Android Mobile Application Development
Week 1: Android Development Intro

Now that we've seen the basic structure of an Android app, let's look at how we can build a basic Android app.

Many application development user interface frameworks are moving from an imperative design approach to a declarative design approach.

Imperative UI design usually involves the developer creating view-based user interfaces, often in XML-based layout files, and then connecting the UI elements to code that handles all the logic the interactivity along with updating the UI elements to reflect the current state.

In declarative UI design the developer designs the user interface in code solely based on how they want to appear at all times. Declarative UI frameworks include the ability to handle the state of saved data and automatically update the UI when that state changes. The code describes what the UI should look like, not how to update and manage it. So developers are no longer responsible for implementing the logic to update the UI.

- Advantages
    - Avoids many errors introduced connecting views with code
    - Reduces code needed to manage the UI
    - Eliminates codes for state management
- Disadvantages
    - Different way to think about the user interface and logic
    - Only supports more recent operating systems
    - Smaller developer community

Along with Google, Apple and Meta have created declarative UI frameworks with SwiftUI and react/React Native, respectively.

Jetpack Compose

Jetpack Compose is Google's modern declarative UI toolkit for Android that replaces Android's built-in view-based framework UI toolkit. Compose is designed in Kotlin and is exclusively available in Kotlin (not available in Java). Compose version 1.0 went stable in the summer of 2021 and is considered the best method for designing native Android apps. Compose is compatible with API 21 and above and can also be integrated in view-based apps as existing apps make the transition. Because Compose is part of the Jetpack suite of libraries it is separate from the Android OS and can be updated separately as you would with any external library.

Composables

In Jetpack Compose composables are the building blocks of your UI. A composable is a normal Kotlin function annotated with `@Composable` annotation.

Jetpack Compose provides composables to create various UI components. You can also create your own composables.

A composable function is a function that uses one or more composables to define part of the UI. It does this using a Kotlin compiler plugin to transform composable functions into the app's UI elements.

A composable function does not have a return value, instead it creates elements that are added to the user interface.

Composable functions can call other composables as well as standard Kotlin functions. Standard Kotlin functions cannot call composable functions, they can only be called from other composable functions.

Built-in Composables

Jetpack Compose includes many composable functions to help you create your UI. We've already seen one composable function, and we'll look at 3 more today
https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#top-level-functions

Text
https://developer.android.com/jetpack/compose/text
The Text function displays text
- text parameter takes a string of the text to be displayed
  - the only required parameter
- The Text composable has many optional parameters to style its content
- fontSize must use the sp unit for scale-independent pixels so it scales based on the user's font size setting.
  - if you do not specify a unit, you will see a compiler error, because Int cannot be converted to sp automatically.

Image
https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary#top-level-functions
https://m2.material.io/design/communication/imagery.html
The Image function lays out and draws an image.
- The painter parameter specifies the image that should be displayed and is required
  - Painter declares something that can be "painted" to the screen, like a vector image, bitmap image, solid color, or gradient.
- The contentDescription parameter is descriptive text for the image that's used for accessibility and is also required
- The Image composable has many optional parameters to customize the image.

Modifiers
https://developer.android.com/jetpack/compose/modifiers
Modifiers let you add extra behavior or decoration to composables by defining properties for composables. Modifier is a Kotlin class built into Compose which allows a wide range of properties to be set on a composable within a single object. With modifiers you can:
- 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

It's best practice to have all your Composables accept a modifier parameter.
- Parameter is expected to be named modifier
- The official convention for the modifier parameter is to place it after the required parameters before the other optional parameters

Modifiers can be changed, but the order of modifiers matters. Modifiers make use of the builder pattern to set the properties so changes are applied in the order of which they were set in the modifier object. Modifiers are invoked from top to bottom. Each makes its contribution to the appearance of the composable, and then the next modifier is applied inside it. Once all modifiers have been applied, the content of the composable is placed inside the final modifier.

Jetpack Compose provides many built-in modifiers.
- Padding adds space all around an element
  - All, top, bottom, start, end, vertical, horizontal
  - Use dp for density-independent pixels
- Size lets you set the width and height
- Offset lets you position a layout relative to its original position
  - Offset does not change the measurements of a composable as padding does

There are many other that we will explore as well.

