COMP-304 Winter 2023

Review of Lecture 12

- Background work falls into one of three primary categories:
  1. **Immediate**: Needs to execute right away and complete soon.
  2. **Long Running**: May take some time to complete.
  3. **Deferrable**: Does not need to run right away.
- **WorkManager API**
  - Create a background task by extending **Worker** class.
  - Override **doWork** method
- Android Services
  - use the **Service** class.
  - Override **onBind** to bind an activity to a service, **onStartCommand** starts when the service is started, and **onDestroy** starts when the service is stopped.
  - Declare the service in manifest file:

```
<service
    android:name=".MyServ
    ice" />
```

Review of Lecture 12

- Use the startService() method, like this:

```
startService(new
    Intent(getBaseContex
    t(), MyService.class));
```

- Use the stopService() method to stop the service:

```
stopService(new
    Intent(getBaseContex
    t(), MyService.class));
```

- Performing Long-Running Tasks in a Service
  - Android uses MessageQueue, Looper, Handler, to perform a long-running task in a thread.
  - A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue.
  - When you create a new Handler it is bound to a Looper
  - **MessageQueue** is a Low-level class holding the list of messages to be dispatched by a Looper.

Intro to **Jetpack Compose**

Objectives:

- Discuss **Jetpack Compose architecture**
- **Create Composable Funtions**
- Develop Android **apps with Jet Compose UI**

Introduction to Jetpack Compose

- Jetpack Compose is a modern toolkit for building native Android UI.
  - Simplifies and accelerates UI development on Android with less code, powerful tools, and intuitive Kotlin APIs.
  - **Does not use any XML** layouts or the Layout Editor.
  - Jetpack Compose is built around composable functions
  - Developer calls **composable functions** to define what elements you want, and the **Compose compiler** will do the rest.

https://developer.android.com/jetpack/compose/tutoria

Composable functions

- These functions let you define your app's UI programmatically by describing how it should look and providing data dependencies, rather than focusing on the process of the UI's construction (initializing an element, attaching it to a parent, etc.).
- To create a composable function, just add the @Composable

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


https://developer.android.com/jetpack/compose/tutoria

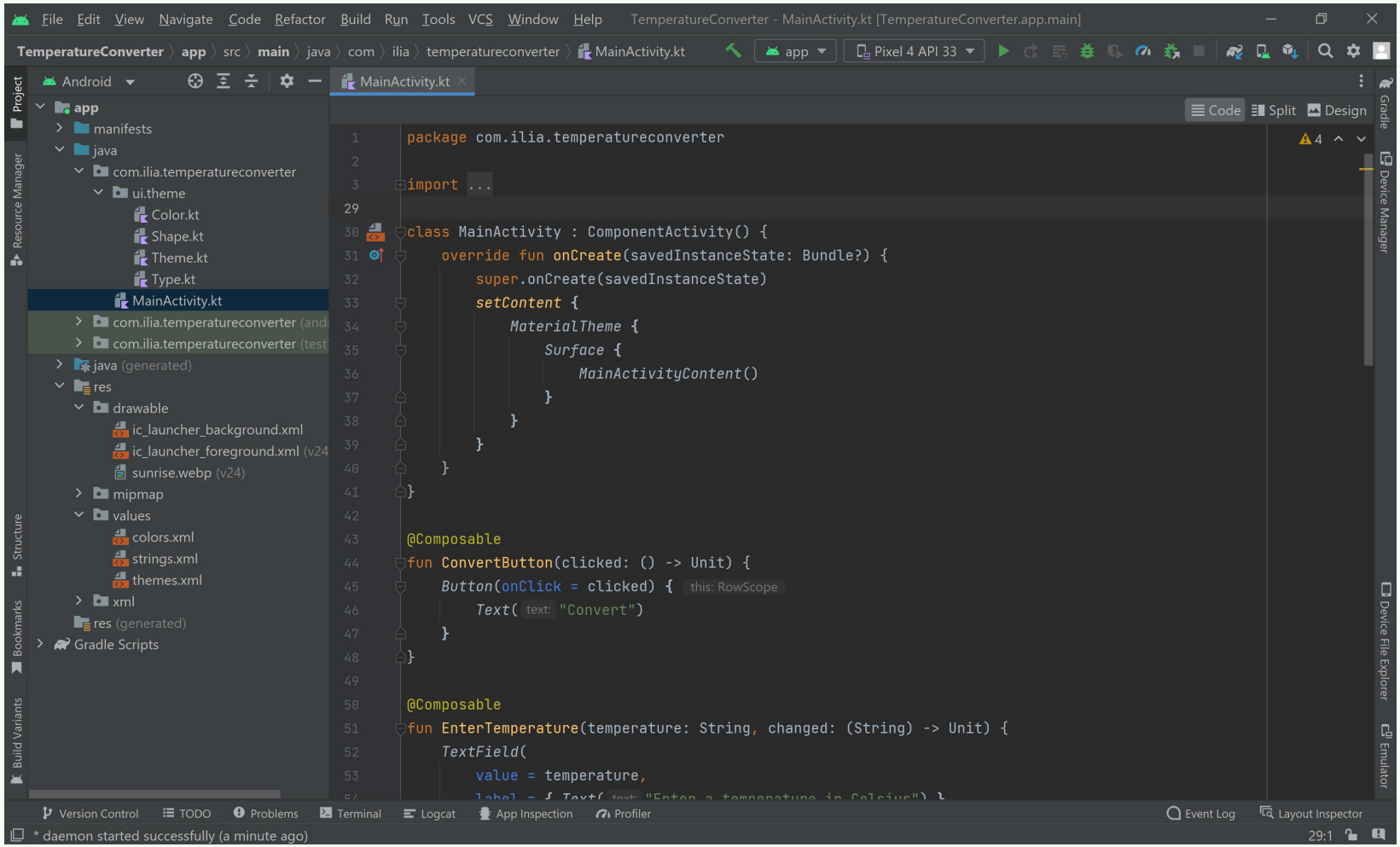
Add a text element

- Create a new app with support for Jetpack Compose:

