

CS 473 - MDP

Mobile Device Programming

© 2021 Maharishi International University

All course materials are copyright protected by international copyright laws and remain the property of the Maharishi International University. The materials are accessible only for the personal use of students enrolled in this course and only for the duration of the course. Any copying and distributing are not allowed and subject to legal action.



Maharishi International
University

CS 473 - MDP

Mobile Device Programming

MS.CS Program
Department of Computer Science
Renuka Mohanraj , Ph.D.



Maharishi International
University

CS 473 – MDP

Mobile Device Programming

LESSON 10

Android Jetpack Components



Maharishi International
University

Wholeness of the lesson

Jetpack is a suite of libraries to help developers follow best practices, reduce boilerplate code, and write code that works consistently across Android versions and devices so that developers can focus on the code they care about. Jetpack architectural components for dealing with data changes and app lifecycle. *Deeper understanding of any field reveals greater order; each stage of the code development process is an orderly transformation of an app. As people continue to practice Transcendental Meditation, different aspects of their lives become more orderly, and therefore more rewarding and successful.*

Introductions

- Android Jetpack, a collection of software components designed to accelerate Android development and make writing high-quality apps easier.
- Jetpack encompasses a collection of Android libraries that incorporate best practices and provide backwards compatibility in your Android apps.
- Eliminate boilerplate code
- Jetpack compose into four categories.
 - Foundation
 - Architecture
 - Behavior
 - UI
 - In this lesson we will discuss about architectural components.
- For more information refer : <https://developer.android.com/jetpack#architecture-components>

Jetpack Components Categories

Foundation	Architecture	Behaviour	UI
<ul style="list-style-type: none">• App Compat	<ul style="list-style-type: none">• Data Binding	<ul style="list-style-type: none">• Download Manager	<ul style="list-style-type: none">• Animations & Transitions
<ul style="list-style-type: none">• Android KTX	<ul style="list-style-type: none">• Life Cycles	<ul style="list-style-type: none">• Media & Playback	<ul style="list-style-type: none">• Auto
<ul style="list-style-type: none">• Multidex	<ul style="list-style-type: none">• Live Data	<ul style="list-style-type: none">• Notifications	<ul style="list-style-type: none">• Emoji
<ul style="list-style-type: none">• Test	<ul style="list-style-type: none">• Navigation	<ul style="list-style-type: none">• Permissions	<ul style="list-style-type: none">• Fragments
	<ul style="list-style-type: none">• Paging	<ul style="list-style-type: none">• Preferences	<ul style="list-style-type: none">• Layout
	<ul style="list-style-type: none">• Room	<ul style="list-style-type: none">• Sharing	<ul style="list-style-type: none">• TV
	<ul style="list-style-type: none">• View Model	<ul style="list-style-type: none">• Slices	<ul style="list-style-type: none">• Wear OS by Google
	<ul style="list-style-type: none">• Work Manager		

Architectural Components

- This lesson focus on JetPack architectural components for dealing with data changes and app lifecycle.
- Android architecture components are a collection of libraries that help you design robust, testable, and maintainable apps.
- Architecture Components could be classified as follows: Room, WorkManager, Lifecycle, Navigation, Paging, Data Binding, ViewModel, and LiveData.
- We will focus on ViewModel, LiveData, Navigation and Room.

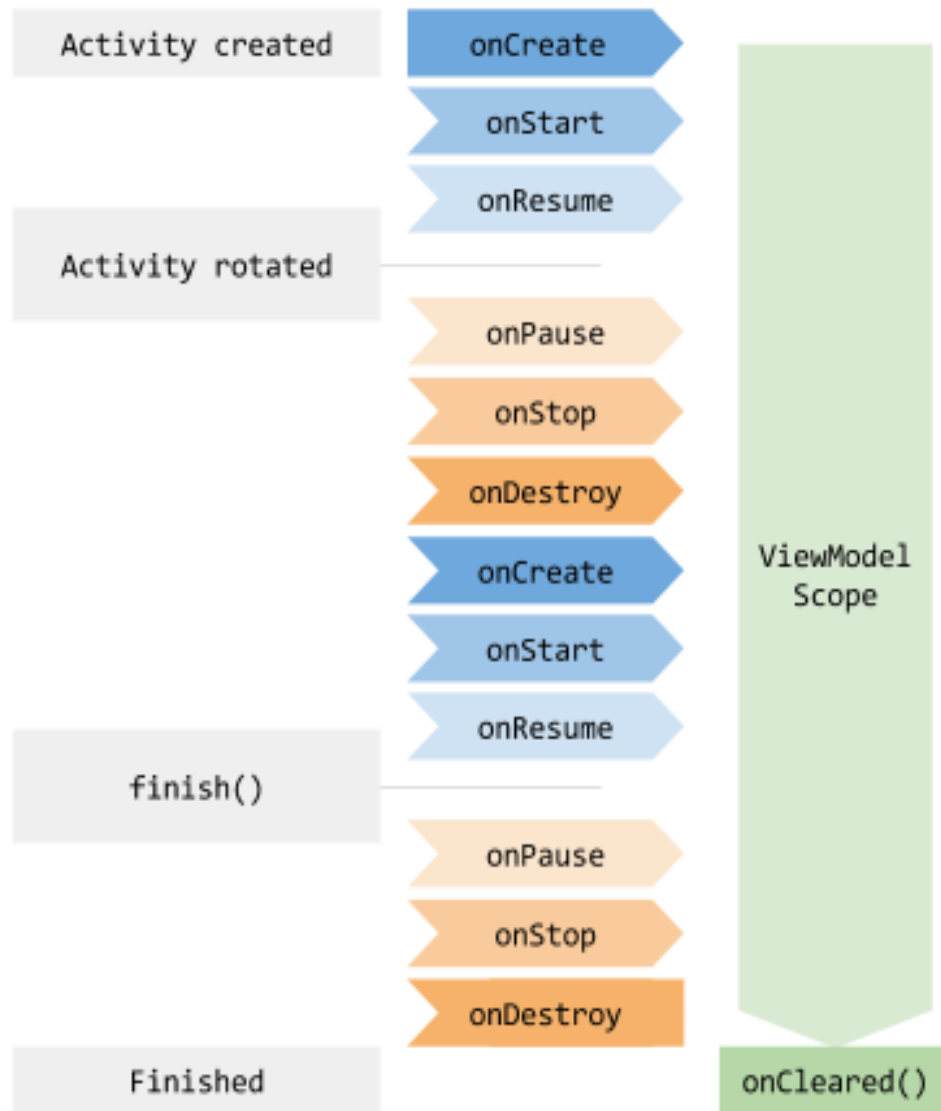
ViewModel

The ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way. The ViewModel class allows data to survive configuration changes such as screen rotations.

It's easier and more efficient to separate out view data ownership from UI controller logic.

Architecture Components provides ViewModel helper class for the UI controller that is responsible for preparing data for the UI.

