Android Mobile Application Development
Week 4: State Management and ViewModel

Until recently, Google did not recommend a specific approach to building Android apps. They just provided the SDK and tools and developers could decide what architecture they wanted to use for their apps. This resulted in a wide range of approaches, many resulting in "God activities" – where all the code was put into one large activity class. This resulted in large, complex classes that were hard to maintain.
In 2017 Google introduced the Android Architecture Components which became part of Android Jetpack when it was released in 2018. These components provide the building pieces needed to make it easier to create components that support single responsibility components with separation of concern. These components are designed to make your app better, more robust, and easier to maintain, but there is a cost, both in learning curve and in initial development time.
Although not forced to use them, this is clearly the path Google is taking moving forward. One of the most important rules for architecting your application is to figure out for yourself for your particular app which of these components are going to make your job easier both for initial development and long-term maintenance.

Architecture
https://developer.android.com/topic/architecture
As Android apps grow in size, it's important to define an architecture that allows the app to scale, increases the app's robustness, and makes the app easier to test. You should design your app architecture to follow a few specific principles.

Separation of concerns
Your app should not have everything thrown into one Activity class. Your app should be organized into classes that have a specific job, or concern. When concerns are well separated your code is easier to test, maintain, and upgrade

Data models
Data models should be used to represent the data of an app and drive the UI. They're independent from the UI elements and other components in your app. This means that they are not tied to the UI and or Activity classes.

Single source of truth
Data, and the methods to manipulate the data, belong to one, and only one, owner. This owner is a class that depending on the complexity of your app might be in a database, ViewModel, or even the UI.

Unidirectional data flow
In unidirectional data flow state flows down the component hierarchy and events that modify the state flow up the hierarchy.

Recommended App Architecture
https://developer.android.com/topic/architecture#recommended-app-arch (slides)
Based on these architectural principles your app should have the following layers.

UI Layer

The UI layer displays app data on the screen. Whenever the data changes, either due to user interaction (such as pressing a button) or external input (such as a network response), the UI should update to reflect the changes.
The UI layer is made up of two pieces:

- UI elements that render the data on the screen. You build these elements using composable functions.
- State holders that hold data, expose it to the UI, and handle logic. State holders can be plain objects or ViewModels depending on their complexity.

Data Layer
The data layer of an app contains the business logic. The business logic is what gives value to your app—it's made of rules that determine how your app creates, stores, and changes data. The data layer is made of a repository class for each different type of data you handle in your app.

Domain layer
For larger apps an optional domain layer can be implemented to simplify and reuse the interactions between the UI and data layers. This layer is optional because not all apps will have these requirements.

Benefits of Architecture
Having a good architecture implemented in your app brings a lot of benefits to the project and engineering teams:

- It improves the maintainability, quality and robustness of the overall app.
- It allows the app to scale. More people and more teams can contribute to the same codebase with minimal code conflicts.
- It helps with onboarding. As architecture brings consistency to your project, new members of the team can quickly get up to speed and be more efficient in less amount of time.
- It is easier to test. A good architecture encourages simpler types which are generally easier to test.
- Bugs can be investigated methodically with well-defined processes.

Investing in a well-defined architecture also has a direct impact in your users. They benefit from a more stable application, and more features due to a more productive engineering team.

ViewModel
https://developer.android.com/topic/libraries/architecture/viewmodel
ViewModel is an architecture component designed to store and manage UI state in a lifecycle conscious way. Using a ViewModel as the state holder separates ownership of the data and logic from activities. All of your business logic and any code that goes to managing data in memory should be stored by the ViewModel, therefore supporting single responsibility for our components.
ViewModel is the recommended implementation for the management of screen-level UI state with access to the data layer. Furthermore, it survives configuration changes automatically. ViewModel classes define the logic to be applied to events in the app and produce updated state as a result.
Each activity has its own ViewModel and the ViewModel that persists between configuration changes including screen rotations so you don't have to deal with life cycle events as much.

- Regardless of how many times a UI controller is recreated during the lifecycle of an app, the ViewModel instances remain in memory thereby maintaining data consistency.

- A ViewModel used by an activity will remain in memory until the activity completely finishes (activity is destroyed or in a single activity app, the app exits).
- Since the ViewModel will outlive the activity it's associated with it shouldn't contain references to any UI controllers.
    - The one exception is if the model needs access to the Application context. This is ok because an Application context is tied to the Application lifecycle which a ViewModel won't outlive. (This is different from an Activity context, which is tied to the Activity lifecycle.)
    - If you need the Application context use the AndroidViewModel class instead as it's basically a ViewModel that includes an Application reference.
- ViewModels handle transient data during the app's lifecycle, but they don't handle data persistence between app launches. We will look at data persistence approaches later in the semester.

