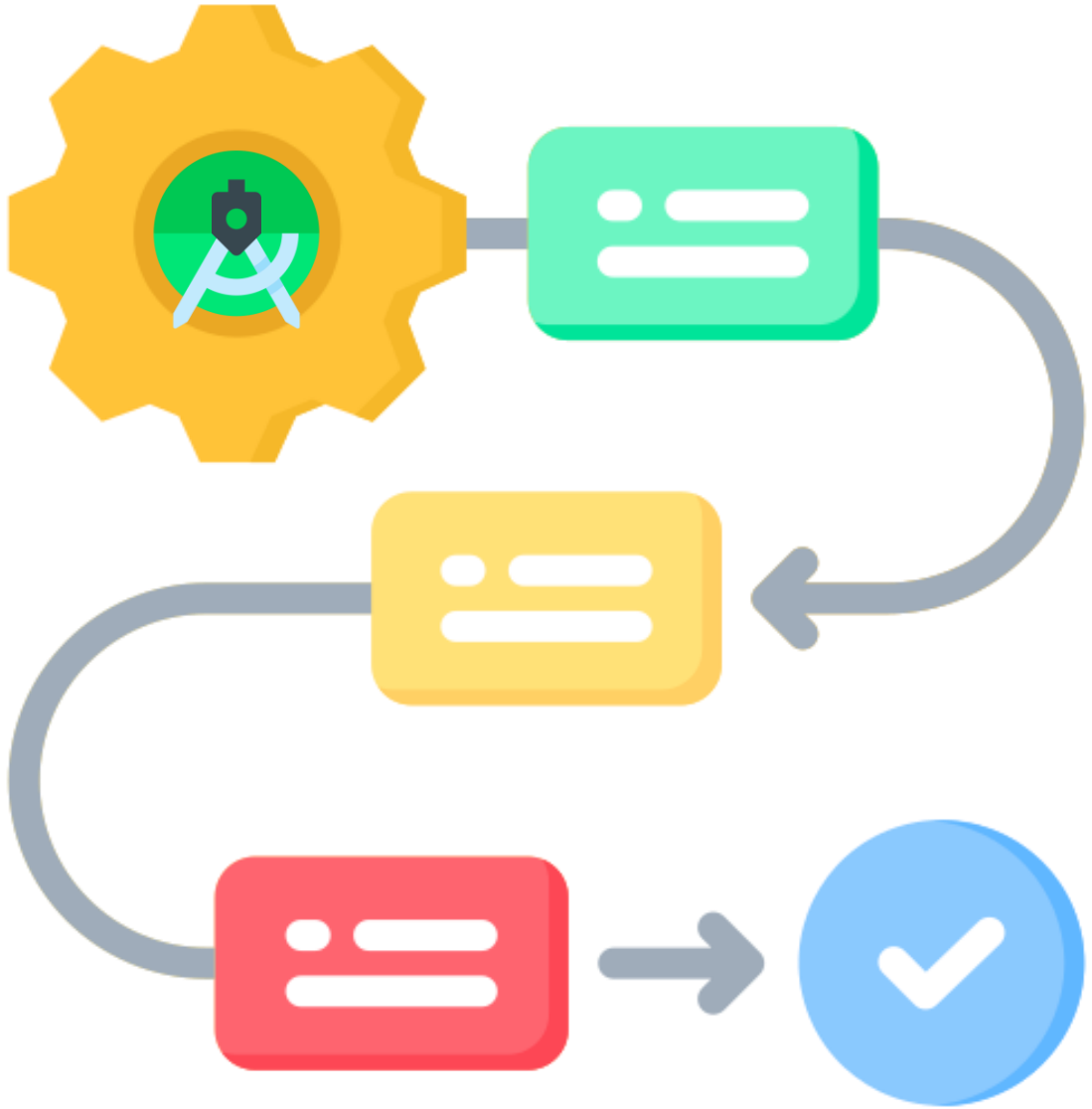
Different ways to display

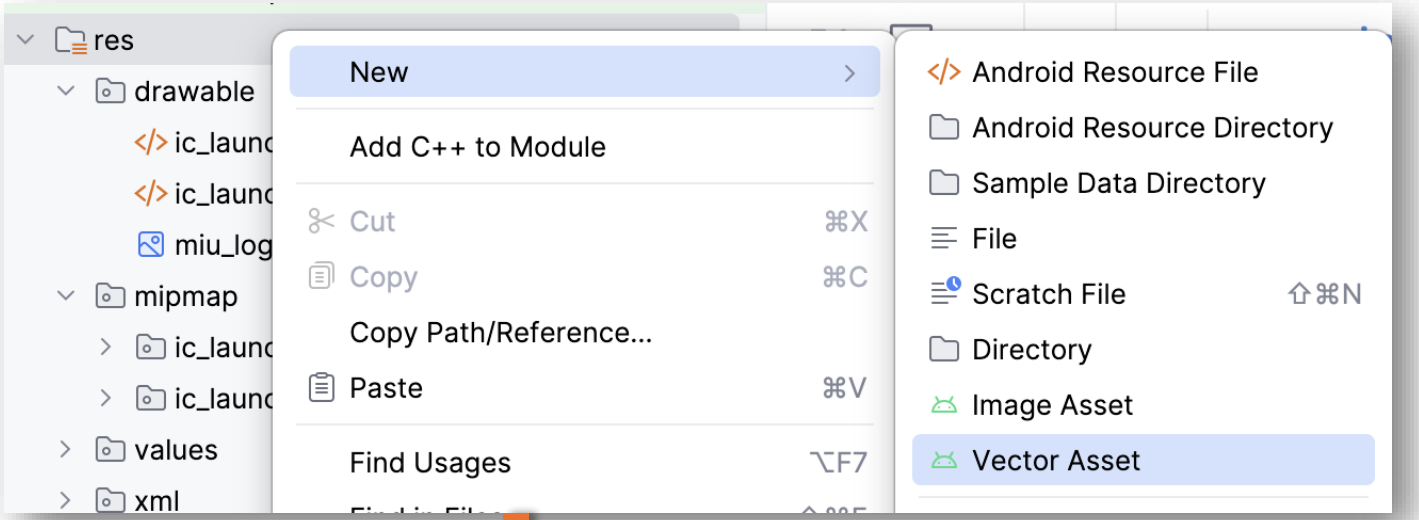


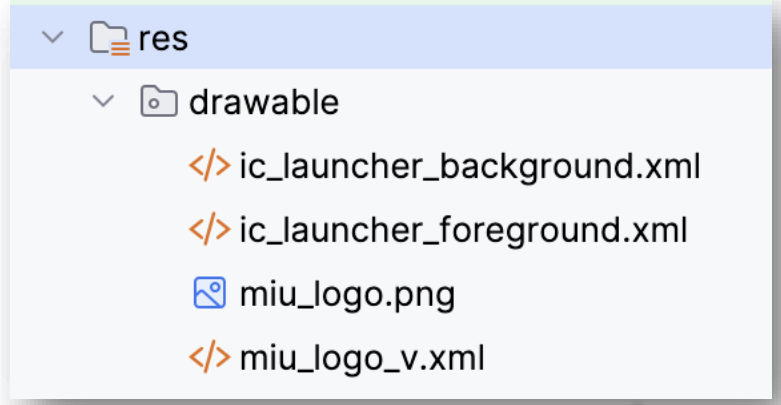

```
53
54  Image(
55      painter = painterResource(id = R.drawable.miu_logo),
56      contentDescription = "image",
57  )
58  Image(
59      imageVector = ImageVector.vectorResource(id = R.drawable.miu_logo_v),
60      contentDescription = "image",
61  )
62  Image(
63      bitmap = ImageBitmap.imageResource(id = R.drawable.miu_logo),
64      contentDescription = "image",
    )
```