1. With Android Studio project open, select **File > New > New Project** from the menu bar.
2. In the Select a Project Template window, select **Empty Compose Activity** and click Next.
3. In the Configure your project window, do the following:
   a) Set the Name, Package name, and Save location as you normally would.
   b) Note that, in the Language dropdown menu, Kotlin is the only available option because Jetpack Compose works only with classes written in Kotlin.
   c) In the Minimum API level dropdown menu, select API level 21 or higher.
4. Click Finish.
5. Verify that the project's **build.gradle** file is configured correctly,

https://developer.android.com/jetpack/compose/tutoria

Jetpack Compose Project



https://developer.android.com/jetpack/compose/tutoria

Jetpack Compose Project

- In the project structure, there is an extra package ***ui.theme***, for the app's theme, along with files for the theme's **colors**, **shapes**, and **typography**.
- The ***res*** package has a bunch of resource files, but **there are not any layout files**.
- **MainActivity** derives from **ComponentActivity**, not **AppCompatActivity**.
- The **onCreate** method calls **setContent** method to inflate the activity's layout.
- Note that we can use bothe composables and views (XML based) in the same UI (AppCompatActivity also derives from ComponentActivity class!)

Add a text element

- The default template already contains some Compose elements:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            JetComposeExample1Theme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colors.background
                ) {
                    Greeting("Android")
                }
            }
        }
    }
}
```

Add a text element

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

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    JetComposeExample1Theme {
        Greeting("Android")
    }
}
```

- Here the Text element is created in composable function Greeting.
- The **setContent** block defines the activity's layout where composable functions are called.
  - Composable functions can only be called from other composable functions.

https://developer.android.com/jetpack/compose/tutoria

Add a text element

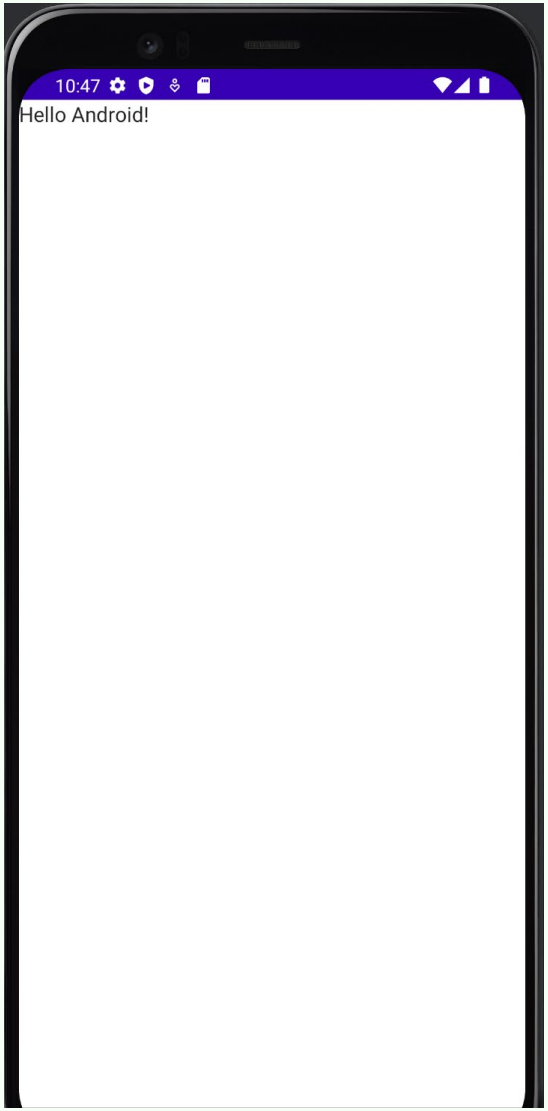
- If we remove the Material design portions of code, will have the following activity:

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

- Jetpack Compose uses a Kotlin compiler plugin to **transform these composable functions into the app's UI elements**.
- For example, the Text composable function that is defined by the Compose UI library displays a text label on the screen.

https://developer.android.com/jetpack/compose/tutoria

Add a text element



https://developer.android.com/jetpack/compose/tutoria

Add a text element

- Let's add a composable function to the activity on slide 9:

