Android Mobile Application Development
Week 4: Lists

Lists
https://developer.android.com/jetpack/compose/lists
Many apps need to display a collection of items, either in a list or a grid.
For a small number of items you can use the Column or Row composables to render child composables. This approach is ok for a small number of items that are static.

- Impact performance because Row and Column render all their children whether or not they're visible
- No built-in scrolling capability

Lazy Lists
If you need to display a large, or dynamic, collection of items you should use one of the Lazy components instead. Lazy composables only render items that are currently visible on the screen. As the user scrolls, items that move out of the viewable area are destroyed to free up resources while those entering the viewable area are created just in time to be displayed. This allows lists of potentially infinite length to be displayed with no performance degradation. Lazy composables also provide built-in support for scrolling.

- LazyColumn produces a vertically scrolling list
- LazyRow produces a horizontally scrolling list
- LazyVerticalGrid displays items in a vertically scrollable container, spanned across multiple columns
- LazyHorizontalGrid displays items in a horizontally scrollable container, spanned across multiple rows

The Lazy components are different to most layouts in Compose. Instead of accepting a `@Composable` content block parameter, allowing apps to directly emit composables, the Lazy components provide a `LazyListScope.()` block. This `LazyListScope` block offers a DSL (domain-specific language) which allows apps to *describe* the item contents. The Lazy component is then responsible for adding each item's content as required by the layout and scroll position.

LazyListScope DSL
The DSL of `LazyListScope` provides a number of functions for describing items in the layout. At the most basic, `item()` adds a single item, and `items(Int)` adds multiple items.

There are also a number of extension functions which allow you to add collections of items, such as a `List`.

- `items(List)` iterates through a list of items and adds them to the lazy list.
- `itemsIndexed()` adds a list of items to the lazy list while having access to the index as well

Item keys
By default, each item's state is keyed against the position of the item in the list or grid. However, this can cause issues if the data set changes, since items which change position effectively lose any remembered state. To handle this you should provide a stable and unique key for each item, providing a block to the `key` parameter. The key's type must be supported by `Bundle`, Android's mechanism for keeping the states when the Activity is recreated. `Bundle` supports types like primitives, enums or Parcelables.

Sticky Headers
Lists also let you group data and create "sticky headers" remain visible on the screen while the current group is scrolling (experimental). Once a group scrolls from view, the header for the next group takes its place.

- the list content must be stored in an Array or List which has been mapped using the Kotlin groupBy() function.

Scroll position
Lists also allow you to find, control, and react to the list's scroll position such as performing an action when a list scrolls to a specified item position.

Dialogs
https://m2.material.io/components/dialogs
Dialogs inform users about a task and can contain critical information, require decisions, or involve multiple tasks. There are different types of dialogs that can be used depending on the priority of the information and the response required from the user.

Dialog
https://developer.android.com/develop/ui/views/components/dialogs
A dialog is a small window that prompts the user to make a decision or enter additional information. A dialog does not fill the screen and is normally used for modal events. They block app usage until the user takes a dialog action or exits the dialog. They should only be used for high priority communication.
The Dialog class is the base class for dialogs, but you should avoid instantiating Dialog directly. Instead, use one of the following subclasses:

- AlertDialog
  - A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.
- DatePickerDialog or TimePickerDialog
  - A dialog with a pre-defined UI that allows the user to select a date or time.

Toast
https://developer.android.com/guide/topics/ui/notifiers/toasts
Toasts are notifications that appear on the screen in a small popup only filling the amount of space required for the message. They provide simple feedback about an operation and the current activity remains visible and interactive. Toasts automatically disappear after a timeout. They are used for low priority communication that is usually just informational.
If your app targets Android 12 (API level 31) or higher, the toast is limited to two lines of text and shows the application icon next to the text.

Use the `makeText()` method to create the Toast, which takes the following parameters:

1. The application Context.
2. The text that should appear to the user.
3. The duration that the toast should remain on the screen.

The `makeText()` method returns a properly initialized `Toast` object.

To display the toast, call the `show()` method.

There are libraries that help create more customized toasts or you can use a snackbar.

Snackbar
https://m2.material.io/components/snackbars
A snackbar provides a quick popup message to the user. The current activity remains visible and interactive while the Snackbar is displayed. They are used for low priority communication that is usually just informational.

- don't require any user action
- automatically disappear after a short time
- Informs users that some action has been performed
- Should be used for low priority messages
- Can optionally contain an action for the user to respond

Snackbars supercede Toasts and are preferred although Toasts are still supported.

Context
Android has a Context class which enables access to application-specific resources and classes, as well as application-level operations, such as launching activities, broadcasting and receiving intents, etc. Context provides the link from any component to the system, enabling access to the global application environment. In other words: the Context provides the answer to the components question of "where am I in relation to app generally and how do I access/communicate with the rest of the app?"
Within an app the context usually represents the current state of the application/object.
Context is often needed in an app when performing an operation that needs access to some system resources.
You can use `LocalContext.current` to receive the context of your Android App inside a Composable function.

