Application Development II

420-5A6-AB

Instructor: Talib Hussain

Day 19-20:

ViewModels, Coroutines, Flows

Objectives

- ViewModel
- Coroutines
- Side-effects
- Flow
- Data Storage
- Work on Assignment #3 / Milestone 2

Parcelize

- In order to use a custom data class in rememberSaveable, it needs to be “parcelable”.
  - This is like Serializable in Java.
- To do this, you add an annotation and type to the data class declaration

```
import android.os.Parcelable
import kotlinx.parcelize.Parcelize

@Parcelize
data class myDataClass(var foo: String, var goo: Int) : Parcelable
```

- In gradle, you also need to add a plugin (in the plugin section at the top).

```
id(“kotlin-parcelize”)
```

- Don’t forget to sync gradle

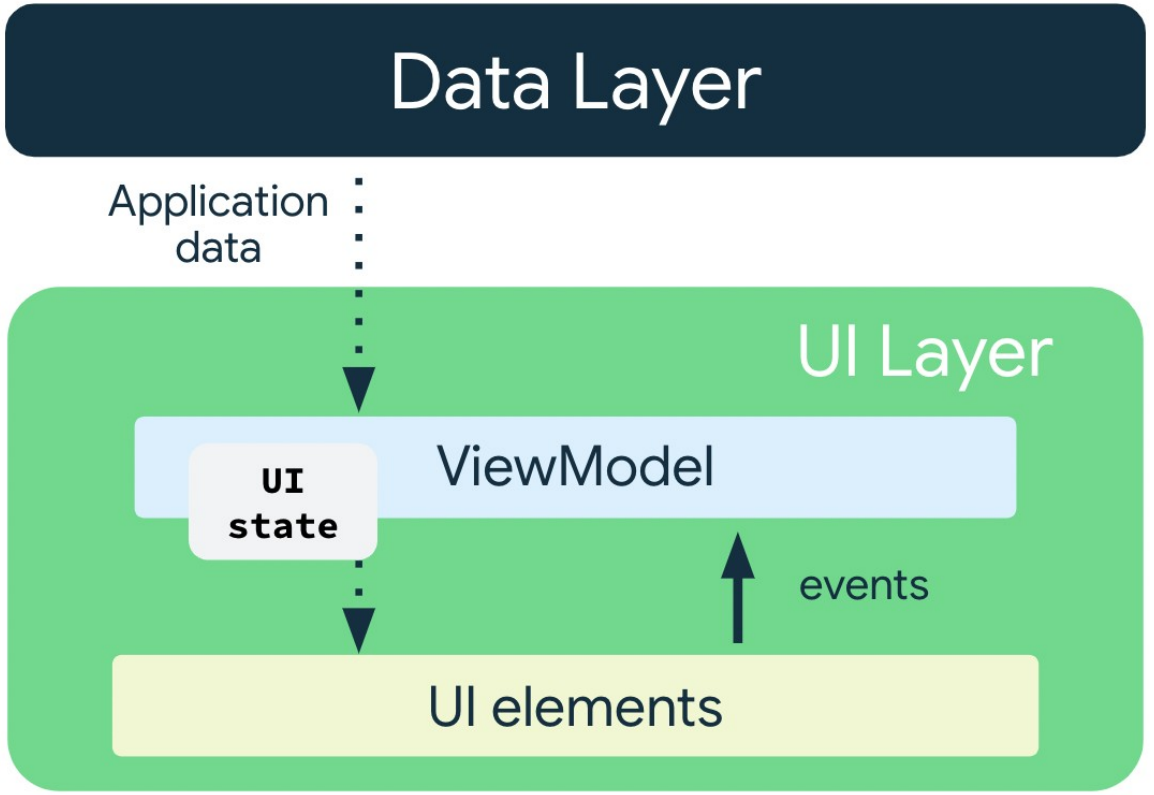
Course Schedule

- ~~Sep 7 – Assignment #1 due at midnight~~
- ~~Sep 11 (Today) – Quiz Kahoot #1 on Kotlin~~
- ~~Sep 13 (Wednesday) – Milestone #1 due before class. Presentations in class.~~
- ~~Sep 24 – Assignment #2 due midnight~~
- ~~Sep 27 – Quiz Kahoot #2 on Compose [Week 6]~~
- ~~Sep 27 – Assignment #3a WBS+LOEs due by midnight~~
- **Sep 29 – Assignment #3b Git/local setup confirmation due by midnight**
- Oct 5 – Assignment #3c due midnight
- **Oct 7 – Sprint 1: Milestone #2a due midnight (Initial project setup)**
- Oct 11 [Monday Schedule] – Quiz #3 on State/Event Handling
- Oct 16 – Quiz Kahoot #4 on Navigation/Routing
- **Oct 18 – Sprint 2: Milestone #2b due before class (Project design and Risk Management Plan). Presentations in class.**
- Oct 26 – Quiz Kahoot #5 on Coroutines/Flow/Storage [Week 10]
- Nov 1: Sprint 3 ends; In-class review with Teacher
- Nov 6 – Quiz Kahoot #6 on Authentication/TBD
- (Tues Nov 14 is Monday schedule)
- Nov 15: Sprint 4 ends; Milestone #3 due (Project design update, Preliminary code/demo)
- Nov 29: Sprint 5 ends; In-class review with Teacher
- Dec 6 [Last class]: Final project due and presentations.

Note: Assignment #4 needs to be fit in here. Likely will make it support Milestone 3.

ViewModel

- So far, we have declared state variables within composables and used remember/ rememberSaveable to persist state
- An alternative approach is to define a ViewMode to hold the state to be displayed by the UI
  - A ViewModel is a class
  - A ViewModel stores the app-related data that isn' destroyed when the activity is destroyed and recreated by the Android framework
  - The ViewModel provides the UI with access to the other layers, like the business and data layers.
  - The app automatically retains ViewModel objects during configuration changes so that the data they hold is immediately available after the recomposition.