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

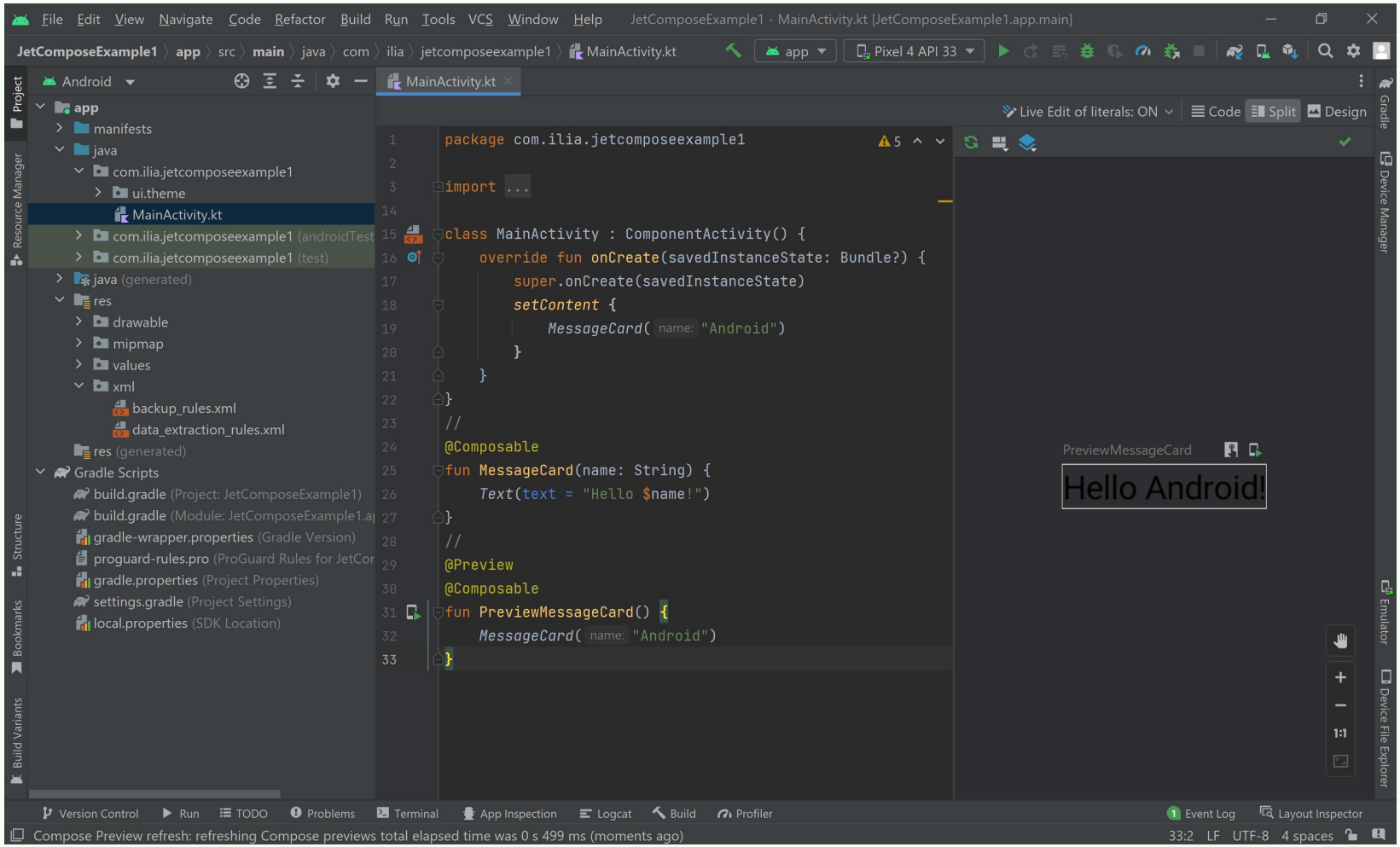
- The @Preview annotation lets you preview your composable functions within Android Studio without having to build and install the app to an Android device or emulator.

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

- Rebuilding the project will add a **preview window** which you can expand by clicking on the **split (design/code) view**.

https://developer.android.com/jetpack/compose/tutoria

Preview window - split (design/code) view



Layouts

- UI elements are hierarchical, with elements contained in other elements.
- In Compose, you build a UI hierarchy by calling composable functions from other composable functions.
- Let's illustrate this by adding:
  - Multiple texts
  - Images
  - Styling