Core Composables
We will also look at two other built-in composables in our example today.
https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#top-level-functions

Surface
The Surface is a built-in Composable that provides a surface container that you can use for your screen, including background color, content color, elevation, and more. Surface has parameters (all optional) for the following:

- shape
- color (for background)
- contentColor
- border
- elevation

Card
https://m2.material.io/components/cards
Cards are Surfaces that display content and actions on a single unit. The Card is a container that only supports one composable function. If you want multiple components in the same card, you need to wrap them in a Column or Row. Card has parameters (all optional) for the following:

- shape
    - RectangleShape

    - CircleShape
    - RoundedCornerShape
    - CutCornerShape
- backgroundColor
- contentColor
- border
- elevation

Book
Create a new project using the Empty Compose Activity template.
Name: Book
Package name: the fully qualified name for the project
Save location: the directory for your project (make sure there is a directory with the name of the project after the location)
Language: Kotlin
Minimum SDK: API 23: Android 6.0 Marshmallow

Now that we're past our first basic app let's set this one up with better structure and organization.

UI
Instead of having everything in MainActivity let's create a new package to hold our UI composables.

In the java folder select the book directory (not androidTest or test)
File | New | Package
Name: ui.compose

Now in the ui package along with theme we have a compose package.
Select the compose package and add a new Kotlin file called BookScreen.
File | New | Kotlin file
Name: BookScreen
Kind: File
It's common to have a Screen composable as the top level composable to draw your screen so let's set that up.
In BookScreen.kt define a composable that will be responsible for drawing the screen.
```
@Composable
fun BookScreen() {
    Text(text = "Screen")
}
```

In MainActivity.kt update onCreate() to call the function you just created.
```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
        BookTheme {
            BookScreen()
        }
    }
}
```

We don't need the Greeting() function so remove that. We also don't need a preview function so delete (or copy) that function. Now MainActivity acts as a container class and not a "god object".

In BookScreen.kt add a preview function.

```
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    BookTheme {
        BookScreen()
    }
}
```

Build and refresh to see the preview.

Data

Now let's add some data to our app. For now we'll just be creating our own data within the app.
We'll create a custom Kotlin class to represent the book data we'll be using.
First create a new package for our model. In the java folder select the book folder (not androidTest or test)
File | New | Package
Name: model

Then select the new model package and add a new Kotlin class called Book.
File | New | Kotlin class
Name: Book
Kind: Class

The Book class will have three data members – an id, book name, and author.

Remember that a Kotlin class whose main purpose is to hold data should be declared as a data class by using the keyword data.

- The Kotlin compiler will automatically override the toString(), equals(), hashCode(), and copy() methods from the Any class and provide implementations for the data class
    - Derived from the properties declared in the primary constructor
- Eliminates the need for you to manually override these methods and implement them for your data class

Data classes must meet these requirements

- The primary constructor needs to have at least one parameter
- All primary constructor parameters need to be marked as val or var
- Data classes cannot be abstract, open, sealed, or inner

Remember that in Kotlin the primary constructor is part of the class header.

```
data class Book (val id: String, val bookName: String, val author:
String){
}
```

Now create another file in the model package called SampleBookData where we'll get our data from.
File | New | Kotlin class

Name: SampleBookData
Kind: File

```
val sampleBookData = listOf(
    Book("1", "Behold the Dreamers", "Imbolo Mbue"),
    Book("2", "Horse", "Geraldine Brooks"),
    Book("3", "A Girl is a Body of Water", "Jennifer Nansubuga
Makumbi"),
    Book("4", "Once We Were Brothers","Ronald Balson"),
    Book("5", "Olga Dies Dreaming", "Xochitl Gonzalez")
)
```

Now let's use this data in BookScreen.kt
Let's create a composable that takes a Book instance as a parameter and displays the name and author in Text composables.

```
@Composable
fun BookItem(item: Book){
   Column()
       {
            Text(text = item.bookName)
            Text(text = item.author)
       }
}
```

Now update the BookScreen composable to show our book data as a list using LazyColumn. We use the items() extension function to iterate through our sampleBookData list and for each book we call BookItem passing that Book instance.

```
@Composable
fun BookScreen() {
    LazyColumn()
    {
        items(sampleBookData){book ->
            BookItem(item = book)
        }
    }
}
```

And there's our list of items. Now let's make it look a little better.

We need more definition between each book. We can add some padding to the column.

```
modifier = Modifier.padding(8.dp)
```

Using a Card will make each book stand out more. Card is a container that only supports one composable function. If you want multiple components in the same card, you need to wrap them in a column or row. So we'll put our Column in a Card. The default Card helps a bit but we can use some of its optional parameters to really make each card stand out more.

```
    Card(
        elevation = 4.dp,
        modifier = Modifier
            .padding(8.dp)
            .fillMaxWidth()
    ) {
        Column(
            modifier = Modifier.padding(16.dp)
        ) {
            Text(text = item.bookName, style =
MaterialTheme.typography.h6)
            Text(text = item.author, style =
MaterialTheme.typography.body1)
        }

    }
```

Elevation shows a shadow so the card looks elevated. We also add a modifier for padding (between the cards) and to have each card be the full width of the screen.
Shape can be used for the shape of the card with the size parameter determining the size of the shape
Border can be used for the stroke width and the color of the card border.

```
Card(
    elevation = 4.dp,
    shape = RoundedCornerShape(10.dp),
    border = BorderStroke(2.dp, color =
MaterialTheme.colors.primaryVariant),
    modifier = Modifier
        .padding(8.dp)
        .fillMaxWidth()
)
```

You could also set the backgroundColor and the contentColor for the Card which we'll do later with a custom theme.