- The state in a ViewModel will persist across configuration changes such as phone rotation



ViewModel

- ViewModels are recommended to be used at screen-level composables, that is, close to a root composable called from an activity or destination of a Navigation graph.
  - E.g., YYYViewModel to support YYYScreen

- 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.
  - E.g., if storing a list in a ViewModel, then the associated Screen should pass that list to a DisplayList composable
  - It should NOT pass the viewModel to the DisplayList composable.

- ViewModels are not part of the Composition. Therefore, you should not hold state created in composables (for example, a remembered value) because this could cause memory leaks.
  - i.e., define the state in the ViewModel, then use that state in the composable

Defining a ViewModel

- Add the following to gradle:
  - implementation("androidx.lifecycle:lifecycle-viewmodel-compose:{latest_version}")
- Create a new folder called viewmodels
- Define a new view model class. It must extend ViewModel

```
class MySimpleViewModel : ViewModel() {
}
```

- Let's create a simple counter that will be used by our view
- To do this, we want to define a private mutable variable in our view model
  - This is because we only want to be able to directly modify it within that class

```
private var _count by mutableStateOf(0)
```

- We can then define a public immutable field and override its getter. This essentially makes the value of our private state variable publicly readable.

```
val count: Int
        get() = _count
```

- Finally, let's define a function that will update our private counter.

```
fun increment() { _count +=1  }
```

```
class MySimpleViewModel : ViewModel() {
    // Declare private mutable variable that can only be modified
    // within the class it is declared.
    private var _count by mutableStateOf(0)

    // Declare a public immutable field and override its getter method.
    // Return the private property's value in the getter method.
    // When count is accessed, the get() function is called and
    // the value of _count is returned.
    val count: Int
        get() = _count

   fun increment() {
        _count +=1
    }
}
```

Using a ViewModel

- To use the viewModel in our screen, we can create it using the `viewModel()` function
- The view model is passed into our screen as a parameter
  - Or, we can create it within the constructor of our Screen. (This is somewhat easier)

```
@Composable
fun MySimpleScreen(myViewModel: MySimpleViewModel = viewModel()) {
}
```

- Within our screen, we can access the public values/functions of our view model
  - E.g., myViewModel.increment() or myViewModel.count