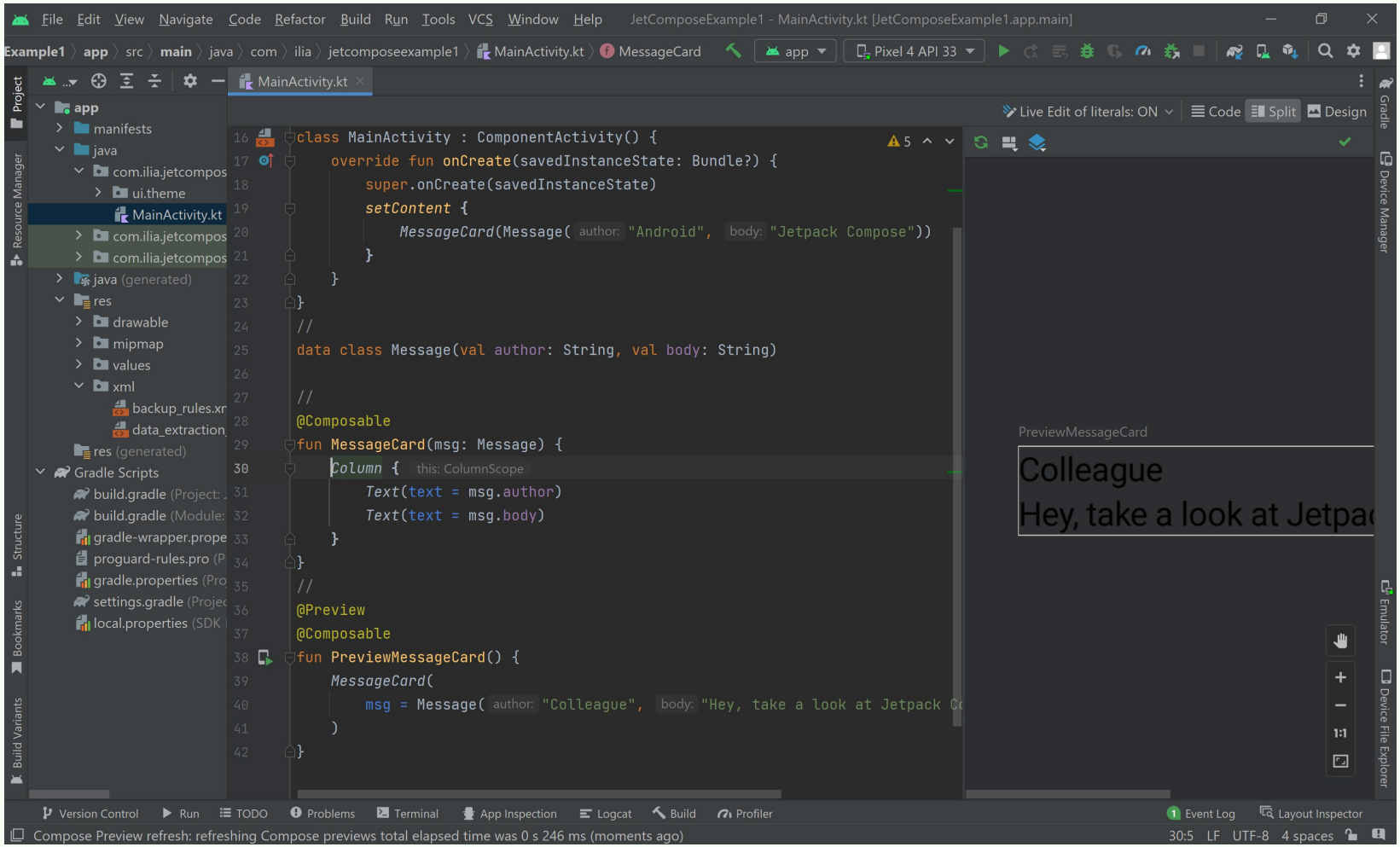
https://developer.android.com/jetpack/compose/tutoria

Adding multiple texts

- Let's make the message composable richer by displaying the name of its author and a message content.
  - first change the composable parameter to accept a Message object instead of a String
  - add another **Text** composable inside the MessageCard composable.
  - Make sure to update the preview as well.
  - Arrange elements vertically using a **Column** (or **Row**, to arrange items horizontally, or **Box** to stack elements).