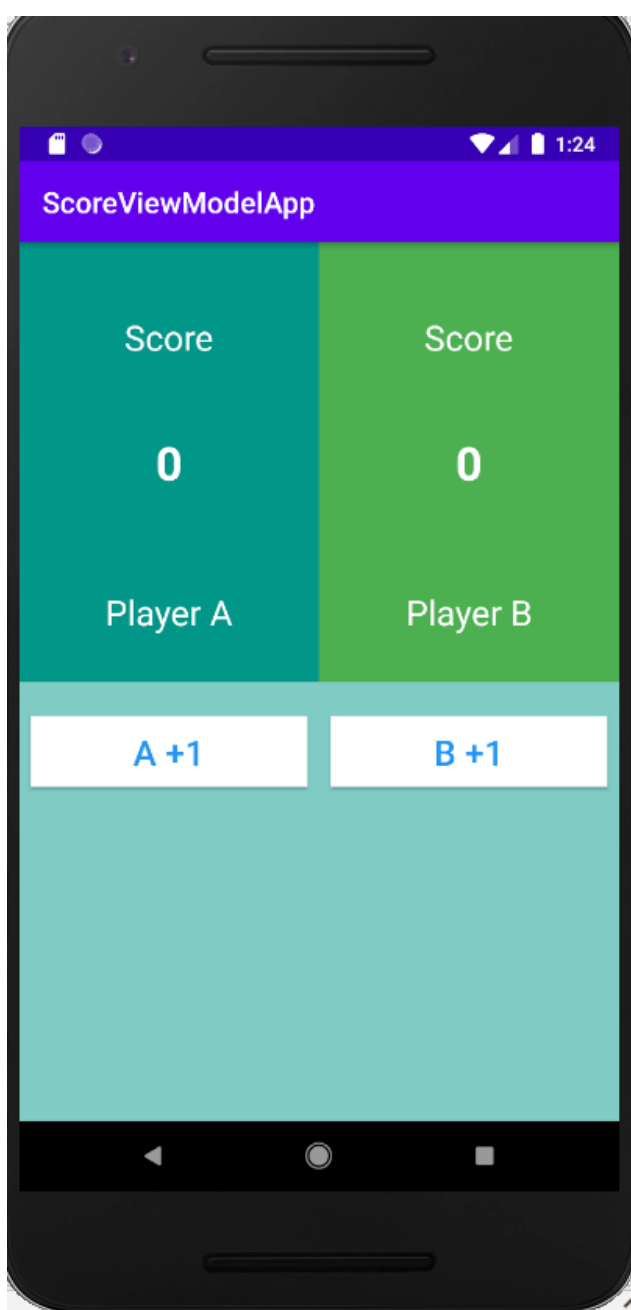
ViewModel objects are automatically retained during configuration changes so that data they hold is immediately available to the next activity or fragment instance.



Activity Life Cycle and ViewModel

ViewModel vs OnSaveInstanceState

- Both will survive configuration changes.
- ViewModel holds lot of data, but OnSaveInstanceState holds small amount of data.



Example

- Implement ViewModel in your Application to handle configuration changes like Device Rotation to maintain the States.
- If you click the A +1 and B +1 buttons increase the score per click to the respective player.
- Refer: ViewModelScore

ViewModel Implementation Steps

Step 1: Make a ViewModel class inherits from ViewModel

```
import androidx.lifecycle.ViewModel  
  
class MainViewModel : ViewModel()
```

Step 2: Put your UI related data need to maintain the configuration changes inside your ViewModel class.

Step 3: Inside your MainActivity onCreate() method, create an object for the ViewModel class using ViewModelProviders.

```
ViewModelProvider(this).get(MainViewModel::class.java )
```

ViewModel class does keep track of the associations between ViewModel and UI controller instance behind the scenes, using the UI controller you pass in as the first argument. Over here (this) is the UI Controller.

MainViewModel.kt

```
import androidx.lifecycle.ViewModel

class MainActivityViewModel: ViewModel() {

    var countA = 0

    var countB = 0

    fun updateCountA(){
        ++countA
    }

    fun updateCountB(){
        ++countB
    }

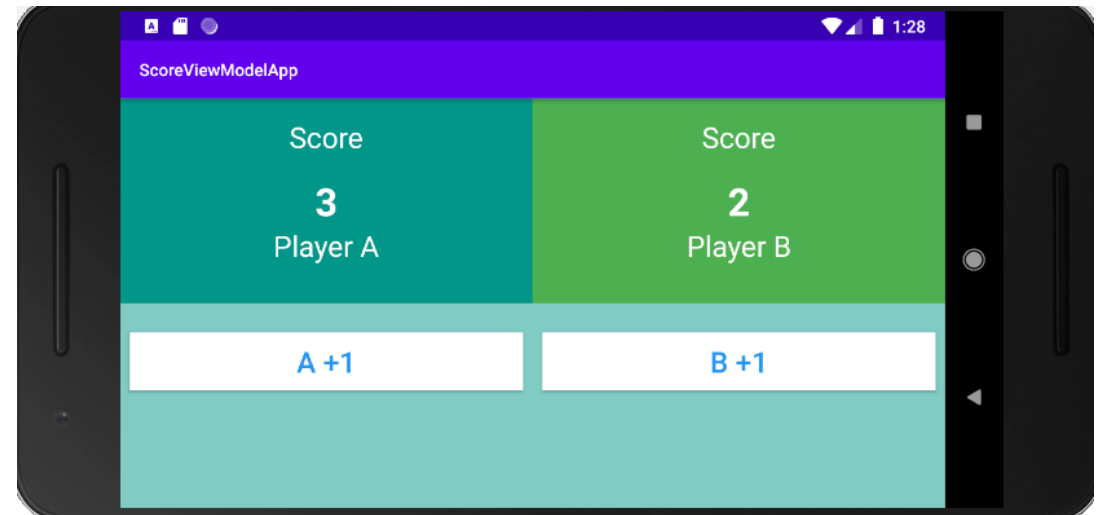
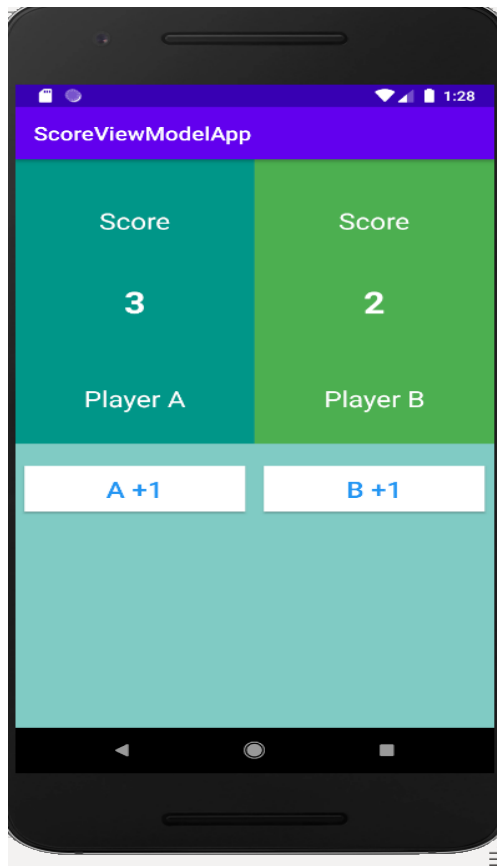
}
```

```
class MainActivity : AppCompatActivity() {  
    private lateinit var binding: ActivityMainBinding  
    private lateinit var mainVM: MainViewModel  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        binding = ActivityMainBinding.inflate(layoutInflater)  
        setContentView(binding.root)  
        // Initialize the ViewModel to retrieve the data from the ViewModel  
        mainVM = ViewModelProvider(this).get(MainViewModel::class.java)  
        // Initially load zero to score field  
        binding.tvScorePlayerA.text = mainVM.countA.toString()  
        binding.tvScorePlayerB.text = mainVM.countB.toString()  
        // Click Listener to increment the playerA Score by 1 for each click  
        // and updated on the ScoreTextView  
        binding.btnPlayerA.setOnClickListener{  
            mainVM.updateCountA()  
            binding.tvScorePlayerA.text = mainVM.countA.toString()  
        }  
        // Click Listener to increment the playerB Score by 1 for each click and updated on the ScoreTextView  
        binding.btnPlayerB.setOnClickListener{  
            mainVM.updateCountB()  
            binding.tvScorePlayerB.text = mainVM.countB.toString()  
        }  
    }  
}
```

MainActivity.kt

Sample Output

The scores are maintained with the help of ViewModel due to configuration changes(Screen Rotation)



LiveData

- LiveData is an observable data holder class.
- LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.
- LiveData considers an observer, which is represented by the Observer class, to be in an active state if its lifecycle is in the STARTED or RESUMED state.
- Refer : <https://developer.android.com/topic/libraries/architecture/livedata>