- Try it – Create a Screen that will display the latest count value and let the user increment the value by pressing a button
  - Once you have it working, rotate the phone – the state should persist.

```
/* Composable that gets all state information from its view model. */
@Composable
fun MySimpleScreen(myViewModel: MySimpleViewModel = viewModel()) {
    Column {
        Button(
            onClick = { myViewModel.increment() },
        ) {
            Text(text = "Increment")
        }
        Text("Total items added by user: ${myViewModel.count}")
}
```

Let's Try a List...

```
 private val _items = mutableStateListOf<String>()
 val items: List<String>
        get() = _items

fun add(String? item) {
        _items.add(item)
}

fun remove(item: String) {
        _items.remove(item)
}
```

- Note: We need to use a MutableStateList, not a MutableList.

- If our screen needs to pass details of the view model to a child composable, such as to display or change the list, it should not pass the viewModel.
  - E.g.,

```
DisplayChangingList(theList = myViewModel.items, add = myViewModel::add, remove =
myViewModel::remove)
```

  - And,in turn,

```
@Composable
fun DisplayChangingList(theList: List<String>,
                        add:(String?) -> Unit,
                        remove:(String) -> Unit) {

}
```

```
class MySimpleViewModel : ViewModel() {
    // Declare private mutable variable that can only be modified
    // within the class it is declared.
    private var _count by mutableStateOf(0)
    private val _items = initialList().toMutableStateList()

    // Declare a public immutable field and override its getter method.
    // Return the private property's value in the getter method.
    // When count is accessed, the get() function is called and
    // the value of _count is returned.
    val count: Int
        get() = _count
    val items: List<String>
        get() = _items

    fun increment() {
        _count +=1
    }
    fun add() {
        _items.add("Item # ${count+10}") // add initial size of list
        _count++
    }
    fun remove(item: String) {
        _items.remove(item)
    }

}
```

```
@Composable
fun MySimpleScreen(myViewModel: MySimpleViewModel = viewModel())
{
    Column {
        Text("Total items added by user: ${myViewModel.count}")
        DisplayChangingList(theList = myViewModel.items, add =
myViewModel::add,
            remove = myViewModel::remove)
    }
}
```

```
@Composable
fun DisplayChangingList(theList: List<String>,
                        add:() -> Unit,
                        remove:(String) -> Unit) {


    LazyColumn {
        item() {
            Button(
                onClick = {add()},
            ) {
                Text(text = "Add Item")
            }
        }
        itemsIndexed(theList) { index, item ->
            Text(
                text = "#$index: $item",
                modifier = Modifier
                    .clickable { remove(item) }
                    .padding(16.dp)
            )
        }
    }
}
```

Try It!

- Step 12 of the codelab we worked on previously introduces the use of a simple ViewModel.
  - Complete the earlier codelab (you may have stopped at step 9)
  - https://developer.android.com/codelabs/jetpack-compose-state#11

- Optional: The following codelab also introduces a ViewModel, but uses some Kotlin language features (e.g., coroutines, flows) that we haven't learned yet.
  - https://developer.android.com/codelabs/basic-android-kotlin-compose-viewmodel-and-state#3

- https://developer.android.com/courses/android-basics-compose/course
  - Unit 5

- https://www.kodeco.com/books/kotlin-coroutines-by-tutorials/v3.0/chapters/1-what-is-asynchronous-programming#toc-chapter-007-anchor-008

Kotlin Coroutine

- Coroutines offer asynchronous programming support at the language level in Kotlin.
- A *coroutine* is an instance of suspendable computation.
  - It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code.
  - However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.
  - Coroutines can be thought of as light-weight threads.
- Compose offers APIs that make using coroutines safe within the UI layer
- The rememberCoroutineScope function returns a CoroutineScope with which you can create coroutines in event handlers and call Compose suspend APIs.
- Try this:

```
fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous one is delayed
}
```