https://developer.android.com/jetpack/compose/tutoria

Adding multiple texts



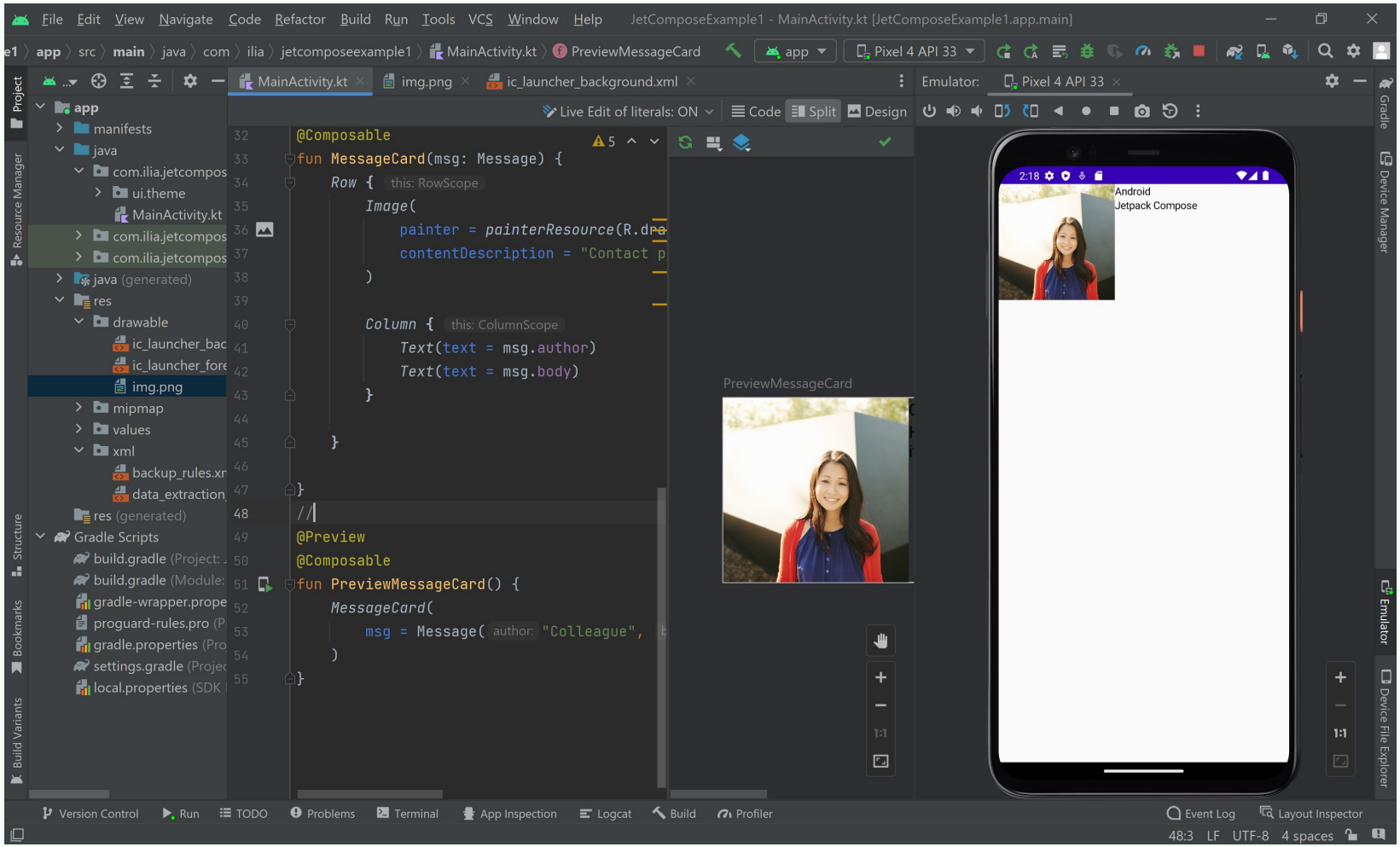
Add an image element

- Use the Resource Manager to import an image from your photo library or use another image.
- Add a **Row** composable to have a well structured design and an Image composable inside it:

```
@Composable
fun MessageCard(msg: Message) {
    Row {
        Image(
            painter = painterResource(R.drawable.img),
            contentDescription = "Contact profile picture",
        )
        Column {
            Text(text = msg.author)
            Text(text = msg.body)
        }
    }
}
```


https://developer.android.com/jetpack/compose/tutoria

Add an image element



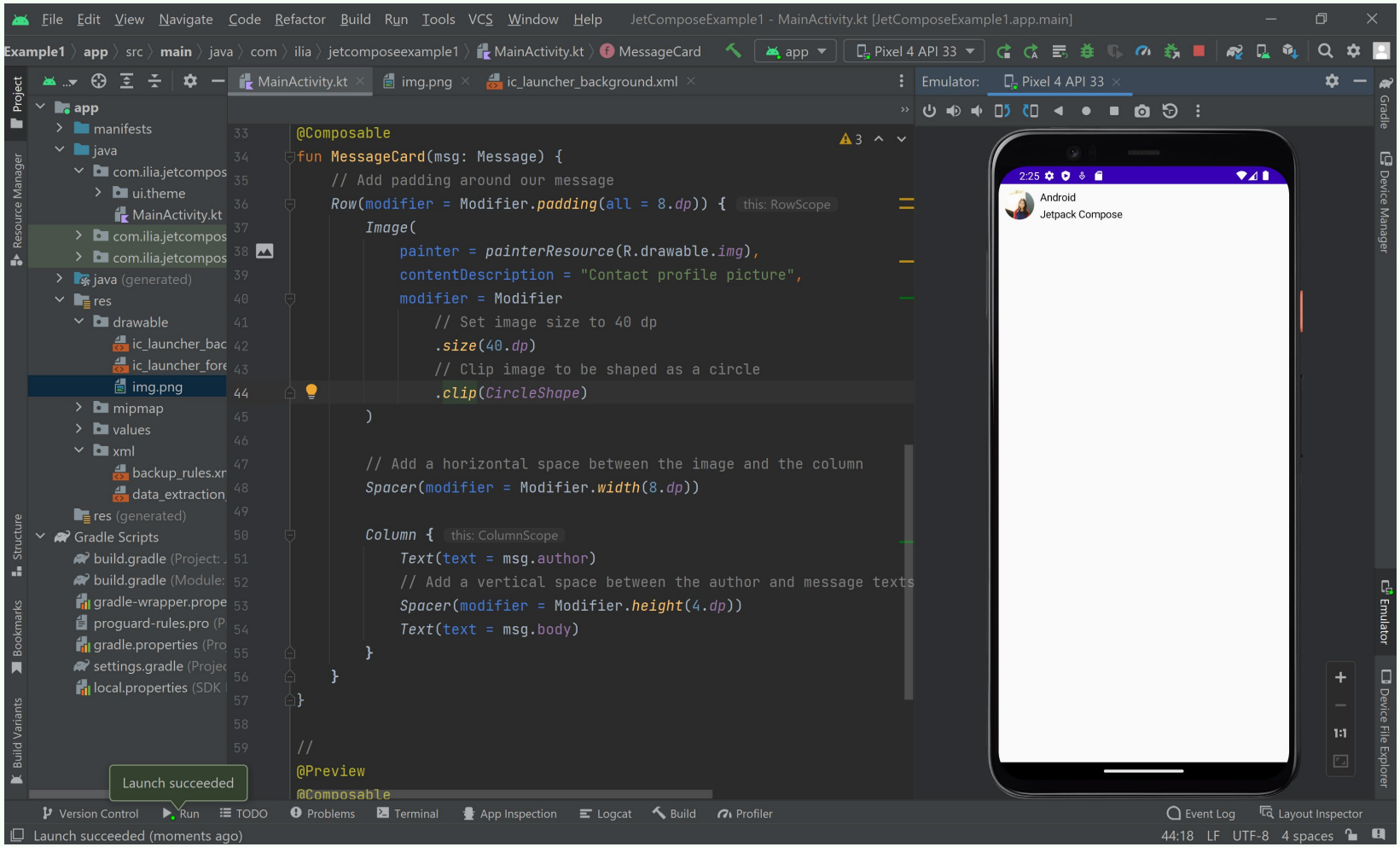
Configuring the layout

- To decorate or configure a composable, Compose uses modifiers.
- They allow you to change the composable's **size**, **layout**, **appearance** or add high-level **interactions**, such as making an element clickable.
- You can chain them to create richer composables.

```
// Add padding around our message
    Row(modifier = Modifier.padding(all = 8.dp)) {
        Image(
            painter = painterResource(R.drawable.img),
            contentDescription = "Contact profile picture",
            modifier = Modifier
                // Set image size to 40 dp
                .size(40.dp)
                // Clip image to be shaped as a circle
                .clip(CircleShape)
        )
```