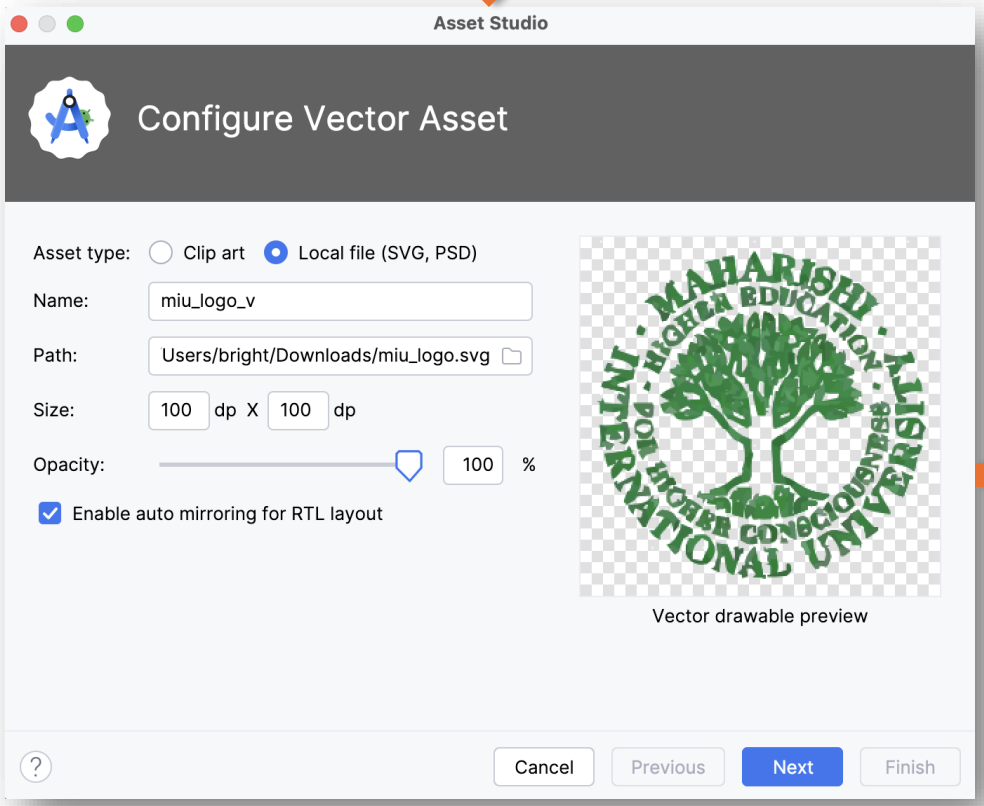
For ImageVector, we need a SVG file.

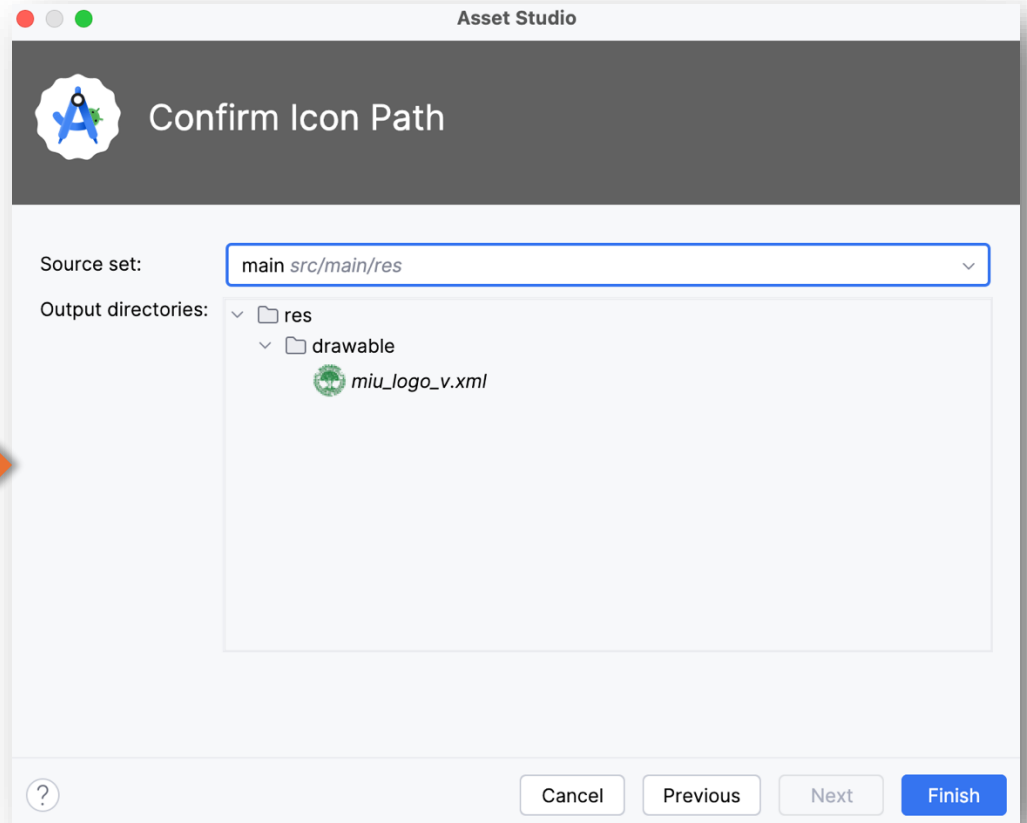
Steps to Add a SVG Image



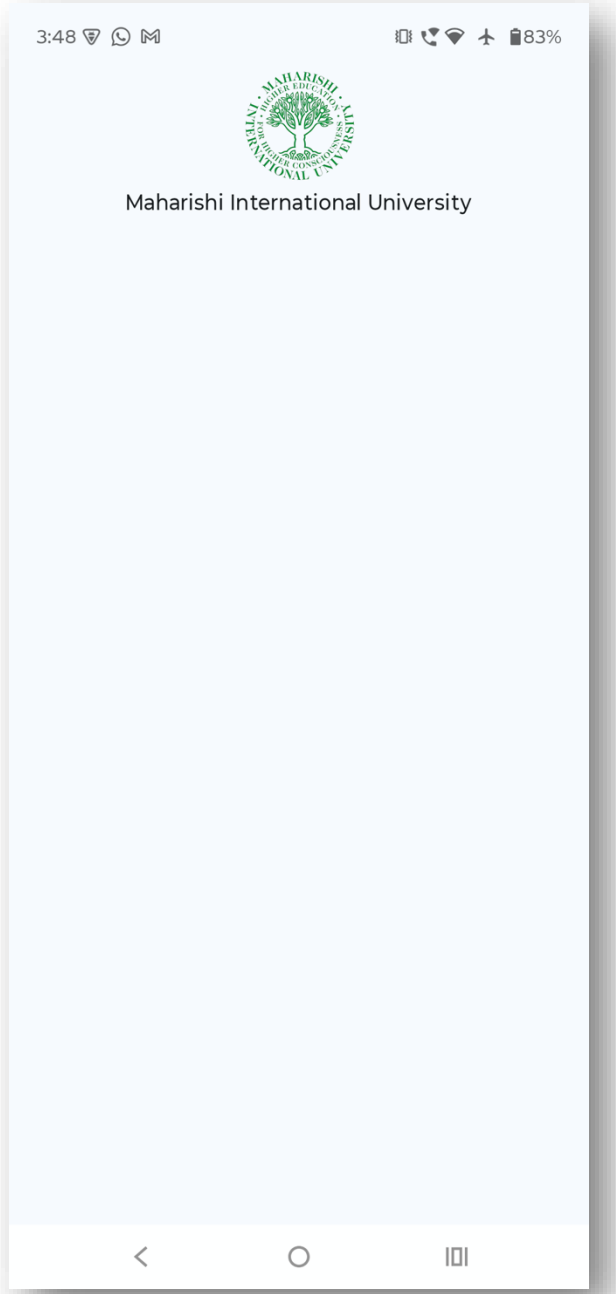

res
drawable
mipmap
values
xml
New
Add C++ to Module
Cut ⌘X
Copy ⌘C
Copy Path/Reference...
Paste ⌘V
Find Usages ⌥F7
Android Resource File
Android Resource Directory
Sample Data Directory
File
Scratch File ⇧⌘N
Directory
Image Asset
Vector Asset


res
drawable
ic_launcher_background.xml
ic_launcher_foreground.xml
miu_logo.png
miu_logo_v.xml

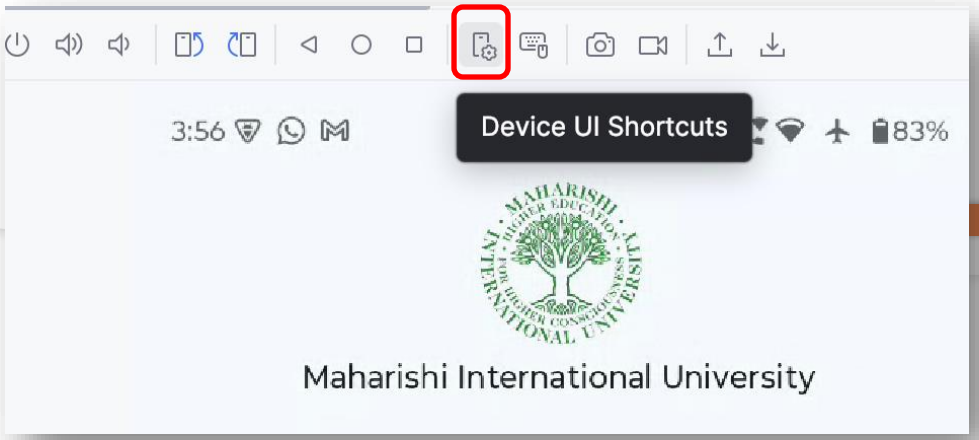

Asset Studio
Configure Vector Asset
Asset type: Clip art Local file (SVG, PSD)
Name: miu_logo_v
Path: Users/bright/Downloads/miu_logo.svg
Size: 100 dp X 100 dp
Opacity: 100 %
Enable auto mirroring for RTL layout
Vector drawable preview
Cancel
Previous
Next
Finish