Try It!: Coroutines

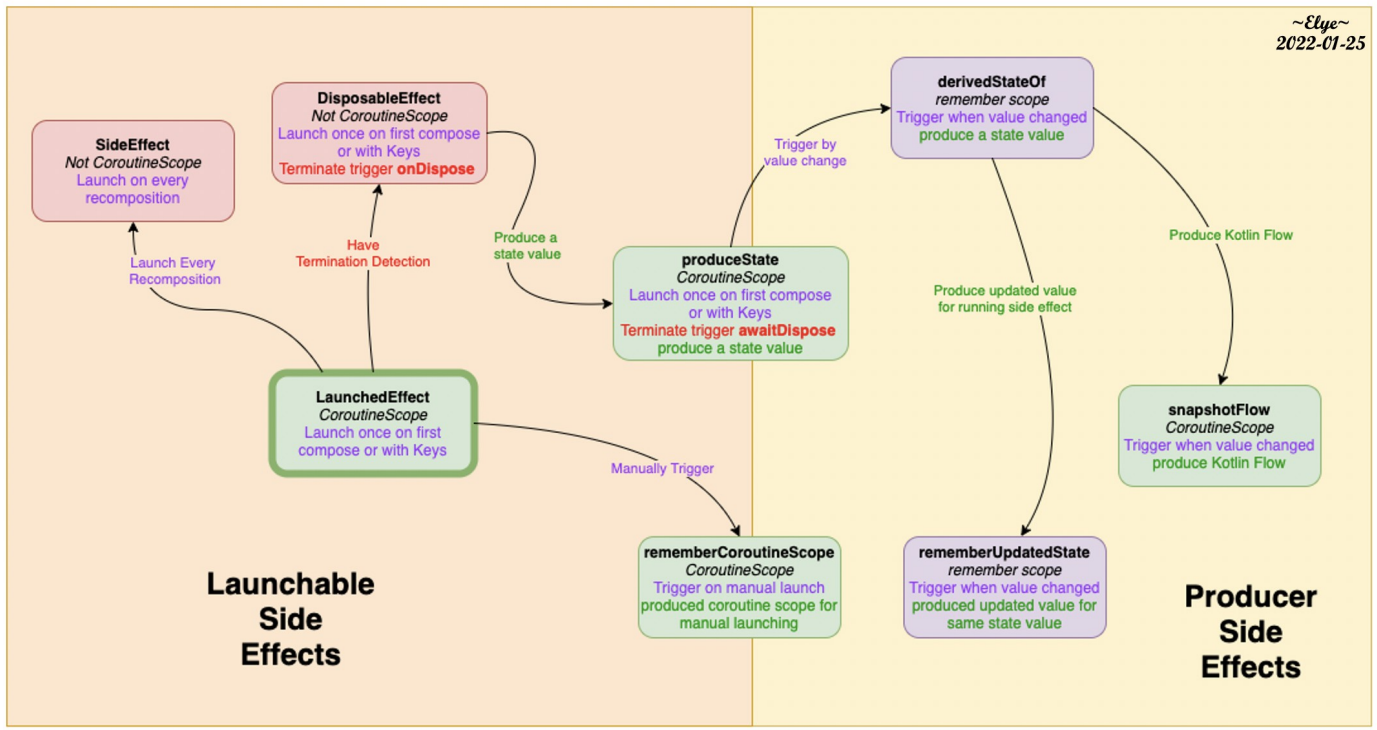
- Visit the following two web pages. Read the pages in detail and try out each code example in an IDE. Play around with it a bit to get a sense of how coroutines and flows work.
  - https://kotlinlang.org/docs/coroutines-basics.html
- Then, complete as much of these three codelabs as possible during remaining classtime.
  - https://developer.android.com/codelabs/basic-android-kotlin-compose-coroutines-kotlin-playground#3
  - https://developer.android.com/codelabs/basic-android-kotlin-compose-coroutines-android-studio
    - Don't do step 7 (unit tests)
- Other available labs/links:
  - https://developer.android.com/codelabs/kotlin-coroutines (not Compose)
  - https://www.baeldung.com/kotlin/coroutines

Side-Effects

- Side-effects: https://developer.android.com/jetpack/compose/side-effects
- https://medium.com/@mortitech/exploring-side-effects-in-compose-f2e8a8da946b
- https://proandroiddev.com/mastering-side-effects-in-jetpack-compose-b7ee46162c01

- https://www.composables.com/tutorials/side-effects

Jetpack Compose Side Effects (relating)