I also used some of the predefined styles for the text of the two Text composables.

And lastly I added some padding to the LazyColumn.

```
LazyColumn(
    contentPadding = PaddingValues(
    vertical = 8.dp,
    horizontal = 8.dp)
)
```

<u>Clickable</u>
It is common for the items in a list to do something when clicked, such as performing an action or even navigate to another screen. We'll just create a simple Toast to see how we can make a list item clickable.
Add the following string resource to be used in our message.

```
<string name="readmsg">You should read</string>
```

Since we want each book item to be clickable we will set this up in our BookScreen composable.
We will declare an event handler to be called when the user clicks on a list item. This handler will be passed the text of the current item which it will use within a toast message.

Before we can create the Toast we need to get access to the Context since that's the first parameter needed to create a Toast. You can use `LocalContext.current` to receive the context of your Android App inside a Composable function. We'll get the context in our BookScreen composable since other functions might need it as well.

```
val context = LocalContext.current
```

Now let's update BookItem so it accepts context as a parameter.

```
fun BookItem(item: Book, context: Context){...}
```

We need to pass the context when we call BookItem. Update the body of LazyColumn in BookScreen.

```
LazyColumn(
    ...
)
{
    items(sampleBookData){book ->
        BookItem(item = book, context)
    }
}
```

Add the .clickable modifier to the Card component in BookItem so it accepts click events. We'll create a Toast using the context and pass it the String for the toast which includes the book name.

```
Card(
    ...
modifier = Modifier
    .padding(8.dp)
    .fillMaxWidth()
    .clickable {
        Toast
            .makeText(
                context,
                stringResource(id = R.string.readmsg) +
item.bookName,
                Toast.LENGTH_SHORT
            )
            .show()
    }
)
```

You'll notice that you get an error that says

```
@Composable invocations can only happen from the context of a
@Composable function
```

This is because `stringResource()` returns a @Composable and although BookItem is a composable function, Toast is not.

Instead of `stringResource()` we can use `context.resources.getString()`

```
        .clickable {
            Toast
                .makeText(
                        context,
                        context.resources.getString(R.string.readmsg) + "
" + item.bookName,
                        Toast.LENGTH_SHORT
                    )
                    .show()
            }
)
```

Now when you run the app you should see the list of books and when you click on one the Toast message should appear.

Theme

We'll create a custom theme by first defining new colors in Color.kt.

```
//primary
val blue = Color(0xFF81d4fa)
val blueLight = Color(0xFFb6ffff)
val blueDark = Color(0xFF0093c4)
val grey = Color(0xFFe0e0e0)

//secondary
val orange = Color(0xFFff9100)
val orangeLight = Color(0xFFffc246)
val orangeDark = Color(0xFFc56200)
```

In Theme.kt update LightColorPalette to use these new colors.

```
private val LightColorPalette = lightColors(
    primary = blue,
    primaryVariant = blueDark,
    onPrimary = Color.Black,
    surface = grey,
    secondary = orange,
    secondaryVariant = orangeDark,
    onSecondary = Color.Black
)
```

Now let's update the Card to use these colors as the background, content, and border colors.

```
Card(
    elevation = 4.dp,
    shape = RoundedCornerShape(10.dp),
    backgroundColor = MaterialTheme.colors.primary,
    contentColor = MaterialTheme.colors.onPrimary,
    border = BorderStroke(2.dp, color =
MaterialTheme.colors.primaryVariant),
...)
```

The border color had already been defined as part of the BorderStroke. The contentColor doesn't change because black is the default. But this is where you would set them with different colors.

Surface

One thing that's missing is a background color. The Surface is a built-in Composable that provides a surface container that you can use for your screen, including its background color.
In MainActivity update setContent so the call to BookScreen() is wrapped in a Surface composable. We use the fillMaxSize modifier so the surface takes up the whole screen.

Note the order here – you always want the theme called first so it's applied to the whole activity, then the surface, then your calls to composables to render the UI.

```
setContent {
    BookTheme {
        Surface(
            modifier = Modifier.fillMaxSize()
        ) {
            BookScreen()
        }
    }
}
```

Since we define a color for surface our Surface will automatically use it for the background but if we wanted to change it we would add a value for the color.

```
color = MaterialTheme.colors.background
```

<u>Launcher Icons</u>
Don't forget to create custom launcher icons. Once you create them and update the AndroidManifest file, it will be used as the icon in your Toast (Android 12/API level 31 or higher).