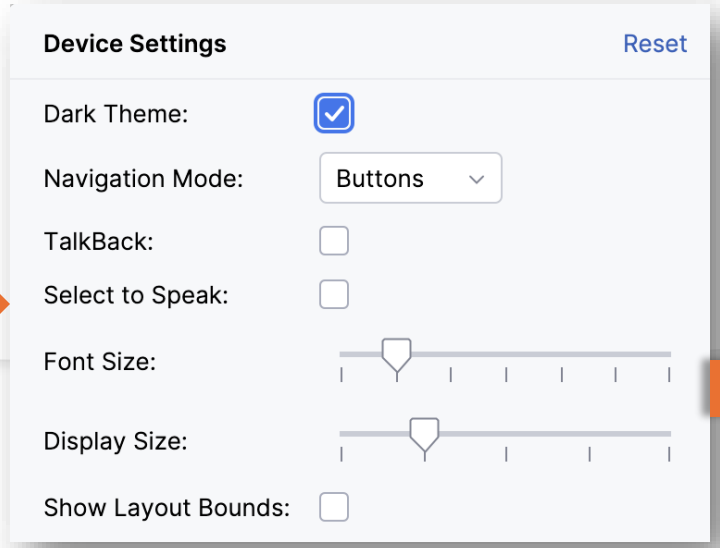

Asset Studio
Confirm Icon Path
Source set: main src/main/res
Output directories:
res
drawable
miu_logo_v.xml
Cancel
Previous
Next
Finish

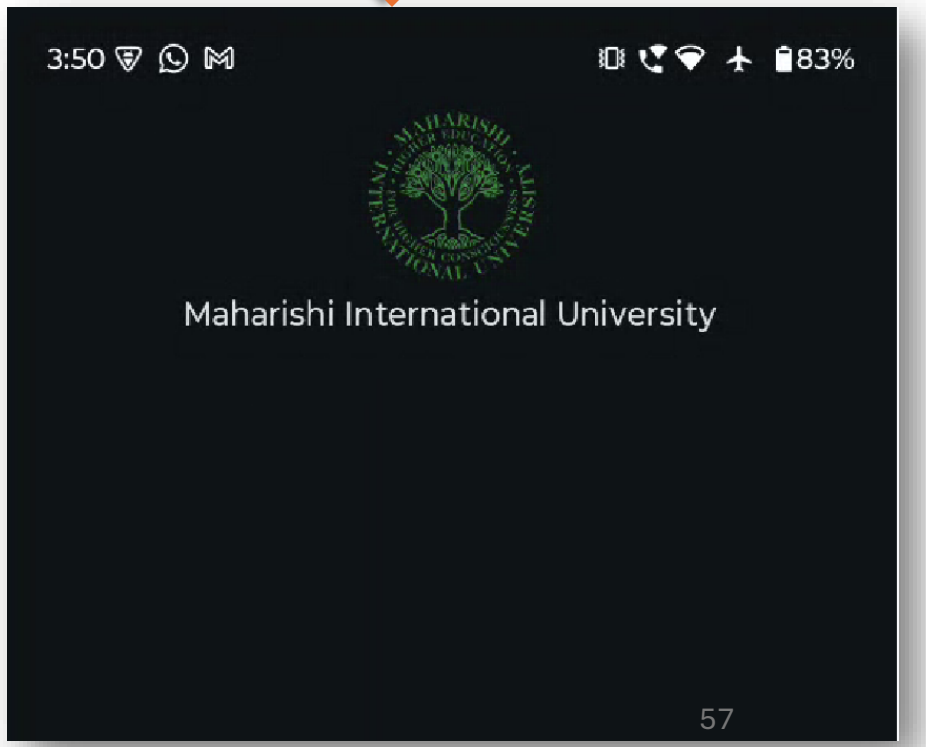
```
Column(
    modifier = Modifier.padding(innerPadding)
        .fillMaxWidth(),
    horizontalAlignment = Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.Top
) {
    Image(
        painter = painterResource(R.drawable.miu_logo),
        contentDescription = "My Image",
        modifier = Modifier
            .border(
                width = 2.dp,
                color = Color.White,
                shape = CircleShape
            )
            .clip(CircleShape)
    )
    Text(
        text = "Maharishi International University"
    )
}
```

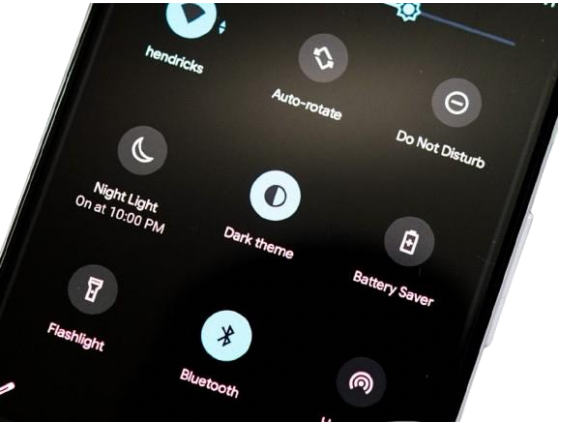


https://github.com/brightgeevarghese/ComposeColumnDemo









How to switch the theme of device in Android Studio?

Box Layout

Stacks child elements in layers

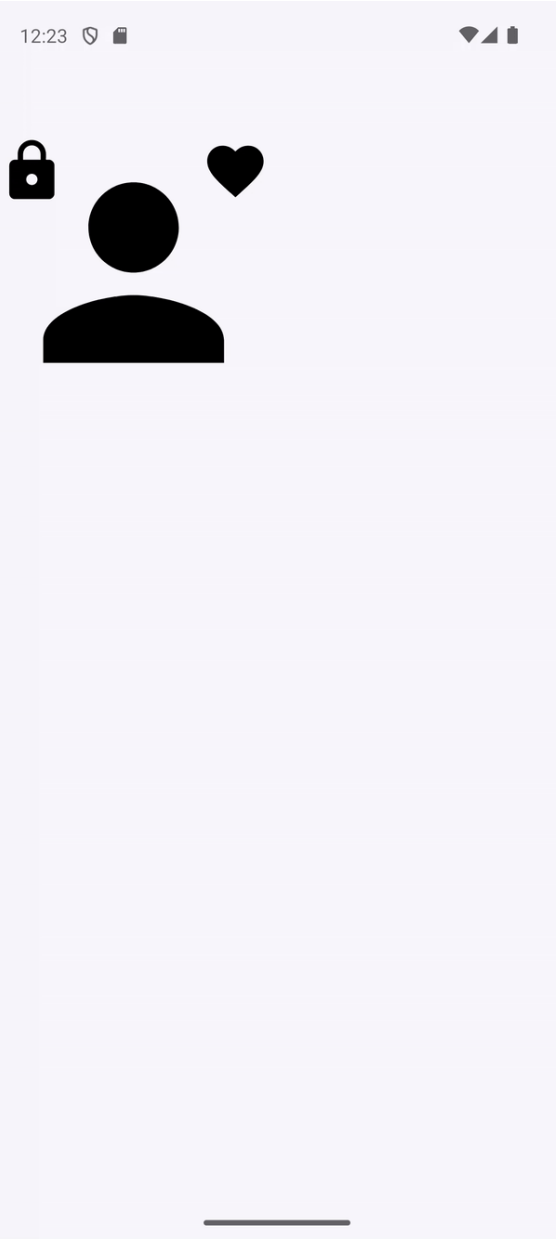
Organizes elements in a single stack, one on top of another

Order of declaration determines stacking: first child at the bottom

Ideal for overlaying elements such as images and text

Allows configuration of element alignment within the stack

Useful for creating complex UIs with elements positioned over others



Box Layout

https://github.com/brightgeevarghese/ComposeBoxLayoutImage

How to make an image clickable?

```
val context = LocalContext.current
Image(
    imageVector = Icons.Default.Lock,
    contentDescription = null,
    modifier = Modifier
        .align(Alignment.TopStart)
        .size(50.dp)
        .clickable(
            onClick = { // When clicked, show a short Toast message
                Toast.makeText(
                    context,                    // Pass current context to Toast
                    "Clicked on Lock",          // Message to display
                    Toast.LENGTH_SHORT          // Duration (short)
                ).show()
            }
        )
)
```

- *Get the current Android Context inside a Composable.*
- *Needed here because Toast requires a Context.*

State and Jetpack Compose

- State in an app is any value that can change over time.
- Compose is declarative and as such the only way to update it is by calling the same composable with new arguments. These arguments are representations of the UI state.
- Any time a state is updated a *recomposition* takes place.