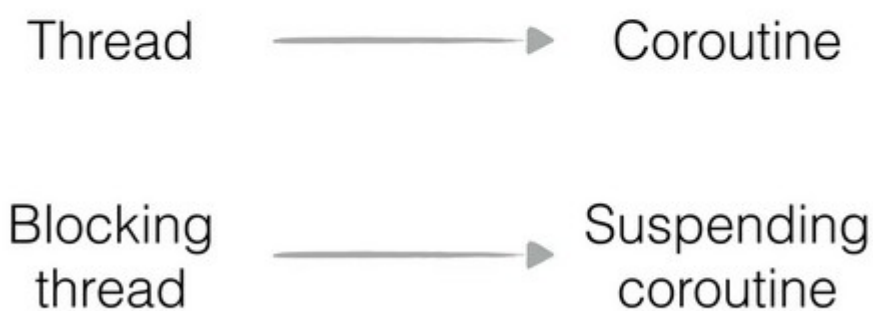
Flow

- Read:
  - https://kotlinlang.org/docs/flow.html#flows-are-cold

- Codelab:
  - https://developer.android.com/codelabs/jetpack-compose-advanced-state-side-effects

  - https://developer.android.com/codelabs/advanced-kotlin-coroutines

From threads to coroutines

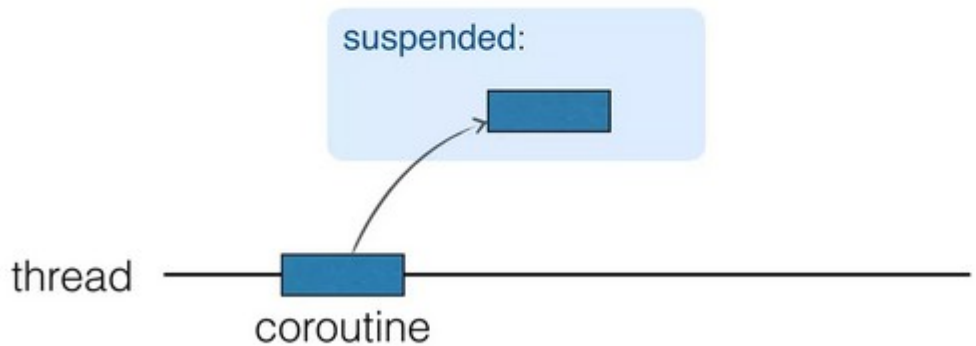


Coroutine

computation that can be suspended

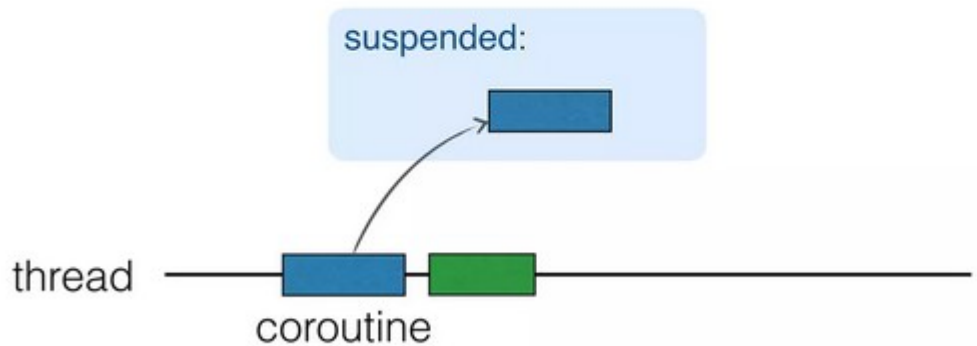
Coroutine

computation that can be suspended



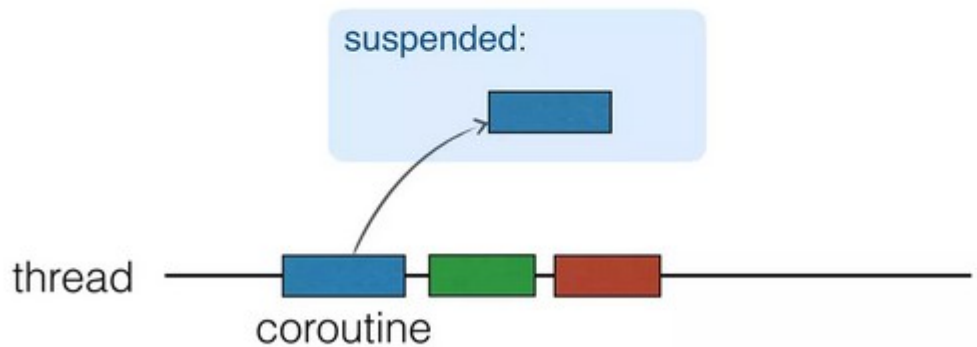
Coroutine

computation that can be suspended