Compose Layout
https://developer.android.com/jetpack/compose/layouts/basics

When you create a layout with several UI elements you need to tell Compose how to arrange them. If you don't, the UI elements will be stacked on top of each other in the top left corner of the screen. Compose provides a collection of layouts to help arrange your UI elements.
The Compose layout system is based on rows and columns.

Column
The column composable places items vertically on the screen.

Row
The Row composable places items horizontally on the screen.

Box
The Box composable stacks items on top of each other.

There are additional arguments for layout and positioning. You can also combine and nest these for more elaborate layouts.
There are others we'll be looking at during the semester as well.

R class
"R" stands for resources. The R class acts as an index to all of the resources identified in your project. The R.java file is automatically generated for every Android project and generates resource IDs for all your app resources. Any time you change, add, or remove a resource, the R class is automatically regenerated.

Resources
https://developer.android.com/jetpack/compose/resources

Resources are the additional files and static content that your app uses, such as images, user interface strings, icons, audio etc.
- Instead of hardcoding text values, you should use string resources in the res/strings.xml resource file
  - Easier to make changes
  - Localization
  - The stringResource() function returns a string value from the strings resource file
    - `stringResource(R.string.stringName)`
- Images are added to the res/drawable folder in your project. You can create folders to hold images for different screen size densities.
  - The painterResource() function will load an image and convert it into a Painter that can be used with your Image.

- painterResource(id = R.drawable.sunrise)
- painterResource can load Bitmap, drawable and VectorDrawable(Vector, png, webp, etc) types

HelloAndroid

Open your HelloAndroid app and let's take a closer look at it.

Open MainActivity.kt.

An activity is a single module of application functionality
- Usually associated with one screen
- Written in Java or Kotlin

MainActivity extends ComponentActivity which is a subclass of the Activity class, and it's used to define a basic activity that uses Compose for its UI instead of a layout file.

The onCreate() method is the first method called when an activity is launched. setContent{} is an extension function that's used to add Compose components, composables, to an activity's UI so they're rendered when the activity gets created.

The first thing you see in setContent{} is a composable function named HelloAndroidTheme{}. This composable function is declared in the Theme.kt located in the ui.theme package. This, along with the other files in the ui.theme package defines the colors, fonts, and shapes to be used by the activity and provides a central location from which to customize the overall theme of the app's user interface. We will look at that more next week.

The HelloAndroidTheme composable function contains a Surface composable. Surface is a built-in Compose component designed to provide a background for other composables.

The Surface composable contains a composable function named Greeting which is passed a string value of "Android".

To simplify this for our first app remove everything from setContent{} other than Greeting().

```
setContent {
    Greeting("Android")
}
```

Outside of the scope of the MainActivity class is the composable function Greeting() which is marked as being composable by the *@ Composable* annotation. The function accepts a parameter called *name* of type String. Greeting() calls the built-in Text composable, passing through a string value containing the word "Hello" concatenated with the name parameter.

We will also simplify our Preview composable to just call Greeting().

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

Preview
Go into Split mode and click on build and refresh or Build | Make Project
Preview let you quickly see the layout of your activity without having to compile and run the app. Every time you change your layout you will need to build and refresh.
Including a background improves visibility.
If you change the string passed to the Greeting() function in the DefaultPreview() function you will see this reflected in the preview.
More significant changes will require a build and refresh before being reflected in the preview.
Preview functions cannot take in any parameters.
You can define multiple Preview functions.
The Preview tool also supports interactive mode which allows you to interact with the UI.

String Resources
It's not good practice to hard code string values in your code, instead you should use string resources.
strings.xml in the res/values folder stores all the string resources.

Open strings.xml and you'll see there's already a string resource for the name of the app.
Let's add another string resource to use instead of "Hello". Start typing the xml but then hit return or tab to accept the autofill.
```xml
<string name="greeting">Hello </string>
```

Code completion
The editor in Android Studio has code completion. As you type you will get selections. To accept the top most suggestion, press the Enter or Tab key on the keyboard. To select a different suggestion, use the arrow keys to move up and down the list, once again using the Enter or Tab key to select the highlighted item.
To see other suggestions, click on a word and then use cntrl-space and the editor will show you a list of alternative suggestions.