State in composables

- Composable functions can use the remember function to store an object in memory.
- A value computed by remember is stored in the Composition during initial composition, and the stored value is returned during recomposition.
- remember can be used to store both mutable and immutable objects.

```
remember { /*...*/ }
```

State in composables

- mutableStateOf creates an observable MutableState<T>, which is an observable type integrated with the compose runtime.
- Any changes to value schedules recomposition of any composable functions that read value.
- There are three ways to declare a MutableState object in a composable:

```
val mutableState = remember { mutableStateOf(default) }

var value by remember { mutableStateOf(default) }

val (value, setValue) = remember { mutableStateOf(default) }
```

State with and without remember

Without Delegation (`.value`)

```
// Case 1: Without remember
var isClicked = mutableStateOf(false)

isClicked.value = true   // works, but resets on recomposition
```

```
// Case 2: With remember
var isClicked = remember { mutableStateOf(false) }

isClicked.value = true   // survives recompositions
```

- Must use **.value** to access or update.
- remember is needed to preserve state across recompositions.

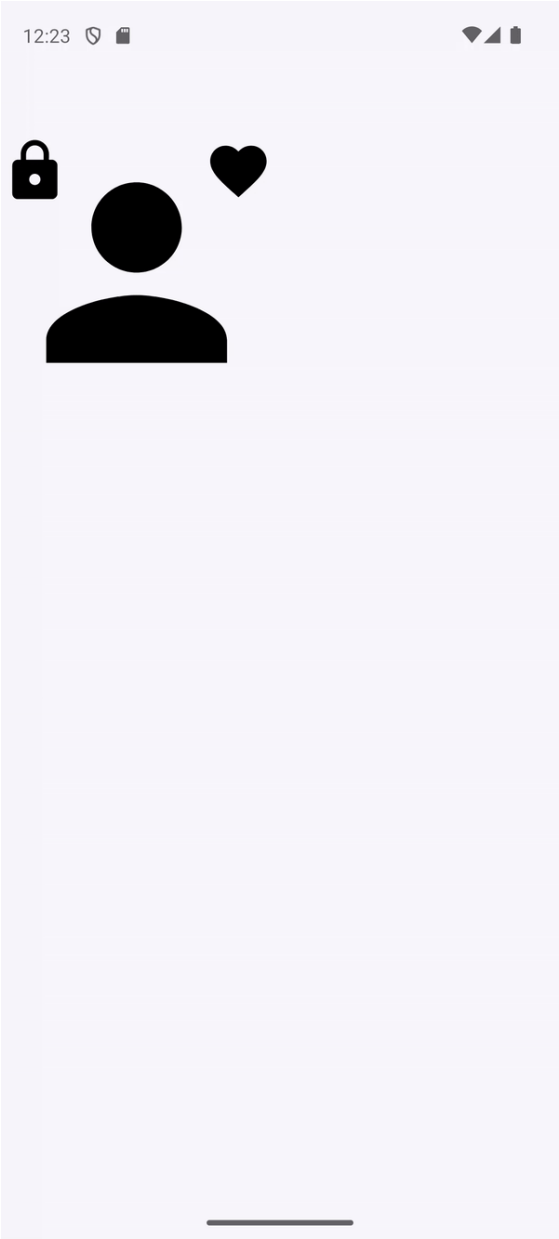
State with and without remember

With Delegation (`by`)

```
// Case 3: Without remember ❌
var isClicked by mutableStateOf(false)

isClicked = true        // resets on recomposition
```

```
// Case 4: With remember
var isClicked by remember { mutableStateOf(false) }

isClicked = true        // survives recompositions
```

- by delegates getter/setter to .value.
- Cleaner syntax => no need to write ~~.value~~.
- remember ensures the value is not reset on every UI refresh.



Box Layout

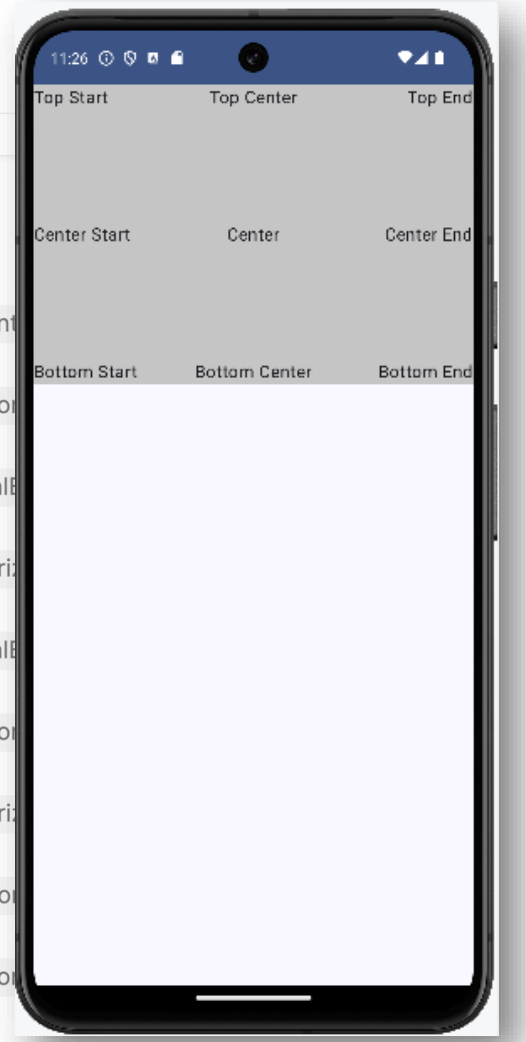
https://github.com/brightgeevarghese/ComposeBoxLayoutImage

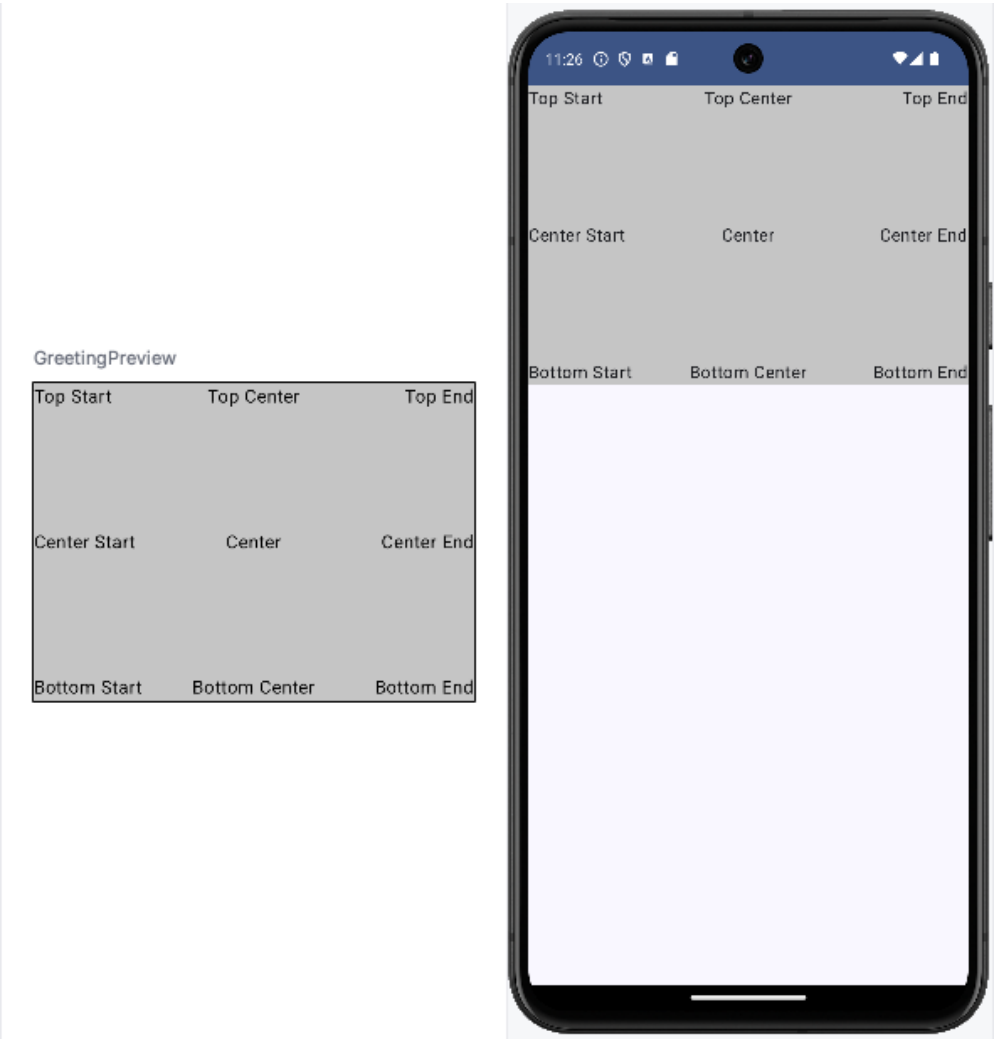
Alignment in Box

```
val configuration = LocalConfiguration.current
val screenWidth = configuration.screenWidthDp.dp
val screenHeight = configuration.screenHeightDp.dp
Box(
   modifier = modifier
      .size(
         width = screenWidth,
         height = screenHeight / 3
      )
      .background(
         Color.LightGray
      )
) {
   Text(
      text = "Center",
      modifier = Modifier
         .align(Alignment.Center)
   )
   ...
}
```