Using LiveData Advantages

Ensures your UI matches your data state

No memory leaks

No crashes due to stopped activities

No more manual lifecycle handling

Always up to date data

Proper configuration changes

Sharing resources

Work with LiveData objects

- Create an instance of LiveData to hold a certain type of data. This is usually done within your ViewModel class.

```
private var countALiveData= MutableLiveData<Int>()  
private var countBLiveData= MutableLiveData<Int>()
```
- Create an Observer object that defines the onChanged() method, which controls what happens when the LiveData object's held data changes.
- Attach the Observer object to the LiveData object using the observe() method. The observe() method takes a LifecycleOwner object. This subscribes the Observer object to the LiveData object so that it is notified of changes.
- You usually attach the Observer object in a UI controller, such as an activity or fragment.
- Reference: LiveDataDemo

MainActivityViewModel.kt

```
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel

class MainActivityViewModel: ViewModel() {
    var countA = MutableLiveData<Int>()
    var countB = MutableLiveData<Int>()
    init {
        countA.value = 0
        countB.value = 0
    }
    fun updateCountA(){
        countA.value = (countA.value)?.plus(1)
    }
    fun updateCountB(){
        countB.value = (countB.value)?.plus(1)
    }
}
```

MainActivity.kt

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    private lateinit var mainViewModel: MainViewModel
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        // Initialize the ViewModel to retrieve the data from the ViewModel
        mainViewModel = ViewModelProvider(this).get(MainViewModel::class.java)
        // countA LiveData observer implementation
        mainViewModel.countA.observe(this) {
            binding.tvScorePlayerA.text = it.toString()
        }
        mainViewModel.countB.observe(this) {
            binding.tvScorePlayerB.text = it.toString()
        }
        binding.btnPlayerA.setOnClickListener{
            mainViewModel!!.updateCountA()
        }
        // Click Listener to increment the playerB Score by 1 for each click and updated on the ScoreTextView
        binding.btnPlayerB.setOnClickListener{
            mainViewModel!!.updateCountB()
        }
    }
}
```

Main Point 1

The purpose of the ViewModel is to acquire and keep the information that is necessary for an Activity or a Fragment. The Activity or the Fragment should be able to observe changes in the ViewModel. ViewModels usually expose this information via LiveData or Android Data Binding. ViewModel manage UI-related data in a lifecycle conscious way. *Science of Consciousness: Group TM practice, for people to change their thinking and do only right things, observes profound changes happen within them, brought about by a more profound effect than resting at the end of the day. For real transformation of society, people have to be in an atmosphere that is deeply harmonious and good; then people change from within.*

Navigation Component

- A Collection of libraries, a plugin and tooling that simplifying Android Navigation.
- Android Jetpack's Navigation component helps us to implement navigation, from simple button clicks to more complex patterns, such as app bars and the navigation drawer.
- Benefits
 - Simplified setup for common navigation patterns (like Bottom Navigation)
 - Handles up and back actions correctly by default
 - Automates fragment transactions
 - Type safe argument passing
 - Handles transition animations
 - Simplified deep linking(open specific activity in your app, not covered here)
 - Centralizes and visualizes the navigation (ie Navigation graph)
 - Built in support with Activities and Fragments

Navigation Component

The Navigation component consists of three key parts that are described below:

1. **Navigation Graph (New XML resource)** - This is a resource that contains all navigation-related information in one centralized location. This includes all the places in your app, known as destinations. A destination is any place you can navigate to in your app, usually a fragment or an activity.
2. **NavHostFragment (Layout XML view)** - This is a special widget you add to your layout. It displays different destinations from your Navigation Graph.
3. **NavController (Kotlin/Java object)** - This is an object that keeps track of the current position within the navigation graph. It composes swapping destination content in the NavHostFragment as you move through a navigation graph.

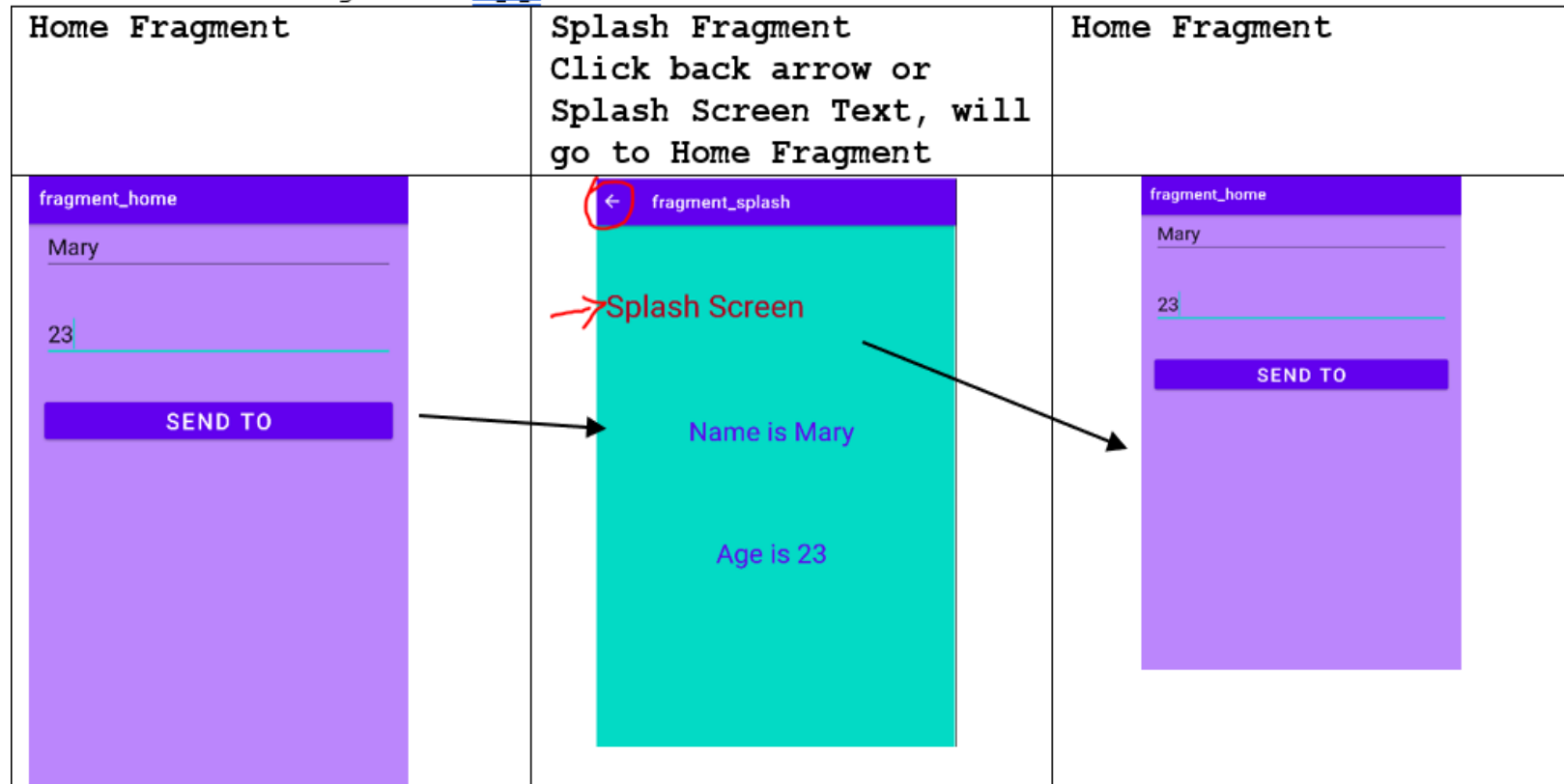
As you navigate through your app, you tell the NavController that you want to navigate either along a specific path in your navigation graph or directly to a specific destination. The NavController then shows the appropriate destination in the NavHost.

Navigation Guidelines

- Starting destination – visible when application is started from a launcher.
- Navigation state is represented by a stack (start destination at the bottom, current destination at the top of the stack)
- Up button takes user to hierarchical parent destination, never exits the app
- Pressing Back button terminate your application when you in the starting destination.
- Reference: <https://developer.android.com/guide/navigation/navigation-getting-started>

Hands on Example

Problem Requirement : Need to create two Fragments and by passing data from Fragment1 to Fragment2 and navigate between both fragments using JetPack Navigation Component. Use Safeargs feature from the Navigation Component. Finally add the NavigateUp feature on the action bar.



Gradle setup for Navigation Component

Add the below two dependencies to use Navigation Component

implementation 'androidx.navigation:navigation-fragment-ktx:2.5.3'

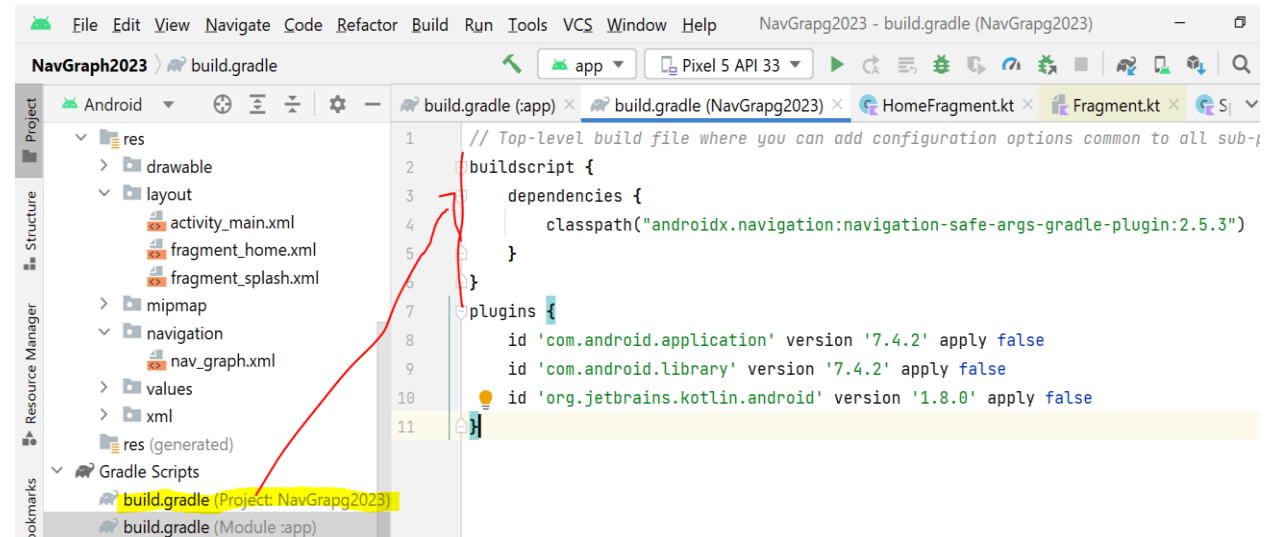
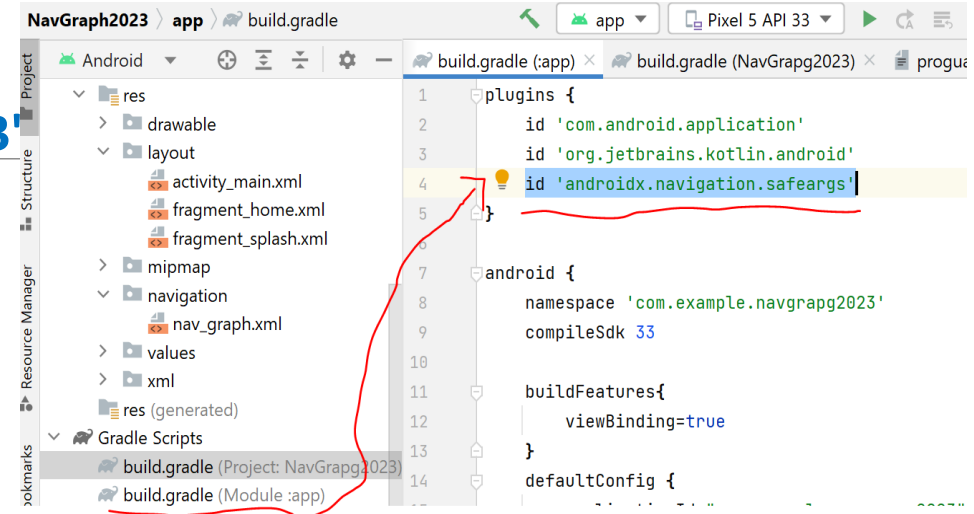
implementation 'androidx.navigation:navigation-ui-ktx:2.5.3'

Add below plugins on the top

id 'androidx.navigation.safeargs'

Inside build.gradle(Project), add the below lines on the top