https://developer.android.com/jetpack/compose/tutoria

Configuring the layout



Material Design

- Compose is built to support Material Design principles.
- Many of its UI elements implement Material Design out of the box.
- Material Design is built around three pillars: **Color**, **Typography**, and **Shape**.
- Let's improve the appearance of our MessageCard composable using Material Design styling.
- **Color**
  - Use **MaterialTheme.colors** to style with colors from the wrapped theme.
  - You can use these values from the theme anywhere a color is needed.
  - Style the title and add a border to the image.
- **Typography**
  - **Material Typography** styles are available in the MaterialTheme, just add them to the **Text** composables.

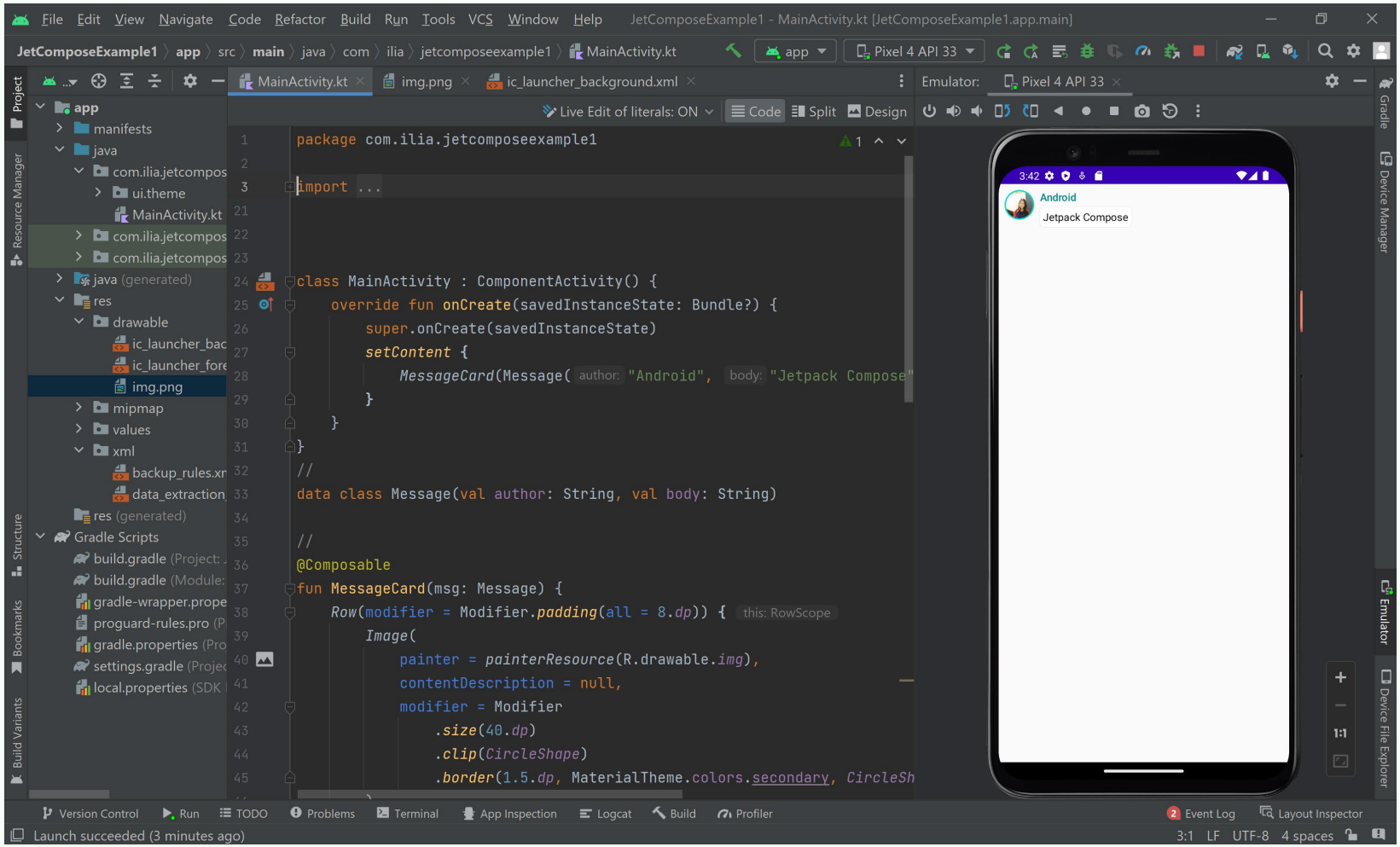
https://developer.android.com/jetpack/compose/tutoria

Material Design

- **Shape**
  - With **Shape** you can add the final touches.
  - First, **wrap the message body text** around a **Surface** composable - doing so allows customizing the message body's shape and elevation.
  - **Padding** is also added to the message for a better layout.
- **Enable dark theme**
  - Dark theme (or night mode) can be enabled to avoid a bright display especially at night, or simply to save the device battery.
  - Jetpack Compose can handle the dark theme by default.
  - Having used Material Design colors, text and backgrounds will automatically adapt to the dark background.