Alignment.kt

```
35    fun interface Alignment {

78        companion object {
79            // 2D Alignments.
80            @Stable
81            val TopStart: Alignment = BiasAlignment( horizont
82            @Stable
83            val TopCenter: Alignment = BiasAlignment( horizo
84            @Stable
85            val TopEnd: Alignment = BiasAlignment( horizontalB
86            @Stable
87            val CenterStart: Alignment = BiasAlignment( hori
88            @Stable
89            val Center: Alignment = BiasAlignment( horizontalB
90            @Stable
91            val CenterEnd: Alignment = BiasAlignment( horizo
92            @Stable
93            val BottomStart: Alignment = BiasAlignment( hori
94            @Stable
95            val BottomCenter: Alignment = BiasAlignment( ho
96            @Stable
97            val BottomEnd: Alignment = BiasAlignment( horizo
```

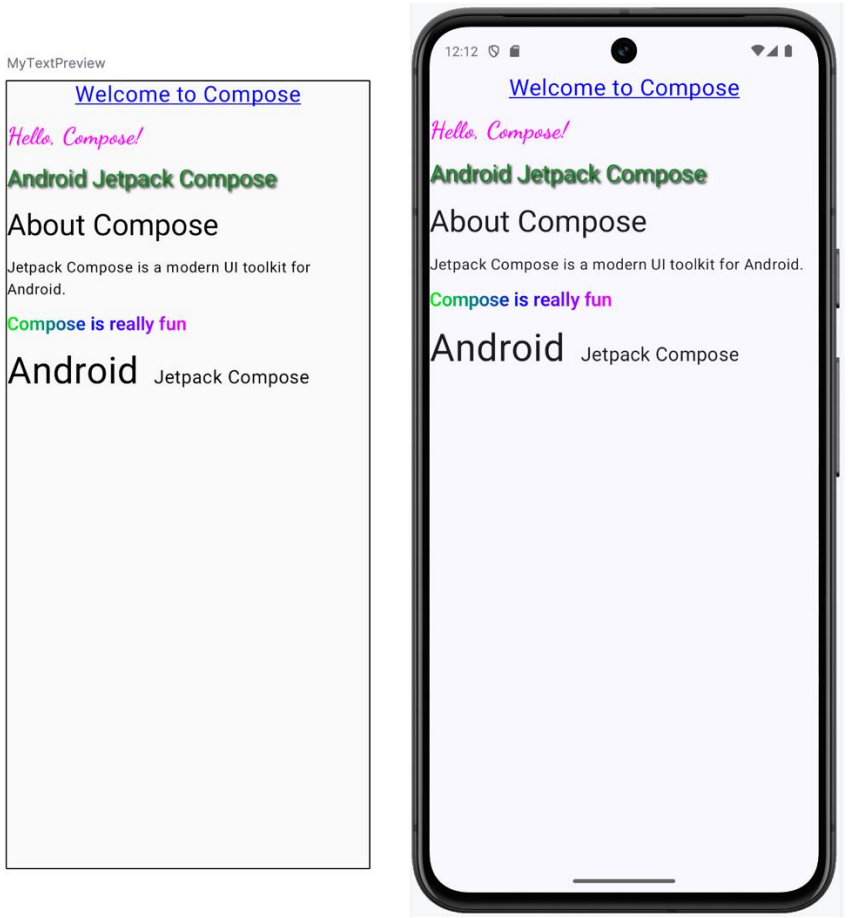




Alignment in Box

https://github.com/brightgeevarghese/BoxLayout-AlignmentInBox

Text Style


MyTextPreview
Welcome to Compose
Hello, Compose!
Android Jetpack Compose
About Compose
Jetpack Compose is a modern UI toolkit for Android.
Compose is really fun
Android Jetpack Compose
12:12
Welcome to Compose
Hello, Compose!
Android Jetpack Compose
About Compose
Jetpack Compose is a modern UI toolkit for Android.
Compose is really fun
Android Jetpack Compose

Text Style

Welcome to Compose

```
color = Color.Blue,
textDecoration = TextDecoration.Underline,
```

Hello Compose

```
color = Color.Magenta,
fontFamily = FontFamily.Cursive
```

Android Jetpack Compose

```
style = TextStyle(
            shadow = Shadow(
                color = Color(0xFF1C862A),
//                  blurRadius = 30f,
                offset = Offset(
                    x = 5f,//The shadow is offset 5 pixels to the right
                    y = 5f//The shadow is offset 5 pixels down from the text
                )
            )
        )
```

Text Style

About Compose

Jetpack Compose is a modern UI toolkit for Android.

Compose is really fun

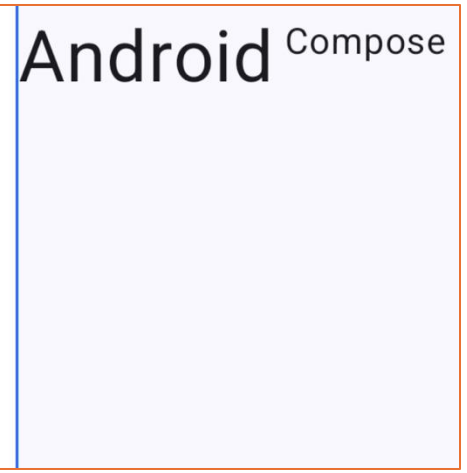
```
style = MaterialTheme.typography.headlineLarge
```

```
style = MaterialTheme.typography.bodyLarge
```

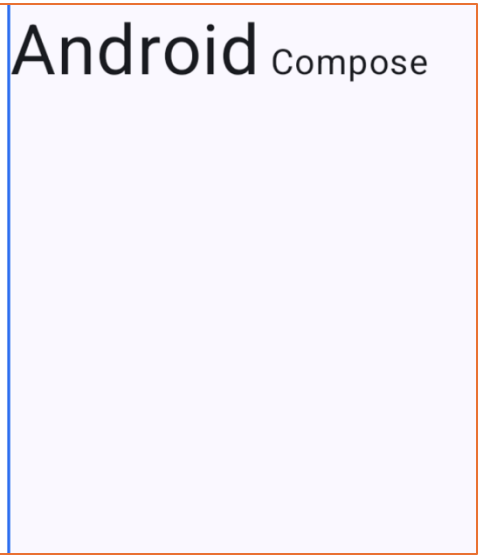
```
style = TextStyle(
    brush = Brush.linearGradient(
        colors = listOf(
            Color.Red,
            Color.Blue,
            Color.Green,
            Color.Red
        )
    )
)
```