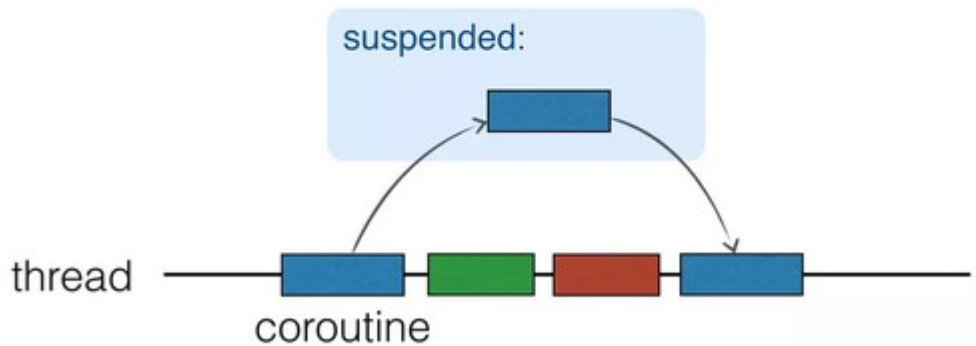
Coroutine

computation that can be suspended



Coroutine

computation that can be suspended



Thread is not blocked!

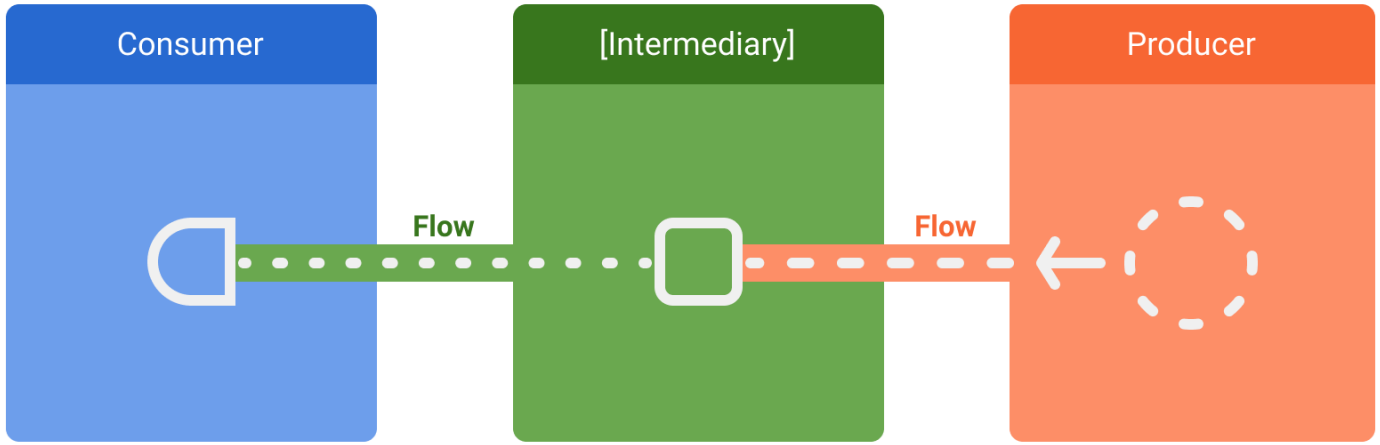
StateFlow

- StateFlow is a data holder observable flow that emits the current and new state updates.
  - Its value property reflects the current state value.
  - To update state and send it to the flow, assign a new value to the value property of the MutableStateFlow class
- In Android, StateFlow works well with classes that must maintain an observable immutable state.