Text
Back in MainActivity replace "Hello" to use our new string resource.

```kotlin
Text(stringResource(R.string.greeting) + " " + name)
```

Refresh the Preview.
We should also change the string "Android" to be a string resource. In strings.xml add
```xml
<string name="scotty">Scotty Dog</string>
```

Then update the call to Greeting in onCreate() as well as DefaultPreview() to
```kotlin
Greeting(stringResource(R.string.scotty))
```
Look at the documentation for the Text composable to see all the optional parameters to style its content. You can hover over Text and its documentation should popup, or command-click to open the documentation, or open the documentation tab. The documentation also pops up when you start typing Text.
We can add as many parameters as we want.
Let's start with color.
```kotlin
Text(stringResource(R.string.greeting) + " " + name, color = Color.Red)
```

Now let's change the font size. Font size always use the sp unit for scale-independent pixels so it scales based on the user's font size setting.

```
fontSize = 24.sp
```

You might be prompted for an import, option enter will add the needed import (or give you choices if there are multiple).

```
import androidx.compose.ui.unit.sp
```

Lastly let's align the text within the Text composable area. This is different than positioning the Text composable on the screen.

```
textAlign = TextAlign.Center
```

If you noticed that both Color and TextAlign are capitalized that's because they are classes in the Android SDK and we use dot notation to access predefined values.
This gives you an idea of how you can use some of the styling parameters for the Text composable.

You need to add imports manually or have Android Studio add them automatically.

1. Open the **Settings** or **Preferences** dialog:
   - On Windows or Linux, select **File > Settings** from the menu bar.
   - On macOS, select **Android Studio > Preferences** from the menu bar.
2. Navigate to Editor | General | Auto Import | Kotlin and check "Add unambiguous imports on the fly" and "Optimize imports on the fly"

ImageView
Drag and drop or copy and paste an image into the drawable folder (not drawable-24). (scottydog.jpg)
The R class uses the name of the drawable resource so make sure this matches the name you gave it in Android Studio.
For the content description we can use the same string resource that we just added.

Let's add an Image below our Text.

```
Image(painter = painterResource(id = R.drawable.scottydog),
contentDescription = stringResource(id = R.string.scotty) )
```

Refresh the preview.
Where's the text? If you hover over the preview you'll see a box where the text is, but the image is over it. This is because if you don't tell Compose how to layout the UI elements they will be stacked on top of each other in the top left corner of the screen.
So let's put these composables in a column so they're layed out vertically.

```
Column {
    Text(stringResource(R.string.greeting) + " " + name)
    Image(
        painter = painterResource(id = R.drawable.scottydog),
        contentDescription = stringResource(id = R.string.scotty)
    )

}
```

Not beautiful but at least we can see both the text and the image.

The Image composable also has optional parameters to customize the image.
The contentScale parameter is used to determine the aspect ratio scaling to be used if the bounds are a different size from the intrinsic size of the image. Right now this wouldn't change anything, but it will be useful in the future.
The alignment parameter used to place the image in the given bounds defined by the width and height. Right now the image is determining the bounds so this also wouldn't change anything at this point.
Let's use the alpha parameter to change the images opacity. Note this parameter accepts a float and you'll be prompted to add the F after the number.

```
alpha = 0.5F
```

<u>Layout</u>
Let's make this look better using layout elements and modifiers.
I moved the Image above the Text and removed the opacity parameter.
Let's add a modifier to our image as the third parameter, after the required parameters.
I'm going to make the image larger using height.

```
modifier = Modifier
    .height(190.dp)
```

Then I want it to use the full width of the device so I'll chain fillMaxWidth.

```
modifier = Modifier
    .height(190.dp)
    .fillMaxWidth()
```

Now I'd like a little padding above and below the image which is top and bottom.

```
modifier = Modifier
    .height(190.dp)
    .fillMaxWidth()
    .padding(top = 40.dp, bottom = 40.dp)
```