https://developer.android.com/jetpack/compose/tutoria

JetpackComposeExample



Lists and animations

- Compose makes it easy to create lists and fun to add animations.
- Let's create a Conversation function that will show multiple messages.
- For this use case, use Compose's **LazyColumn** and **LazyRow**.
- @Composable

```
fun Conversation(messages: List<Message>) {
    LazyColumn {
        items(messages) { message ->
            MessageCard(message)
        }
    }
}
```

- These composables render only the elements that are visible on screen, so they are designed to be very efficient for long lists.

https://developer.android.com/jetpack/compose/tutoria

Lists and animations

- **LazyColumn** has an items child.
- It takes a **List** as a parameter and its lambda receives a parameter we've named message (we could have named it whatever we want) which is an instance of Message.
- In short, this lambda is called for each item of the provided **List**.

https://developer.android.com/jetpack/compose/tutoria

Lists and animations

- We can add the ability to expand a message to show a longer one, **animating both the content size and the background color**.
- To store this local UI state, you **need to keep track of whether a message has been expanded or not**.
- To keep track of this state change, you have to use the functions **remember** and **mutableStateOf**.

```
// We keep track if the message is expanded or not in this
// variable
var isExpanded by remember { mutableStateOf(false) }
```

- Composable functions can store local state in memory by using remember, and track changes to the value passed to **mutableStateOf**.
- Composables (and their children) using this state will get redrawn automatically when the value is updated.
- This is called **recomposition**.

https://developer.android.com/jetpack/compose/tutoria

Lists and animations

- By using Compose's state APIs like remember and **mutableStateOf**, any changes to state automatically update the UI.
- You can change the background of the message content based on **isExpanded** when we click on a message.
- You will use the clickable modifier to handle click events on the composable.
- You will animate the background color by gradually modifying its value from **MaterialTheme.colors.surface** to **MaterialTheme.colors.primary** and vice versa.
- To do so, you will use the **animateColorAsState** function.
- Lastly, you will use the **animateContentSize** modifier to animate the message container size smoothly:

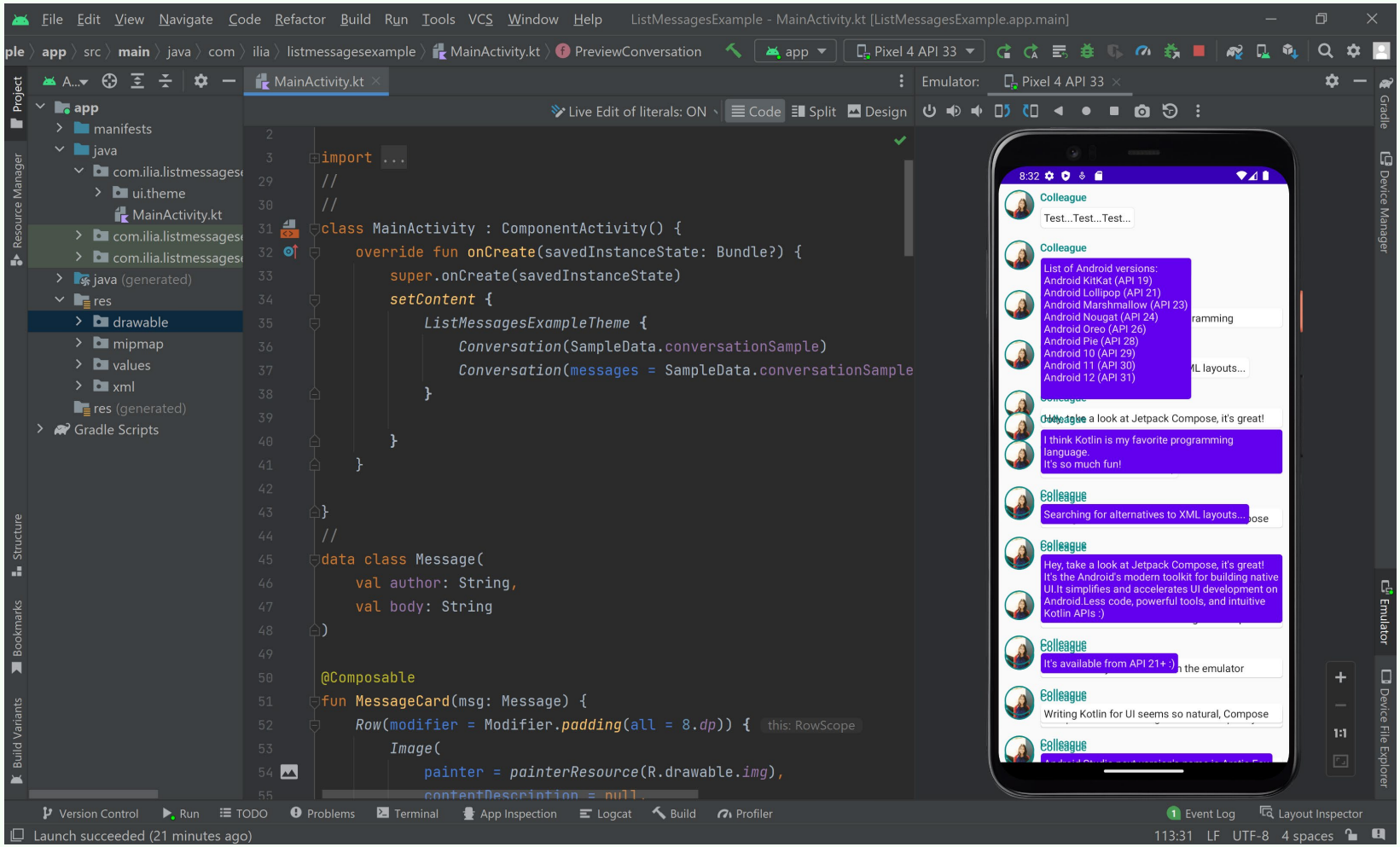
https://developer.android.com/jetpack/compose/tutoria