To use the ViewModel component you will need to add the dependencies to the app's grade file:
`implementation "androidx.lifecycle:lifecycle-viewmodel-compose:2.5.1"`

To create a ViewModel class you create a class and extend the ViewModel class.
`class ItemViewModel: ViewModel() { … }`

ViewModels are recommended to be used at screen-level composables, that is, close to a root composable called from an activity. It's common to have a composable dedicated solely for this task and located at the top of the screen's composable hierarchy. You pass an instance of the ViewModel as a parameter to that composable from which the state values and functions can be accessed. Then the model state and event handler functions can then be passed to child composables as necessary. ViewModels should never be passed down to other composables, instead you should pass only the data they need and functions that perform the required logic as parameters.

`viewModel()` returns an instance of `ViewModel` by returning an existing instance or creates a new one in the given scope. The ViewModel instance is retained as long as the scope is alive. For example, if the composable is used in an activity, `viewModel()` returns the same instance until the activity is finished or the process is killed. Since the state is kept outside of the composition and stored by the ViewModel, mutations survive configuration changes.

ViewModels are not part of the composition. Therefore, you should not hold state created in composables (for example, a remembered value) in your ViewModel because this could cause memory leaks.

Book VM
As we continue building onto our Book app so we can add and delete books we'll also incorporate a ViewModel class to be our state holder.

Add the Compose view model library dependency into the app's grade file and sync.
`implementation 'androidx.lifecycle:lifecycle-viewmodel-compose:2.5.1'`

We already have our data class so now we'll create a ViewModel with the list of books and logic to manipulate the list.

<u>ViewModel</u>

Now we'll create our view model class.

Select the model package and add a new class called BookViewModel.
File | New | Kotlin class
Name: BookViewModel
Kind: Class

Our BookViewModel class subclasses the ViewModel class.
Mutable objects such as ArrayList are not observable so changes to them won't trigger recomposition when they change. Instead we need to use an observable data holder to create a mutable List that is observable. The `mutableStateListOf` function returns a new MutableList which is observable so Compose will automatically recompose when the state changes.
Our view model also includes methods to add and delete a book.

```
class BookViewModel: ViewModel() {
    var bookList = mutableStateListOf<Book>()

    fun add(newBook: Book){
        bookList.add(newBook)
    }

    fun delete(book: Book){
        bookList.remove(book)
    }
}
```

ViewModels are recommended to be used at screen-level composables, so as close to a root composable as possible. ViewModels should never be passed down to other composables, instead you should pass only the data and functions needed by other composables.

Update BookScreen with parameters that accept all the data and functions of our ViewModel that we'll want to use – the bookList, the add book and delete book functions.

```
fun BookScreen(bookList:List<Book>, addBook:(Book) -> Unit,
deleteBook:(Book) -> Unit) {...}
```

Update the Preview composable as well.

```
@Preview(showBackground = true)
@Composable
fun DefaultPreview(model: BookViewModel = viewModel()) {
    BookTheme {
        BookScreen(model.bookList, {model.add(it)},
{model.delete(it)})
    }
}
```

In MainActivity we'll add a ScreenSetup composable where we define an instance of our ViewModel in the primary constructor using `viewModel()`. Then we pass the needed viewModel data and functions to BookScreen.

```
@Composable
fun ScreenSetup(viewModel: BookViewModel = viewModel()){
    BookScreen(viewModel.bookList, {viewModel.add(it)},
{viewModel.delete(it)})
}
```

Update onCreate() to call ScreenSetup.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
        BookTheme {
            Surface(
                modifier = Modifier.fillMaxSize()
            ) {
                ScreenSetup()
            }
        }
    }
```

Add Items

A Floating Action Button (FAB) is used for the primary action on a screen so we'll use that to add a book.

Scaffold provides slots for the most common top-level Material components, such as TopAppBar, BottomAppBar, FloatingActionButton, and Drawer. By using Scaffold, it's easy to make sure these components are properly positioned and work together correctly.

In BookScreen we'll add a Scaffold with a FAB that uses the filled add icon.

Our LazyColumn will move to the content parameter of the Scaffold.

We won't be using our sample data anymore so we'll pass `bookList` instead of `sampleBookData` to the items function.