```
buildscript {  
    dependencies {  
        classpath("androidx.navigation:navigation  
-safe-args-gradle-plugin:2.5.3")  
    }  
}
```



Visual representation of Navigation graph in the design mode

The screenshot displays the Android Studio interface with the navigation graph in design mode. The left sidebar shows the project structure, with the `nav_graph.xml` file highlighted under the `navigation` folder. The main editor area shows the `Destinations` and `GRAPH` tabs. The `Destinations` tab lists `HOST` and `activity_main (fragment)`, with an arrow pointing to the text `activity_main.xml having Nav_Host Fragment`. The `GRAPH` tab lists `homeFragment - Start` and `splash1Fragment`, with an arrow pointing to the text `Two Fragments are added, homeFragment is the Starting Fragment`. On the right, a visual representation of the navigation graph shows two fragments: `homeFragment` (labeled `Home Screen`) and `splash1Fragment` (labeled `Splash Screen`). Arrows indicate the flow of navigation: from `homeFragment` to `splash1Fragment`, and from `splash1Fragment` back to `homeFragment`.

Android

app

- manifests
- java
 - com.miu.navigationcomponent
 - HomeFragment
 - MainActivity
 - Splash1Fragment
 - com.miu.navigationcomponent (androidT)
 - com.miu.navigationcomponent (test)
- java (generated)
- res
 - drawable
 - layout
 - mipmap
 - navigation
 - nav_graph.xml
 - values
- Gradle Scripts
 - build.gradle (Project: NavigationComponent)

activity_main.xml x MainActivity.kt x nav_graph.xml x Splash1Fragment.kt x HomeFragment.kt

Destinations

HOST

activity_main (fragment)

activity_main.xml having Nav_Host Fragment

GRAPH

homeFragment - Start

splash1Fragment

Two Fragments are added, homeFragment is the Starting Fragment


homeFragment

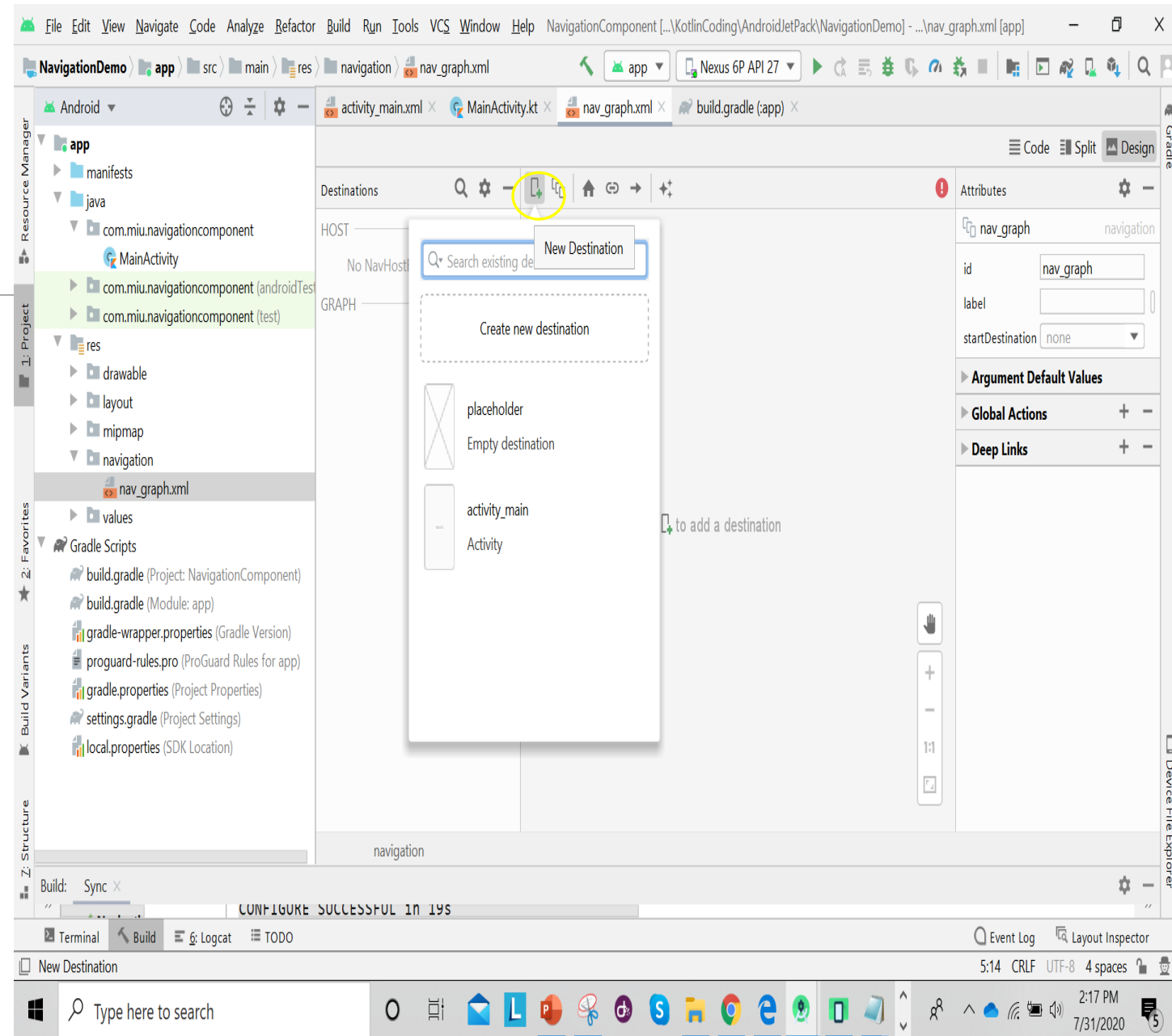
splash1Fragment

Home Screen

Splash Screen

Destinations

- Navigation Graph includes destinations as Activities or Fragments.
- Destinations can be created in the nav_graph design editor or from an existing activities or fragments from the project.
- Click on the small icon  to add the Fragments, here called as New Destination.




XML Representation – Fragment included