Lists and animations

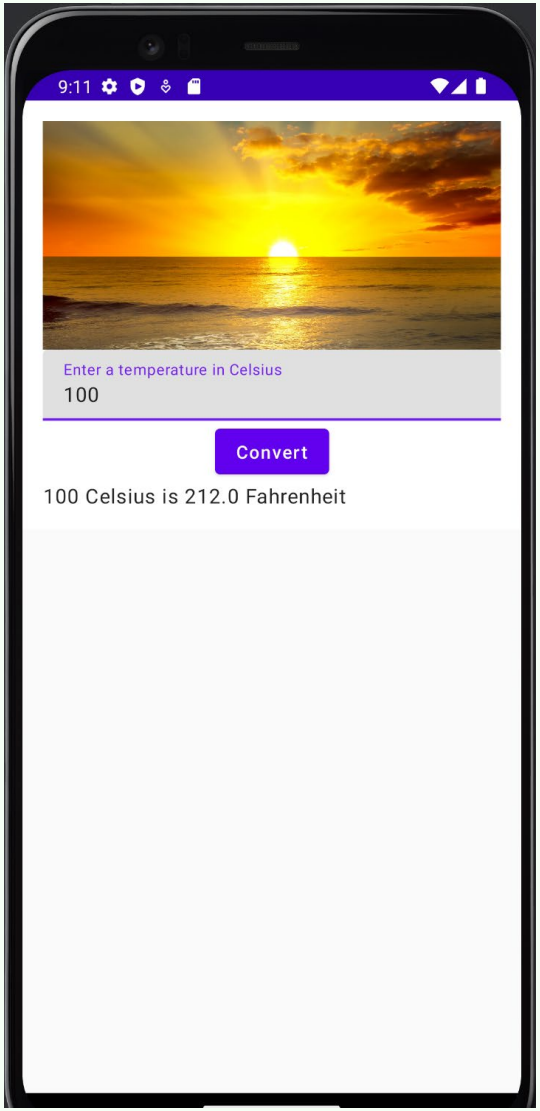
```
….
// surfaceColor will be updated gradually from one color to the other
 val surfaceColor by animateColorAsState (
  if (isExpanded) MaterialTheme.colors.primary else
MaterialTheme.colors.surface,
        )
…
// surfaceColor color will be changing gradually from primary to
surface
color = surfaceColor,
// animateContentSize will change the Surface size gradually
modifier = Modifier.animateContentSize().padding(1.dp)
```


https://developer.android.com/jetpack/compose/tutoria

ListMessagesExample



TemperatureConverter Example



TemperatureConverter Example

- Build a UI that lets you enter a temperature in degrees Celsius.
- Create UI components as Composable functions:

```
@Composable
fun ConvertButton(clicked: () -> Unit) {
    Button(onClick = clicked) {
        Text("Convert")
    }
}
@Composable
fun EnterTemperature(temperature: String, changed: (String) -> Unit) {
    TextField(
        value = temperature,
        label = { Text("Enter a temperature in Celsius") },
        onValueChange = changed,
        modifier = Modifier.fillMaxWidth()
    )
}
```

TemperatureConverter Example

```
@Composable
fun Header(image: Int, description: String) {
    Image(
        painter = painterResource(image),
        contentDescription = description,
        modifier = Modifier
            .height(180.dp)
            .fillMaxWidth(),
        contentScale = ContentScale.Crop
    )
}
@Composable
fun TemperatureText(celsius: Int) {
    val fahrenheit = (celsius.toDouble()*9/5)+32
    Text("$celsius Celsius is $fahrenheit Fahrenheit")
}
```

TemperatureConverter Example

```
@Composable
fun MainActivityContent() {
    val celsius = remember { mutableStateOf(0) }
    val newCelsius = remember { mutableStateOf("") }
    Column(modifier = Modifier.padding(16.dp).fillMaxWidth()) {
        Header(R.drawable.sunrise, "sunrise image")
        EnterTemperature(newCelsius.value) { newCelsius.value = it }
        Row(modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.Center) {
            ConvertButton {
                newCelsius.value.toIntOrNull()?.let {
                    celsius.value = it
                }
            }
        }
        TemperatureText(celsius.value)
    }
}
```

TemperatureConverter Example

```
@Preview(showBackground = true)
@Composable
fun PreviewMainActivity() {
    MaterialTheme {
        Surface {
            MainActivityContent()
        }
    }
}
```

Button Composable

- You add a Button using **Button** composable:

  Button(**onClick**={ // pass onClick a lambda to specify what happens when the button is clicked

  // code that runs when clicked

  }) {**Text**("Button Text") } // text that appears on the button

- The TemperatureConverter example uses a new composable function(named ConvertButton) that displays a Button:

```
ConvertButton (clicked: () -> Un it {
        Button (onClicked = clicked){
            Text("Convert")
        }
}
```


android

References

- Reference Textbook, Head First Android Development Third Edition
- Android Documentation: https://developer.android.com/jetpack/compose/tutorial