```
fun BookScreen(bookList:List<Book>, addBook:(Book) -> Unit,
deleteBook:(Book) -> Unit) {
...
Scaffold(
    floatingActionButton = {
        FloatingActionButton(
            onClick = { /*TODO*/ }
        )
        {
            Icon(Icons.Filled.Add, contentDescription = "")
        }
    },
    content = {
        LazyColumn(
```

```
                contentPadding = PaddingValues(
                    vertical = 8.dp,
                    horizontal = 8.dp
                )
            )
            {
                items(bookList) { book ->
                    BookItem(item = book, context)
                }
            }
    }
)

}
```

Dialog

When the user taps the FAB we'll use an AlertDialog for the user to enter a book name and author to add to the list.

Add these strings to your string resource file since we'll be using them in our alert dialog.

```
<string name="addBook">Add Book</string>
<string name="bookName">Name</string>
<string name="authorName">Author</string>
<string name="add">Add</string>
<string name="bookAdded">Book Added</string>
<string name="cancel">Cancel</string>
```

In Compose you use state to determine when to present the dialog.

In BookScreen add a Boolean state variable initialized with the value of false.

```
var showDialog by remember { mutableStateOf(false) }
```

When the user clicks the FAB it will be set to true and the dialog will be composed. When the user dismisses the dialog it will be set to false and the dialog will be removed from the composition.

For this to work we need a custom composable for our dialog.

We'll create a basic alert dialog to get it working before we set it up for input.

```
@Composable
fun addBookDialog(context: Context, dismissDialog:() -> Unit,
addBook: (Book) -> Unit){
    AlertDialog(
        onDismissRequest = { dismissDialog},
        title={Text(text = stringResource(id = R.string.addBook))},
        text = {Text(text=stringResource(id = R.string.bookName))},
        confirmButton = {
            Button(
             onClick = { /*TODO*/
                dismissDialog()
             }
            )
            {
                Text(text = stringResource(id = R.string.add))
```

```
            }
        },dismissButton = {
            Button(
             onClick = { /*TODO*/
                dismissDialog()
             }
            )
           {
                Text(text = stringResource(id = R.string.cancel))
            }
        }
    )
}
```

In BookScreen update the FAB onClick event handler.

```
floatingActionButton = {
    FloatingActionButton(
        onClick = {showDialog = true}
    )
```

In the content block we'll use an if statement to determine if the dialog should be shown.

```
content = {
    if (showDialog){
        addBookDialog(context, dismissDialog = {showDialog = false},
addBook)
    }
```

Now if you run it clicking the FAB should show the basic dialog and the buttons should dismiss it.
Now we'll add two text fields to our dialog which we'll put in a Column in the text parameter. They will also both need state variables to hold the value the user enters.
In the confirm button event handler we'll also create a new Book object that we'll pass to addBook() to add to our list. Each Book instance needs a unique id and we use the `UUID` class and the `randomUUID()` method which generates a random id. We use `.toString()` to convert it to the needed String type.
We also add a Toast to notify the user that the book was added (although it's not really necessary since the list will update and they'll see it.).

```
@Composable
fun addBookDialog(context: Context, dismissDialog:() -> Unit,
addBook: (Book) -> Unit){
    var bookTextField by remember {
        mutableStateOf("")
    }
    var authorTextField by remember {
        mutableStateOf("")
    }

    AlertDialog(
        onDismissRequest = { dismissDialog},
```

```
        title={Text(text = stringResource(id = R.string.addBook),
style = MaterialTheme.typography.h6)},
        text = {
            Column(modifier = Modifier.padding(top=20.dp)) {
                TextField(label = {Text(text=stringResource(id =
R.string.bookName))}, value = bookTextField, onValueChange =
{bookTextField=it})
                Spacer(modifier = Modifier.height(10.dp))
                TextField(label = {Text(text=stringResource(id =
R.string.authorName))},value = authorTextField, onValueChange =
{authorTextField=it})
            }
        },
        confirmButton = {
            Button(onClick = {
                if(bookTextField.isNotEmpty()) {
                    val newID = UUID.randomUUID().toString();
                    addBook(Book(newID, bookTextField,
authorTextField))
                    Toast.makeText(
                        context,

context.resources.getString(R.string.bookAdded),
                        Toast.LENGTH_SHORT
                    ).show()
                }
                dismissDialog()
            })
            {
                Text(text = stringResource(id = R.string.add))
            }
        },dismissButton = {
            Button(onClick = {
                dismissDialog()
            }) {
                Text(text = stringResource(id = R.string.cancel))
            }
        }
    )
}
```

The FAB should now let you add items in an alert dialog and then when the view model book list is updated it causes the BookScreen composable to recompose and the LazyColumn of books is updated. When either the confirm or dismiss button in the AlertDialog is clicked we call dismissDialog() which is a lambda expression that sets showDialog to false. This causes BookScreen to be recomposed and `if (showDialog)` now evaluates to false and the addBookDialog composable is removed from the composition.

Delete Items

We want to use the long click event on an item to delete a book. We'll use an AlertDialog to verify the delete.
Add these strings to your string resource file since we'll be using them in our alert dialog.