```
<?xml version="1.0" encoding="utf-8"?>
<navigation
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/nav_graph"
  app:startDestination="@id/homeFragment">
  <fragment
    android:id="@+id/homeFragment"

    android:name="com.miu.navigationcomponent.HomeFragment"
    android:label="fragment_home"
    tools:layout="@layout/fragment_home" > </fragment>
</navigation>
```

app:startDestination="@id/homeFragment"

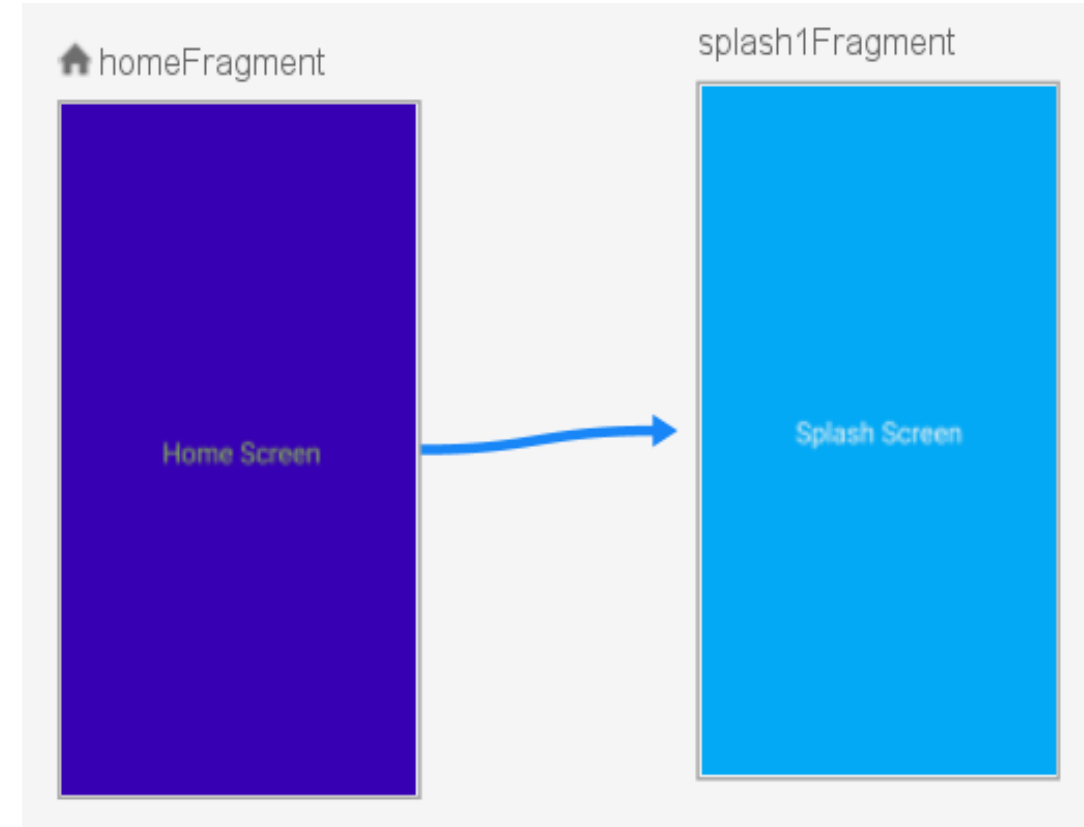
The above xml code says homeFragment is the first screen user can see, once application launched.

In Graph Editor it is shown with icon and the fragment name. Example :  homeFragment

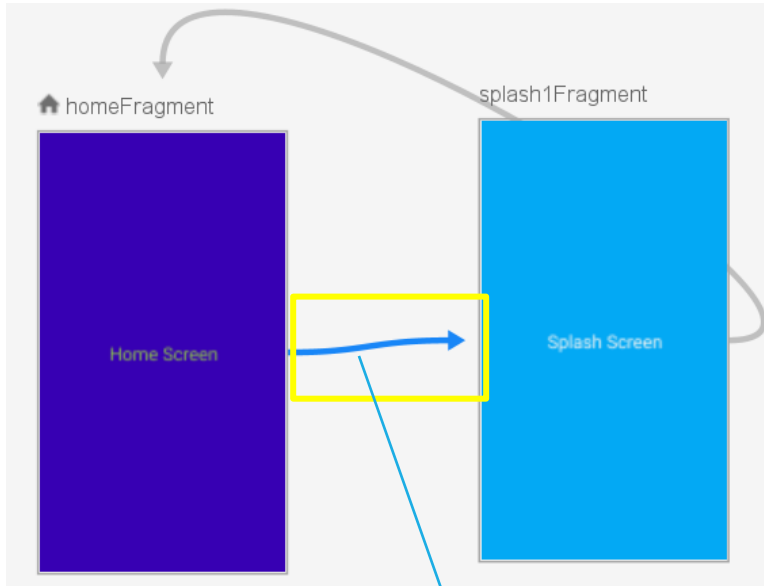
Connections

Destinations are connected using actions. Lines shown in the navigation graph are visual representations of actions.

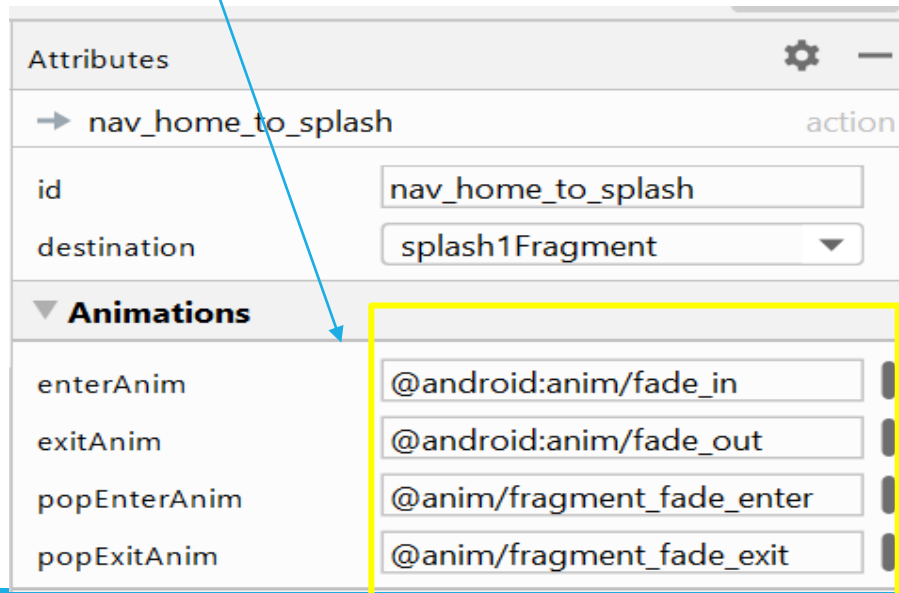
```
<fragment
    android:id="@+id/homeFragment"
    android:name="com.miu.navigationcomponent.HomeFragment"
    android:label="fragment_home"
    tools:layout="@layout/fragment_home" >
    <action
        android:id="@+id/nav_home_to_splash"
        app:destination="@id/splashFragment"/>
</fragment>
```



Transition between destinations for the highlighted action and the XML code