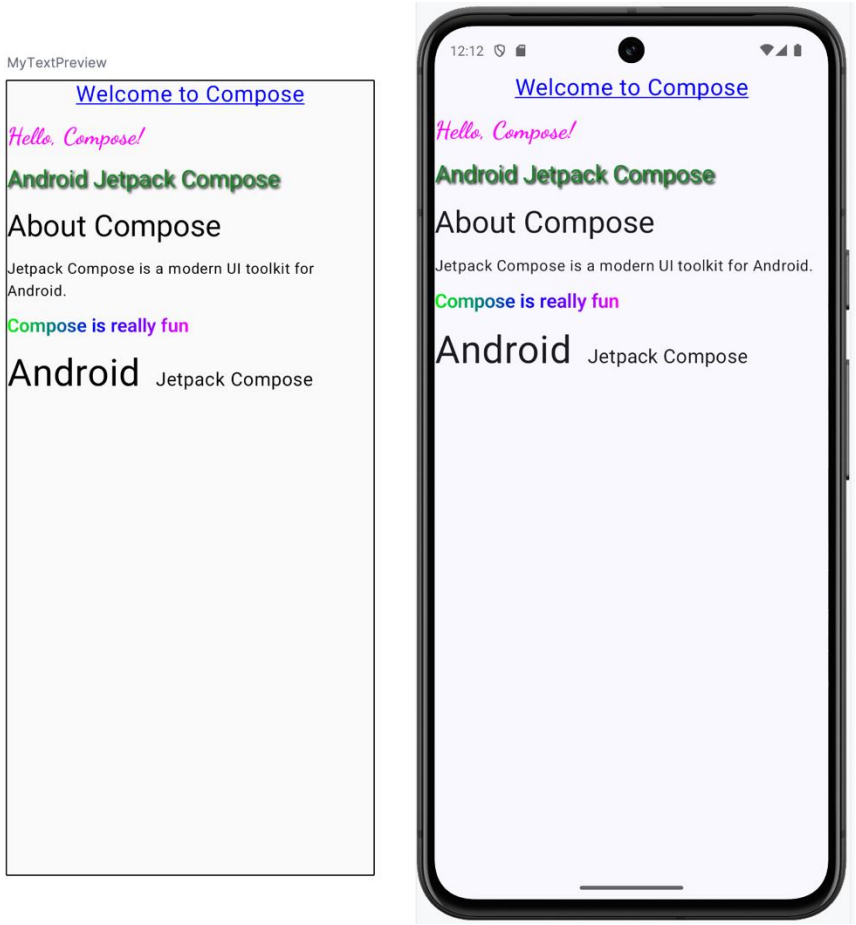
Text Style

```
Row {
    Text(
        text = "Android",
        fontSize = 32.sp,
    )
    Text(
        text = " Compose",
        fontSize = 16.sp,
    )
}
```



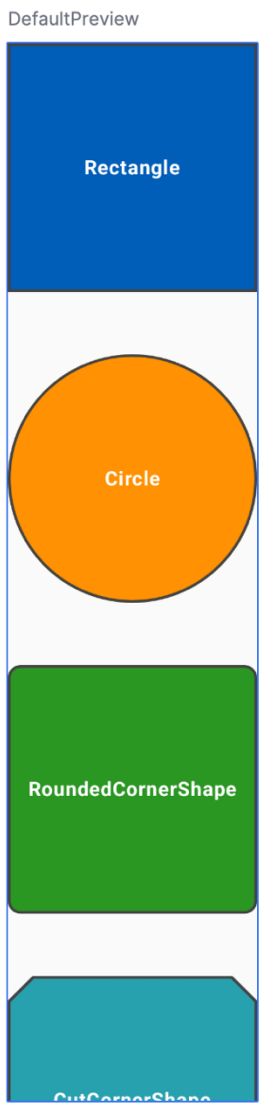

```
Row {
    Text(
        text = "Android",
        fontSize = 32.sp,
        modifier = Modifier.alignByBaseline()
    )
    Text(
        text = " Compose",
        fontSize = 16.sp,
        modifier = Modifier.alignByBaseline()
    )
}
```

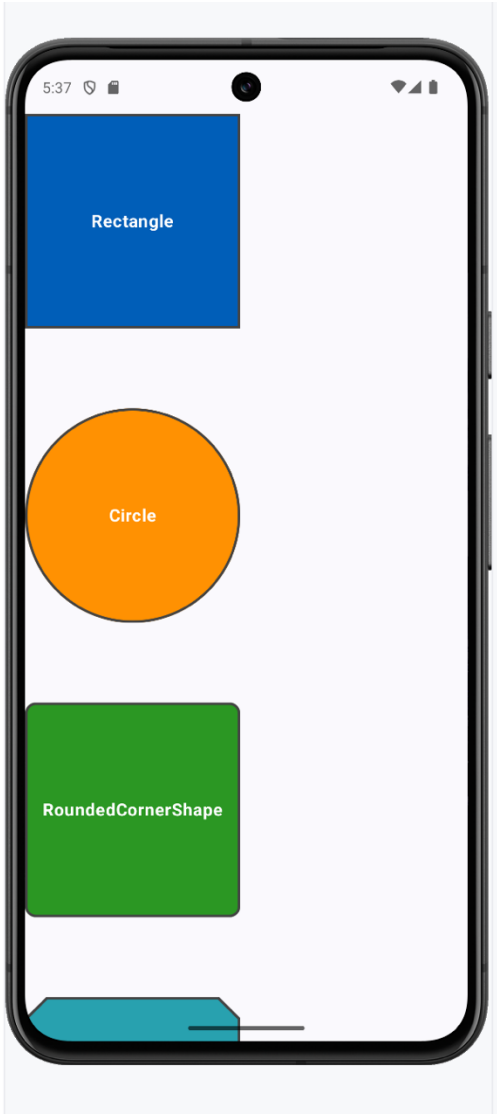




Text Styles

https://github.com/brightgeevarghese/ComposeTextStyles





Box Shapes

https://github.com/brightgeevarghese/ComposeBoxShapes

TextField Composable UI

- TextField is a user interface control that is used to allow the user to enter the text and modify the text.
- TextField is the Material Design implementation
- OutlinedTextField is the outline styling version
- BasicTextField – does not provide decorations like hint or placeholder
- Recommend to use TextField or OutlinedTextField
- TextField as a stateful composable function containing a state variable and an event handler that changes the state based on the user's keyboard input.
- The result is a text field in which the characters appear as they are typed.

TextField Overview

```
TextField(value = "",
    onValueChange ={},

    label = { Text("First Name")}
)

OutlinedTextField(
    value = "",
    onValueChange ={},
    label = { Text("Last Name")}
)
OutlinedTextField(
    value = "",
    onValueChange ={},
    placeholder = { Text("Last Name")}
)
```

In the above codes, user cannot enter the Inputs.

value → the input text to be shown in the text field.

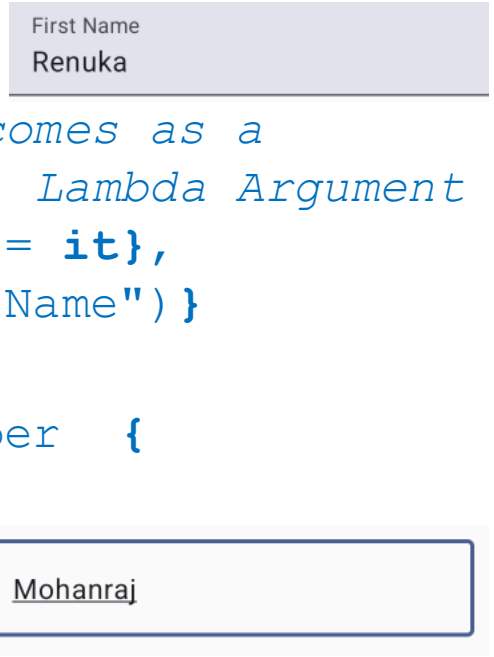
onValueChange → the callback that is triggered when the input service updates the text. An updated text comes as a parameter of the callback.

label --> the optional label to be displayed inside the text field container.

Placeholder -->the optional placeholder to be displayed when the text field is in focus and the input text is empty. Like hint.

TextField – How to interact the UI

```
var fname:String by remember  {
            mutableStateOf("")
}
TextField(value =fname,
 // it-->An updated text comes as a
parameter of the callback. Lambda Argument
    onValueChange ={fname = it},
    label = { Text("First Name")}
)
var lname:String by remember  {
       mutableStateOf("")
}

OutlinedTextField(value = lname,
    onValueChange ={lname=it},
    placeholder = { Text("Last Name")}
)
```



Key Concepts

- **State is any value that can change over time.**
- **mutableStateOf**
  - A function to create an observable state that triggers UI updates.
- **By Keyword (Delegation)**
  - Delegates the getter and setter of a property to the value.
- **remember**
  - Remember a value across recompositions.
- **Recomposing** simply means that the function gets called again and passed the new state value.

TextField – Maintain a state

State Variables in Compose

```
var fname: String by remember {  mutableStateOf("") }
```

- A mutable state variable named fname of type String.
- Initialized to an empty string.
- Automatically updates the UI when its value changes.