```
<string name="delete">Delete Book?</string>
<string name="deleteBook">Book Deleted</string>
<string name="yes">Yes</string>
<string name="no">No</string>
```

Similar to how we handled adding a book, we'll create a composable function that shows an AlertDialog to delete a book.

When the user clicks the confirm (yes) button, we'll call the delete method in our viewmodel.

```
@Composable
fun deleteBookDialog(context: Context, dismissDialog:() -> Unit,
item: Book, deleteBook: (Book) -> Unit){
    AlertDialog(
        onDismissRequest = { dismissDialog},
        title={Text(text = stringResource(id = R.string.delete),
style = MaterialTheme.typography.h6)},
        confirmButton = {
            Button(onClick = {
                deleteBook(item)
                    Toast.makeText(
                        context,

context.resources.getString(R.string.deleteBook),
                        Toast.LENGTH_SHORT
                    ).show()
                dismissDialog()
            })
            {
                Text(text = stringResource(id = R.string.yes))
            }
        },dismissButton = {
            Button(onClick = {
                dismissDialog()
            }) {
                Text(text = stringResource(id = R.string.no))
            }
        }
    )
}
```

Since deleting a book will be done on each book it makes sense to call deleteBookDialog from BookItem so BookItem will need the deleteBook function.
Update the header of BookItem.

```
fun BookItem(item: Book, context: Context, deleteBook: (Book) ->
Unit) {...}
```

And pass it in the items() body in BookScreen.

```
items(bookList) { book ->
    BookItem(item = book, context, deleteBook)
}
```

In BookItem we need a state variable to track if the delete dialog should be shown.

```
var showDeleteDialog by remember { mutableStateOf(false) }
```

In BookItem our Card already uses the .clickable modifier for the click event. To also listen for the longClick event we can use the combinedClickable modifier.

```
Card(...
    modifier = Modifier
        .padding(8.dp)
        .fillMaxWidth()
        .combinedClickable(
            onClick = {
                Toast
                    .makeText(
                        context,
                        context.resources.getString(R.string.readmsg)
+ " " + item.bookName,
                        Toast.LENGTH_SHORT
                    )
                    .show()
            },
            onLongClick = { showDeleteDialog = true }
        )
)
```

When you add .combinedClickable you will get prompted that this method is marked as `@ExperimentalFoundationApi.` Accept the prompt and it will add the following declaration at the top of your file.

```
@file:OptIn(ExperimentalFoundationApi::class)
```

Then we can use the state variable to determine if the deleteBookDialog composable should be called.

```
if (showDeleteDialog){
    deleteBookDialog(context, dismissDialog = {showDeleteDialog =
false}, item, deleteBook)
}
```

Now each book item card is listening for both the click and longClick events. When the longClick event fires showDeleteDialog is set to true and the deleteBookDialog composable is composed and the AlertDialog is presented for that book. If the user choses to delete the book the viewModel delete function is called and that book is deleted from our viewModel.
The BookScreen composable takes the bookList as a parameter so it is recomposed and the UI reflects the updated list of books.

When either the confirm or dismiss button in the AlertDialog is clicked we call dismissDialog() which is a lambda expression that sets showDeleteDialog to false. This causes BookItem to be recomposed and `if (showDeleteDialog)` now evaluates to false and the deleteBookDialog composable is removed from the composition.

Why do you think I defined showDeleteDialog in BookItem and not in BookScreen?

Deleting an item from the list changes the order of the list which could cause recomposition to recompose all of the book items because Compose doesn't have any information to uniquely identify each call to BookItem. We can add a key, using each book's unique id, to identify each BookItem composition so during recomposition Compose won't recompose the BookItem composables that haven't changed.

```
items(bookList, key={book -> book.id}) { book ->
    BookItem(item = book, context, deleteBook)
}
```

You can also implement swipe to delete in Compose if you prefer deleting items in a list that way.

In testing the app make sure you test rotation as well. You'll notice that our list remains the same, and adding and deleting books continues to work, all without specifically implementing any methods to save state. This is part of the benefit of ViewModel, it is persistent across launches of our activity.