```
<fragment
    android:id="@+id/homeFragment"
    android:name="com.miu.navigationcomponent.HomeFragment"
    android:label="fragment_home"
    tools:layout="@layout/fragment_home" >
    <action
        android:id="@+id/nav_home_to_splash"
        app:destination="@id/splashFragment"
        app:enterAnim="@android:anim/fade_in"
        app:exitAnim="@android:anim/fade_out"
        app:popEnterAnim="@anim/fragment_fade_enter"
        app:popExitAnim="@anim/fragment_fade_exit" />
    </fragment>
```



Passing data between destinations

Every destination can define what arguments it can receive.

Two possible ways of passing data between destinations:

- using Bundle
- type-safe way using safeargs Gradle plugin

It can be done easily in the Destination's Attributes panel in the Arguments section by clicking + icon and need to provide name, type and default value(optional).

Ex : rannumber: integer (100)

Attributes

☐ splash1Fragment fragment

id

label

name

▼ Arguments + -

a rannumber: integer (100)

▼ Actions + -

→ homeFragment (nav_splash_to_home)

Add Argument

Name

Type

Array ☐

Nullable ☐

Default Value

Add Cancel

Passing data between destinations- xml code

```
<fragment
    android:id="@+id/splash1Fragment"
    android:name="com.miu.navigationcomponent.Splash1Fragment"
    android:label="fragment_splash1"
    tools:layout="@layout/fragment_splash1" >
    <action
        android:id="@+id/nav_splash_to_home"
        app:destination="@id/homeFragment" />
    <argument
        android:name="rannumber"
        app:argType="integer"
        android:defaultValue="100" />
</fragment>
```

Passing data in a Type safe way

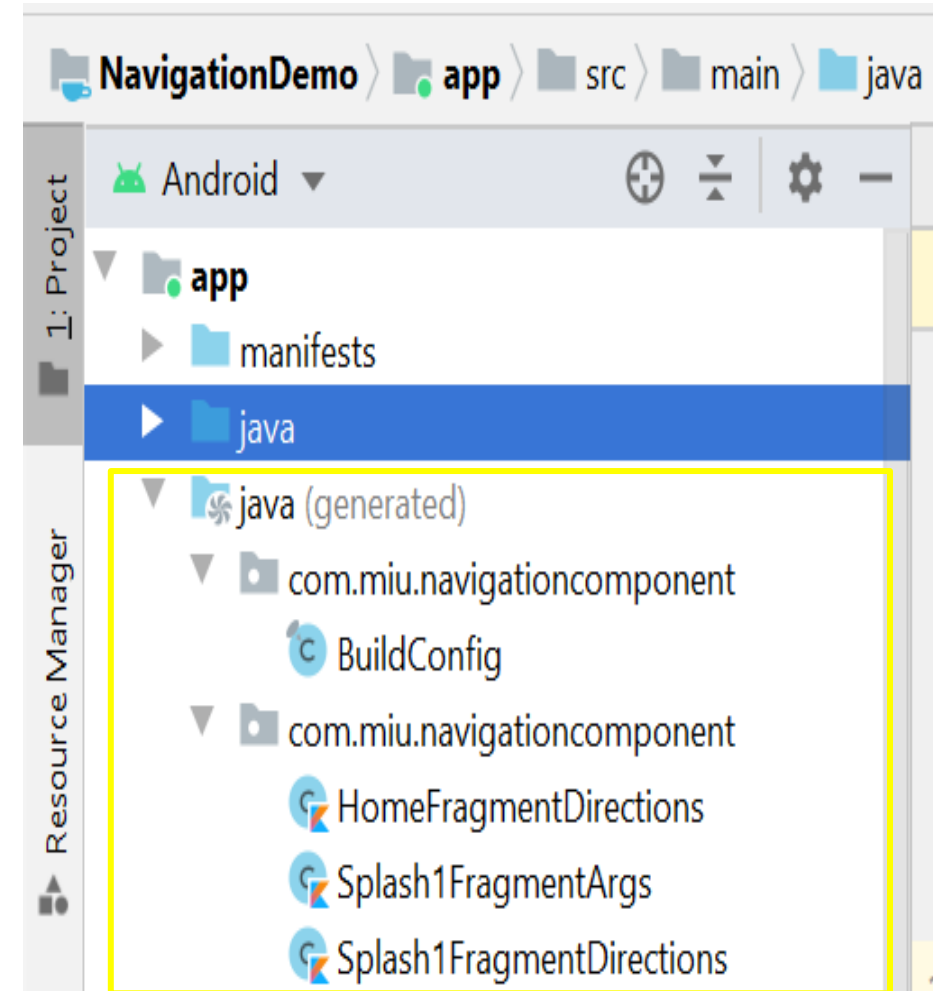
Gradle plugin safe args

id : 'androidx.navigation.safeargs'

Note: Latest version of Android studio will not show the Generated classes.

It generates object and builder classes for type-safe access arguments under java(gererated):

- A class for the destination where the action originates in the format of ClassNameDirections
- A class for the action used to pass the arguments
- A class for the destination where the arguments are passed in the format of DestinationClassNameArgs



```

<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="h
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <fragment
        android:id="@+id/fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:navGraph="@navigation/nav_graph" />
    </androidx.constraintlayout.widget.ConstraintLayout>

```

NavHostFragment

- NavHostFragment swaps in and out different fragment destinations as user navigates through the navigation graph.
- The `app:defaultNavHost="true"` is simply stating that you want this to be the NavHost that intercepts and works as the back button on your device.
- The `app:navGraph="@navigation/nav_graph"` states that Navigation graph is associated by navGraph .

Navigate to a destination

- Navigating to a destination is done using a NavController, an object that manages app navigation within a NavHost. navigate() method is used to navigate to a destination. It accepts ID of a destination or of an action.
- To pass the data, you can use the system generated directions classes with action. A class is created for each destination where an action originates. The name of this class is the name of the originating destination, appended with the word "Directions".
- Eg : If you have HomeFragment with an action, the generated class name is HomeFragmentDirections. This class has a method for each action defined in the originating destination.
- Code to pass the argument from HomeFragment to SplashFragment.