Kotlin Flows

- A suspending function asynchronously returns a single value, but how can we return multiple asynchronously computed values? This is where Kotlin Flows come in.
- Using the List<Int> result type, means we can only return all the values at once.
  - To represent the stream of values that are being computed asynchronously, we can use a Flow<Int> type just like we would use a Sequence<Int> type for synchronously computed values:
- Let's review this page together:
  - https://kotlinlang.org/docs/flow.html
- Flows are cold streams similar to sequences — the code inside a flow builder does not run until the flow is collected.
- Terminal operators on flows are suspending functions that start a collection of the flow. The collect operator is the most basic one, but there are others such as toList(), toSet(), first(), reduce()
  - Calling toList() on a Flow collects all of the objects emitted by the Flow and returns them to you in a List.
- https://bladecoder.medium.com/kotlins-flow-in-viewmodels-it-s-complicated-556b472e281a
- https://developer.android.com/kotlin/flow
- https://developer.android.com/kotlin/flow/stateflow-and-sharedflow
- https://blog.canopas.com/7-useful-ways-to-create-flow-in-kotlin-577992b73315

Flow



Misc Flows

- collectAsState is an extension on StateFlow.
- Collects values from this StateFlow and represents its latest value via State.
- You need to handle the collection as per appropriate Lifecyle.

- You may see documentation online regarding LiveData. Kotlin does something different with its Flows
  - https://medium.com/androiddevelopers/migrating-from-livedata-to-kotlins-flow-379292f419fb

Flow: List vs non-List

- From ChatGPT (even more typing saved!)
- The difference between Flow<List<Frog>> and Flow<Frog> lies in the nature of the emitted values and the structure of the resulting flow.
- Flow<List<Frog>>:
  - This represents a flow that emits lists of Frog objects.
  - Each emission of the flow contains a list of Frog objects.
  - It is useful when you expect multiple Frog objects to be emitted together as a batch or collection.
  - For example, if you want to fetch a list of all frogs from a database and receive the entire list at once.
- Flow<Frog>:
  - This represents a flow that emits individual Frog objects.
  - Each emission of the flow contains a single Frog object.
  - It is useful when you want to process Frog objects one at a time or react to individual Frog objects as they are emitted.
  - For example, if you want to observe a stream of real-time updates for individual frogs in a dynamic manner.
- In summary, Flow<List<Frog>> emits lists of Frog objects, while Flow<Frog> emits individual Frog objects. The choice between the two depends on the specific use case and whether you need to work with batches of frogs or process them individually.

DataStore

- Simple walkthrough: https://medium.com/jetpack-composers/android-jetpack-datastore-5dfdfea4a3ea
- Room: https://developer.android.com/courses/pathways/android-basics-compose-unit-6-pathway-2
- https://developer.android.com/courses/pathways/android-basics-compose-unit-6-pathway-3
- https://developer.android.com/codelabs/android-preferences-datastore#7
- https://medium.com/androiddevelopers/all-about-preferences-datastore-cc7995679334
- https://developer.android.com/topic/libraries/architecture/datastore?gclid=CjwKCAiA55mPBhBOEiwANmzoQtX8aFaxx5WFTDOpYVN429tF3U8X3BnZu8ZMfJhRqGtyme_PzaypHhoCQDsQAvD_BwE&gclsrc=aw.ds#datastore-typed