Why does the image look smaller?
Remember that the way chaining works is each modifier in the chain is applied in order. So the padding is being applied after the height and fillMaxWidth. Let's put the padding at the top of the chain.

```
modifier = Modifier
    .padding(top = 40.dp, bottom = 40.dp)
    .height(190.dp)
    .fillMaxWidth()
```

That looks much better. I thought it would be fun to make the image a circle which can be done with the clip modifier.

```
.clip(CircleShape)
```

But that didn't work. The height was forcing it to be that height so I changed to use the size modifier instead.

```
modifier = Modifier
    .padding(top = 40.dp, bottom = 40.dp)
    .size(190.dp)
```

```
        .fillMaxWidth()
        .clip(CircleShape)
```

That resulted in the circle but now it wasn't centered so I added the align modifier.
```
.align(Alignment.CenterHorizontally)
```

That worked. And you don't need the fillMaxWidth modifier. So the final result.
```
modifier = Modifier
        .padding(top = 40.dp, bottom = 40.dp)
        .size(190.dp)
        .clip(CircleShape)
        .align(Alignment.CenterHorizontally)
```

Now let's fix the Text.
First I'd like the Text composable to use the max width so it's centered.
```
modifier = Modifier
        .fillMaxWidth(),
```

A good way to see what is actually being drawn is to give it a background color.
```
modifier = Modifier
        .fillMaxWidth()
        .background(Color.Black),
```

I'd like to see more of the background above and below the text.
Any thoughts on how I can achieve this?
I added the height modifier.

```
modifier = Modifier
        .fillMaxWidth()
        .height(60.dp)
        .background(Color.Black),
```

But the text isn't centered vertically and in a column you can center elements horizontally but not vertically. And in a Row you can center it vertically but not horizontally.
How could I fix this?
I found two approaches.
Instead of height I could add padding around the text. And this time I do want the padding last because I want it applied after the background.
```
    modifier = Modifier
        .fillMaxWidth()
        .background(Color.Black)
        .padding(top = 20.dp, bottom = 20.dp),
```

The second approach was to use a Box as center alignment is both vertical and horizontal.
```
        Box(
            contentAlignment = Alignment.Center,
            modifier = Modifier
                .fillMaxWidth()
                .height(60.dp)
                .background(Color.Black)
```

```
    ) {
    Text(
        stringResource(R.string.greeting) + " " + name,
        color = Color.Red,
        fontSize = 24.sp,
        textAlign = TextAlign.Center
    )
}
```

Note that within the modifier you use a period to chain the modifiers. But the modifier is a parameter so you still need a comma between parameters.

Apply Changes

To run your app again in the emulator you have a few choices.

- **Run** deploys all changes and restart the application.
    o Use this option when the changes that you have made cannot be applied using either of the Apply Changes options.
- **Apply Changes and Restart Activity** ⟳ will attempt to apply your resource and code changes and restart only your activity without restarting your app.
    o Generally, you can use this option when you've modified code in the body of a method or modified an existing resource.
- **Apply Code Changes** ⟱ will attempt to apply only your code changes without restarting anything.
    o Generally, you can use this option when you've modified code in the body of a method but you have not modified any resources.

Enable Run fallback for Apply Changes

After you've clicked either **Apply Changes and Restart Activity** or **Apply Code Changes**, Android Studio builds a new APK and determines whether the changes can be applied. If the changes can't be applied and would cause Apply Changes to fail, Android Studio prompts you to Run your app again instead. However, if you don't want to be prompted every time this occurs, you can configure Android Studio to automatically rerun your app when changes can't be applied.

To enable this behavior, follow these steps:

3. Open the **Settings** or **Preferences** dialog:
    o On Windows or Linux, select **File > Settings** from the menu bar.
    o On macOS, select **Android Studio > Preferences** from the menu bar.
4. Navigate to **Build, Execution, Deployment > Deployment**.
5. Select the checkboxes to enable automatic Run fallback for either of the Apply Changes actions.
6. Click **OK**.

Note: To open a project in Android Studio there is no one file you can click on to open the project. To open a project go into Android Studio and open an existing project from there. If you're opening a project that you did not create, such as my samples, chose import project instead to avoid configuration errors.