```
btnSend.setOnClickListener {
```

```
    // Navigate through Generated Directions through the action Home Fragment to SplashFragment
```

```
    val directions =HomeFragmentDirections.actionHomeFragmentToSplashFragment(
```

```
        etName.text.toString(),etAge.text.toString().toInt())
```

```
    // Calling this on a Fragment to navigate your Directions
```

```
    findNavController().navigate(directions)
```

```
}
```

Sending and Receiving the Safeargs

- A class is created for the receiving destination. The name of this class is the name of the destination, appended with the word "Args".
- Example : If your receiving class name is SplashFragment, then the auto generated class is SplashFragmentArgs.
- Need to declare an object to receive the Navigation arguments from the Generated Args class.

```
private val nargs : SplashFragmentArgs by navArgs()  
binding.tvName.text = "Name is ${nargs.pname}"  
binding.tvAge.text = "Age is ${nargs.page.toString()}"
```

// Calling this on a Fragment to navigate Directions id

```
findNavController().navigate(R.id.action_splashFragment_to_homeFragment)
```

Add NavigateUp feature

```
class MainActivity : AppCompatActivity() {  
    // Declare Navigation Controller Object to manage navigation within NavHost  
    private lateinit var mNavController: NavController  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        // Need this support to interact the fragments associated with the Activity  
        val navHostFragment = supportFragmentManager.findFragmentById(R.id.fragmentContainerView) as NavHostFragment  
        mNavController = navHostFragment.navController  
        // Code to link the navigation controller to the app bar  
        NavigationUI.setupActionBarWithNavController(this, mNavController)  
    }  
    // override the onSupportNavigateUp() method to call navigateUp() in the navigation controller  
    override fun onSupportNavigateUp(): Boolean {  
        return mNavController.navigateUp()  
    }  
}
```

Reference

Refer the complete step by step implementation for the Navigation Component hands on example from

[Navigation Component Demo Step by Step Implementation.pdf](#)

Main Point 2

Android Jetpack's Navigation component helps you implement navigation, from simple button clicks to more complex patterns, such as app bars and the navigation drawer. The Navigation component also ensures a consistent and predictable user experience by adhering to an established set of principles for the greater transformation between activities or fragments. **Science of Consciousness:** *Deeper understanding of any field reveals greater order; each stage of the navigation process is an orderly transformation of inner self. As people continue to practice Transcendental Meditation, different aspects of their lives become more orderly, and therefore more rewarding and successful.*

Room Database - Agenda

- Various Dependencies
- How to work with Room DB
- Room DB Annotations
- Used Navigation Component
- Room DB functionalities cannot run on Main Thread
 - To Manage, separate thread using Kotlin Coroutines
- Suspend function
- Kotlin higher order scope functions (let)
- Kotlin Extension function

Required dependencies on build.gradle

Dependencies need to include build.gradle

It should be at the top of build.gradle

id '**androidx.navigation.safearg**'

id '**kotlin-kapt**' → Kotlin Annotation Processing Tool

dependencies{ *// version of the room* Link :

<https://developer.android.com/topic/libraries/architecture/room>

val room_version = "2.5.1" [Kverson may change

// Room Library

implementation "**androidx.room:room-runtime:\$room_version**"

kapt "**androidx.room:room-compiler:\$room_version**"

Required dependencies on build.gradle

Navigation Library & Safe Args

Version of Navigation- [//https://developer.android.com/guide/navigation/navigation-getting-started](https://developer.android.com/guide/navigation/navigation-getting-started)

```
val nav_version = " 2.5.2 "
```

```
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
```

```
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

Refer : Slide 26 – Navigation Component Dependency

```
// Kotlin Coroutines
```

```
implementation "androidx.room:room-ktx:$room_version"
```

```
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.1")
```

Room Database

Included with the Android Architecture Components and the part of JetPack, the Room persistence library is designed specifically to make it easier to add database storage support to Android apps in a way that is consistent with the Android architecture guidelines.

It provides an abstraction layer over SQLite(before Room DB, android used it) to allow for more robust database access while harnessing the full power of SQLite.

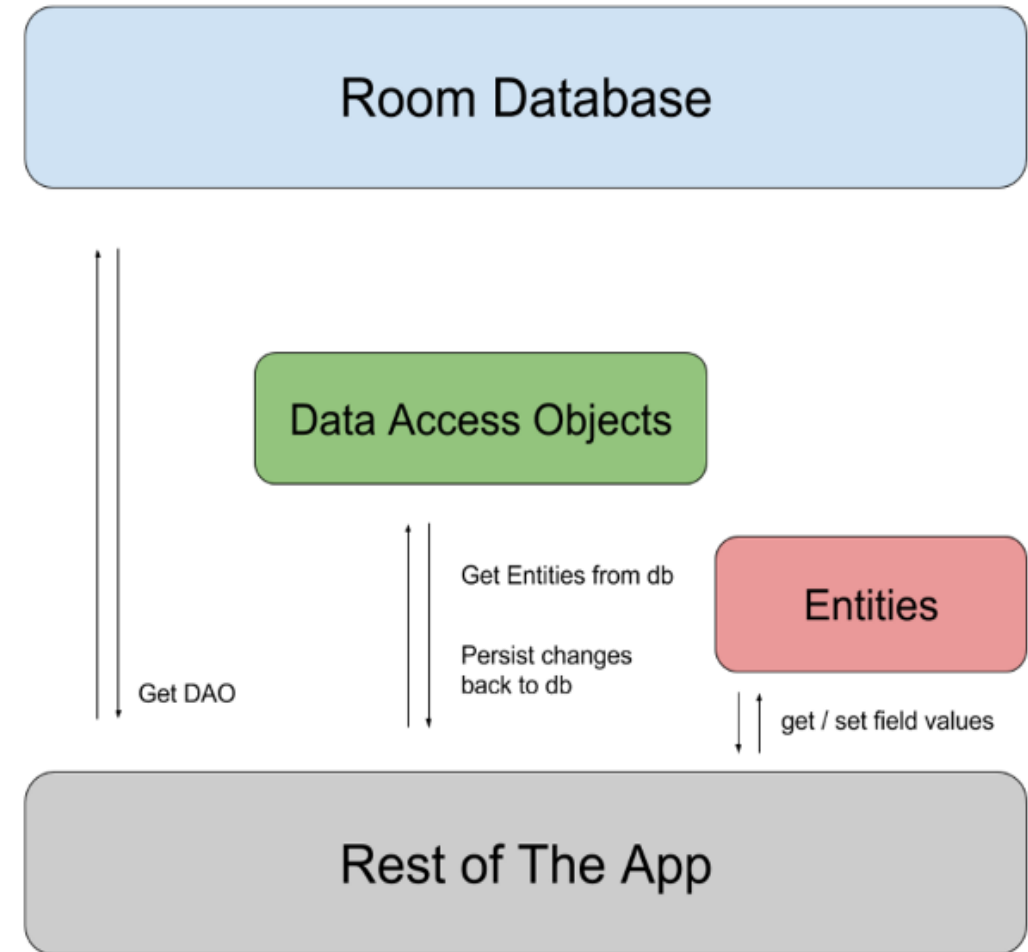
Advantages of Room over SQLite

- In case of SQLite, There is no compile time verification of raw SQLite queries. But in Room there is SQL validation at compile time.
- As your schema changes, you need to update the affected SQL queries manually. Room solves this problem by doing Migration.
- You need to use lots of boilerplate code to convert between SQL queries and Java data objects. Room maps our database objects to Java/Kotlin Object without boilerplate code.
- Room is built to work with LiveData for data observation, while SQLite does not.