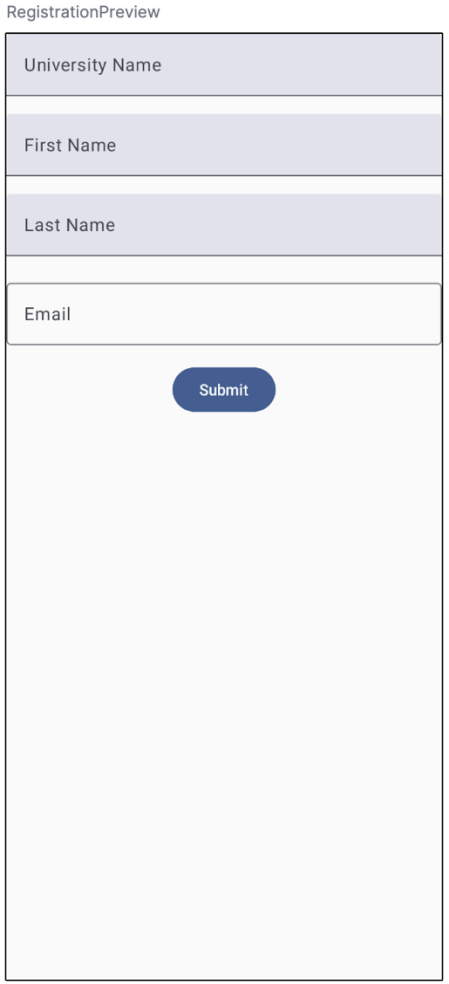
- **Primitive Type Optimizations**
  - Variants like mutableIntStateOf, mutableLongStateOf, etc., optimized for primitive types.

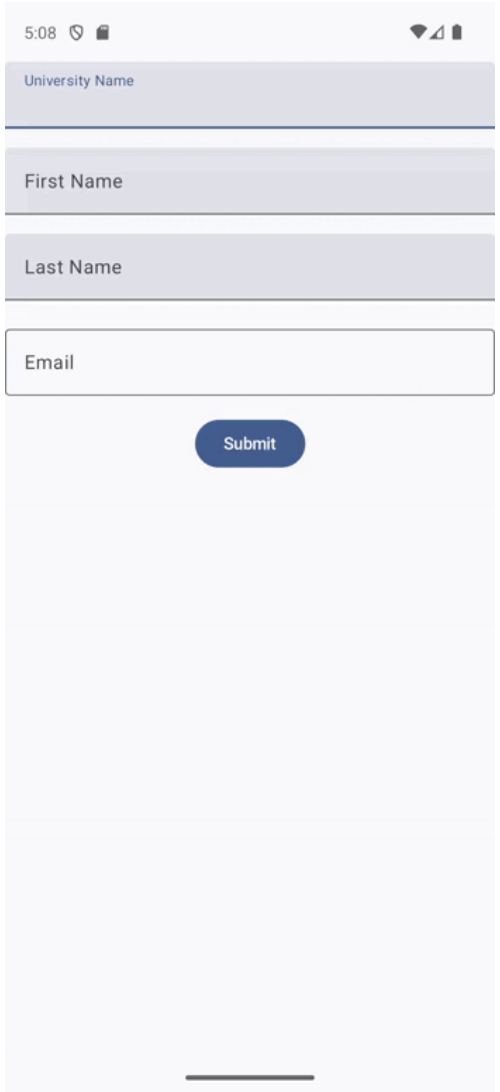
```
TextField(
   value = fname,
   onValueChange = { fname = it },
   label = { Text("First Name") }
)
```

- value is assigned to fname.
- onValueChange updates fname with the new text.

Imports needed

```
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.runtime.remember
```



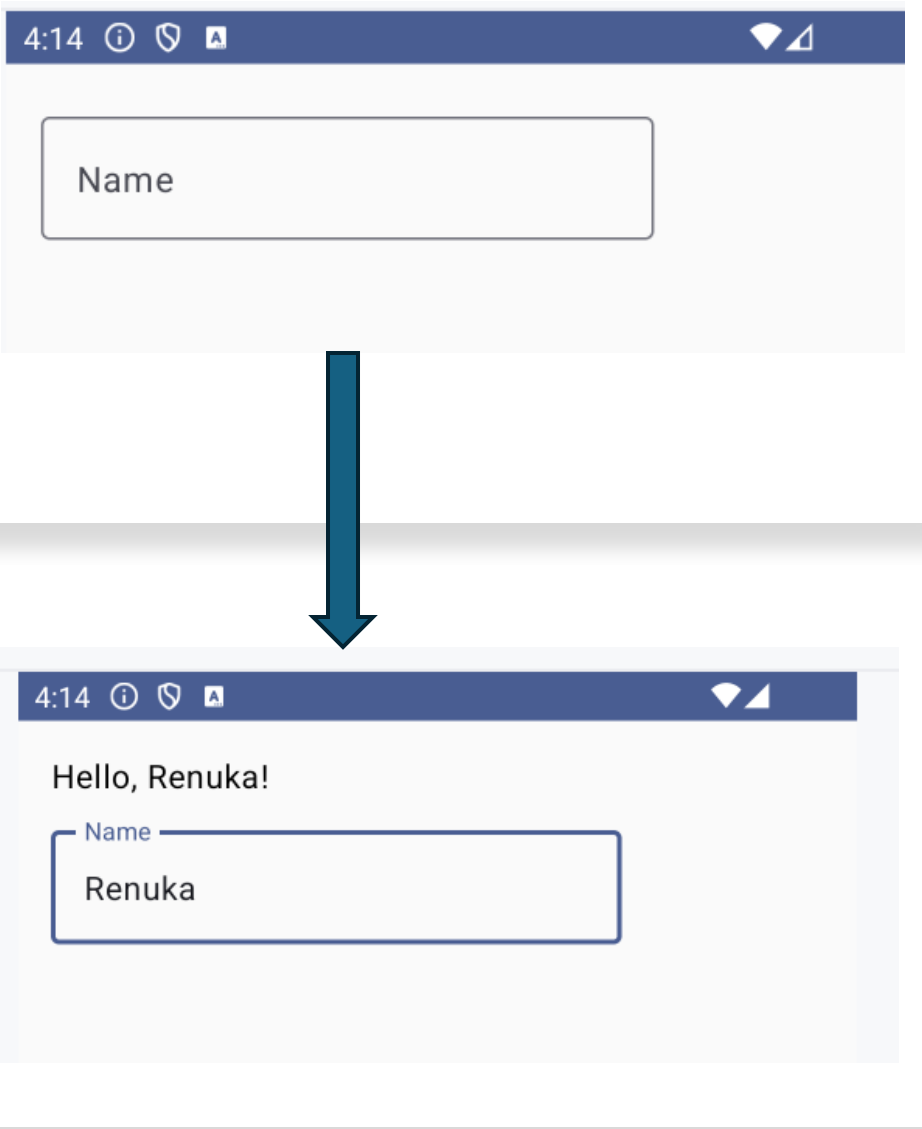


TextField & Button

https://github.com/brightgeevarghese/ComposeTextField

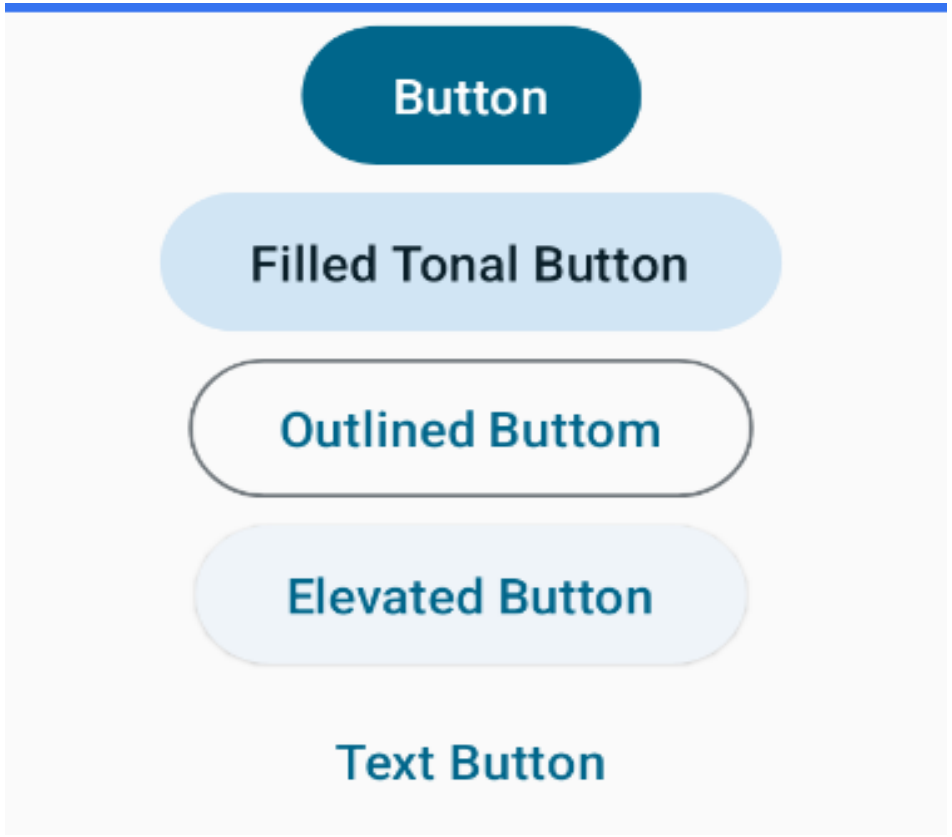
TextField Demo Code

```
@Composable
fun SayHello() {
    Column(modifier = Modifier.padding(16.dp)) {
        var name by remember { mutableStateOf("") }
        if (name.isNotEmpty()) {
            Text(
                text = "Hello, $name!",
                modifier = Modifier.padding(bottom = 16.dp),
                style = MaterialTheme.typography.bodyLarge
            )
        }
        OutlinedTextField(
            value = name,
            onValueChange = { name = it },
            label = { Text("Name") }
        )
    }
}
```