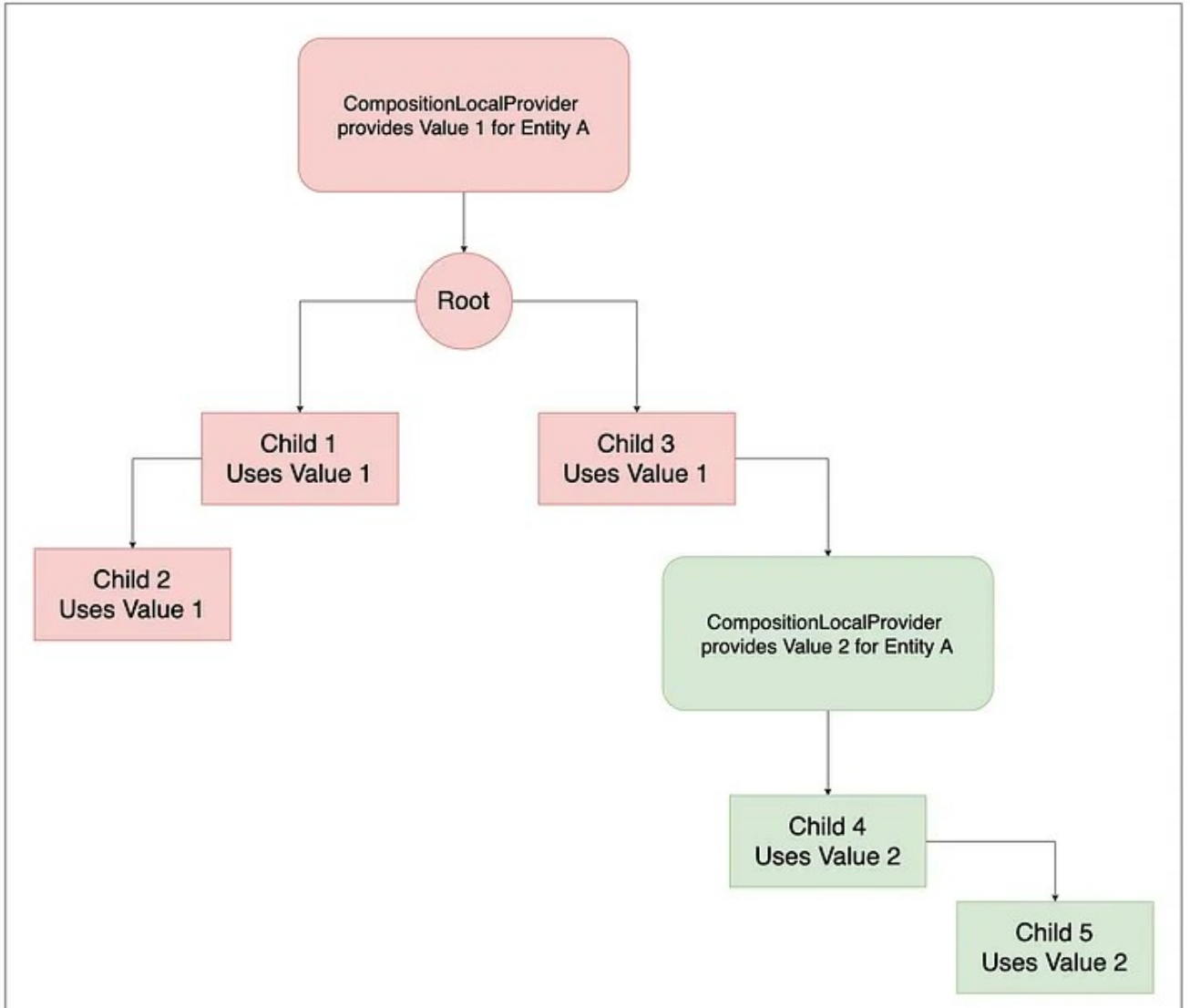
- https://android-developers.googleblog.com/2020/09/prefer-storing-data-with-jetpack.html
  - Nice walkthrough
- Firebase:
  - https://firebase.google.com/codelabs/build-android-app-with-firebase-compose#2

Error Handling

- https://levelup.gitconnected.com/error-handling-in-clean-architecture-using-flow-and-jetpack-compose-b39c729a68eb

Advanced: Provider Pattern: Interesting "Override" Behavior

- This may not be something we use in this course, but it is interesting
- In the example to the right, one CompositionLocalProvider is declared at the root level of the UI tree for A
  - Normally, the entire tree gets access to the provided instance of A.
- But, we can change this value, by wrapping a sub-tree under a new CompositionLocalProvider.
  - That way, the following sub-tree will use the latest value, and the earlier value will be overridden.
  - This only applies for the sub-tree which was wrapped – all the other nodes will keep on getting the value provided at the root level:



Misc

- FlowRow/FlowColumn: https://developer.android.com/jetpack/compose/layouts/flow
- Adaptive layouts: https://developer.android.com/jetpack/compose/layouts/adaptive