Button Composable UI

- Buttons are fundamental components that allow the user to trigger a defined action. There are five types of buttons.




```
@Composable
fun ColumnArrangement(){
  Column(modifier = Modifier.fillMaxWidth(),
      horizontalAlignment =  Alignment.CenterHorizontally,
     verticalArrangement = Arrangement.Center) {
     Button(onClick = {  }) {
        Text(text = "Button")
     }
     FilledTonalButton(onClick = {  }) {
        Text("Filled Tonal Button")
     }
     OutlinedButton(onClick = {  }) {
        Text("Outlined Buttom")
     }
     ElevatedButton(onClick = {  }) {
        Text("Elevated Button")
     }
     TextButton(
        onClick = {  }
     ) {
        Text("Text Button")
     }
  }
}
```

onClick: The function called when the user presses the button

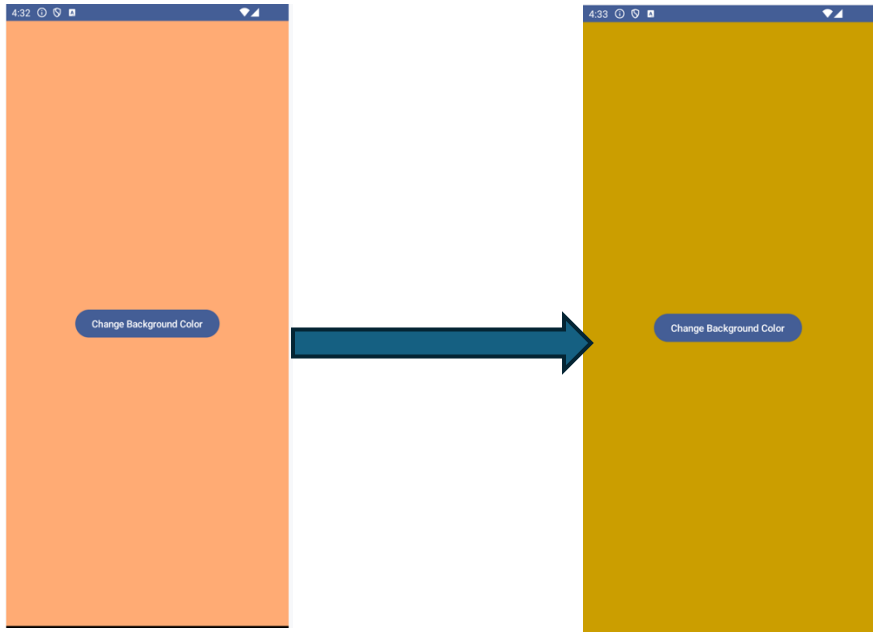
Event Handling

- Event handling is a fundamental part of interactive application development.
- It refers to the process of programming the application's response to user actions or system-generated events, such as clicks, key presses, a swipe etc.,
- The onClick event handler is a function that gets called when a user clicks on a Composable.
- It's often used to update some state or perform some action

Button Click Demo

Press/Click the button to change the background.

```
@Composable
fun ChangeBackgroundColor() {
    var color by remember { mutableStateOf(Color.White) }
    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(color),
        contentAlignment = Alignment.Center
    ) {
        Button(onClick = {
            color = Color(
                red = Random.nextFloat(),
                green = Random.nextFloat(),
                blue = Random.nextFloat()
            )
        }) {
            Text(text = "Change Background Color")
        }
    }
}
```

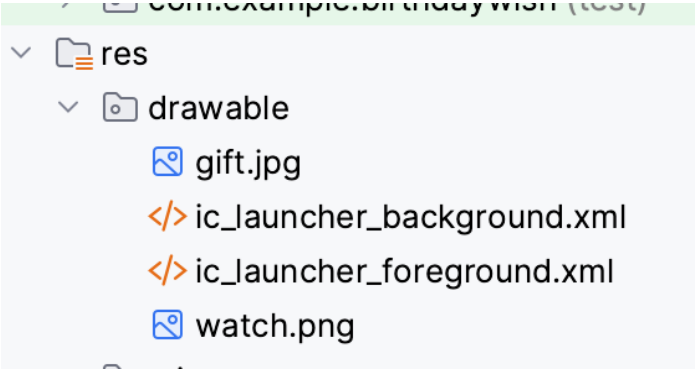


Loading Image

Use the Image composable to display a graphic on screen.

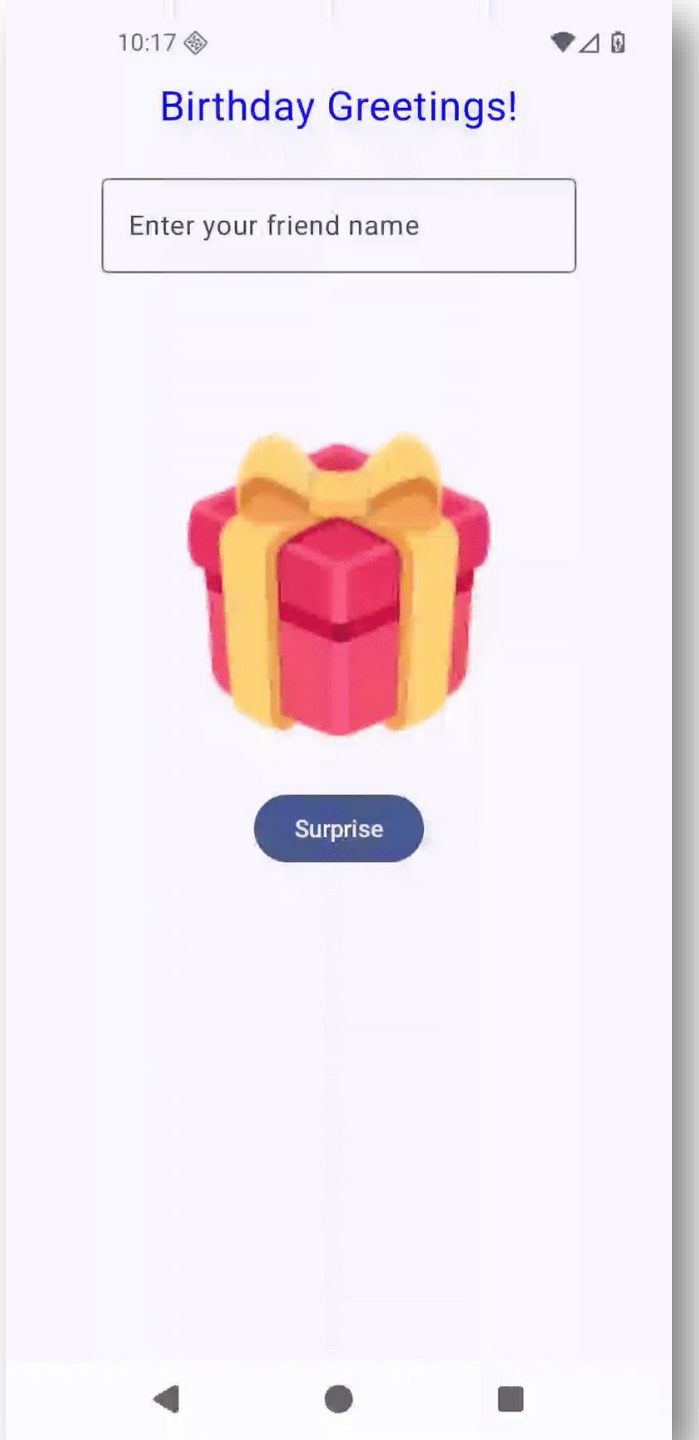
Syntax

```
Image(
    painter = painterResource(id = R.drawable.your_image_name),
    contentDescription = "Content description for accessibility",
    modifier = Modifier.size(200.dp) // can include modifiers
)
```



Copy the image into drawable resources and use the painterResource API with your image reference.

To ensure that your app is accessible, supply a contentDescription for visual elements on screen. TalkBack reads out the content description, so you must ensure that the text is meaningful if read out loud and translated.



Exercise: Birthday Wish App

Solution

Ref: https://github.com/brightgeevarghese/BirthdayWish/tree/main