Room Architecture Diagram

There are three major components in Room

1. Database
2. DAO (mapping Queries)
3. Entity (DB table)



Room Components for Notes app

Entity: Represents a table within the database. An exclusive database table is created for each class annotated with `@Entity`.

/ Annotate as Entity and define columns of the table, it will define the same name in the table column, if you want to provide the different name use as @ColumnInfo(name = "note_title") before each column*/*

@Entity

```
data class Note(  
    @PrimaryKey(autoGenerate = true)  
    val id: Int,  
    val title : String,  
    val note : String  
)
```

Table Name : Note (with three columns)	
Id(Primary Key, Auto generated)	Int
title	String
note	String

DAO

DAO: Contains the methods used for accessing the database. The DAO is the interface annotated with `@Dao` that mediates the access to objects in the database and its tables. There are four specific annotations for the basic DAO operations: `@Insert`, `@Update`, `@Delete`, and `@Query`.

The app uses the Room database to get the data access objects, or DAOs, associated with that database. The app then uses each DAO to get entities from the database and save any changes to those entities back to the database. Finally, the app uses an entity to get and set values that correspond to table columns within the database.

Room Components for Notes app

DAO – Define the functions need to manipulate the database with help of interface

@Dao

```
interface NoteDao {
```

```
    @Insert
```

```
    fun addNote(note:Note)
```

```
    @Query("SELECT * FROM NOTE ORDER BY id DESC")
```

```
    fun getAllNotes():List<Note>
```

```
    @Update
```

```
    fun updateNote(note:Note)
```

```
    @Delete
```

```
    fun deleteNote(note: Note)
```

```
}
```

Refer sample queries from: <https://developer.android.com/training/data-storage/room/accessing-data>

Database

- **Database:** Contains the database holder and serves as the main access point for the underlying connection to your app's persisted, relational data.

The class that's annotated with `@Database` should satisfy the following conditions:

- Be an abstract class that extends `RoomDatabase`.
 - Include the list of entities associated with the database within the annotation.
 - Contain an abstract method that has 0 arguments and returns the class that is annotated with `@Dao`.
-
- At runtime, you can acquire an instance of Database by calling `Room.databaseBuilder()`

Room Components for Notes app

Database Creation format

```
@Database(  
    entities = [Note::class],  
    version = 1  
)  
// If you have multiple tables mention inside [Note::class, Detail::class ]  
// Must Inherit from RoomDatabase  
abstract class NoteDatabase():RoomDatabase() {  
    abstract fun getNoteDao() : NoteDao // need to get the Dao for the entity  
}
```

Hands on Example[CRUD operation]

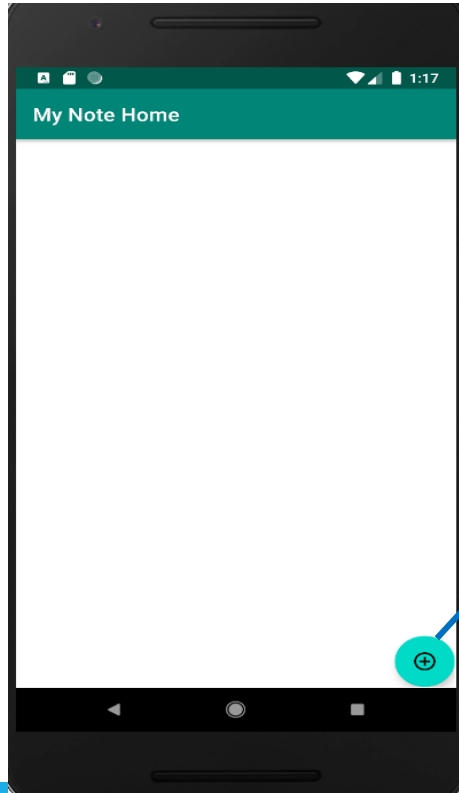
Problem Requirement : Create a Notes App with add note, update note and delete note. Added notes are shown in the RecyclerView List.

Through this example you will learn the following concepts.

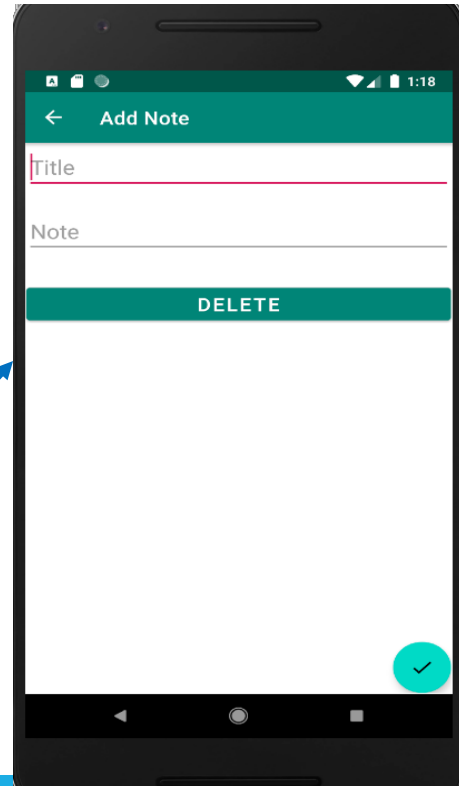
- How to use Room and perform CRUD operations.
- Navigation Component
- Kotlin Coroutines

Outcome of the Noteapp – Add Note

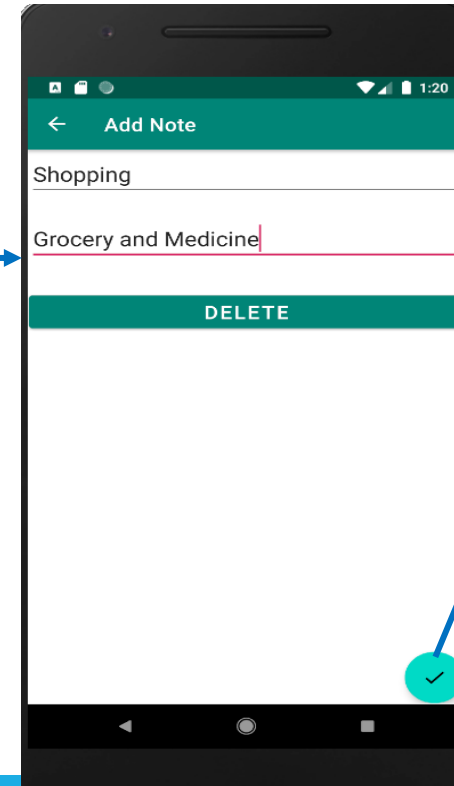
Home Screen
to Add notes



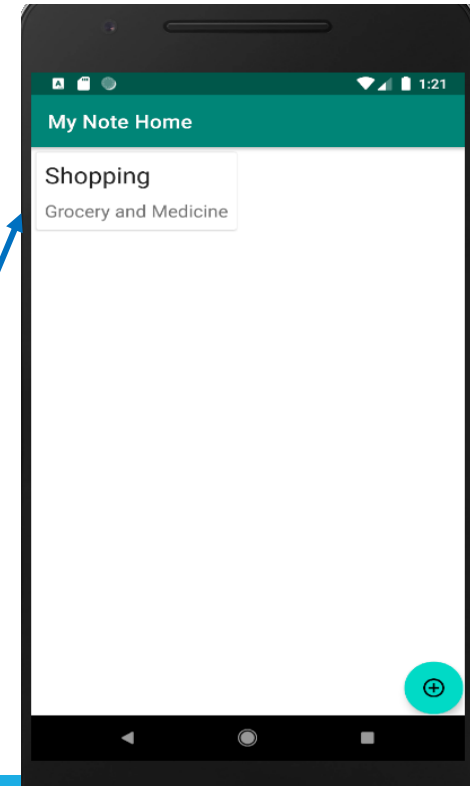
Click + FAB to get Add Note
Activity from Home Screen



Input title and Note
Press the Tick FAB to
save Note

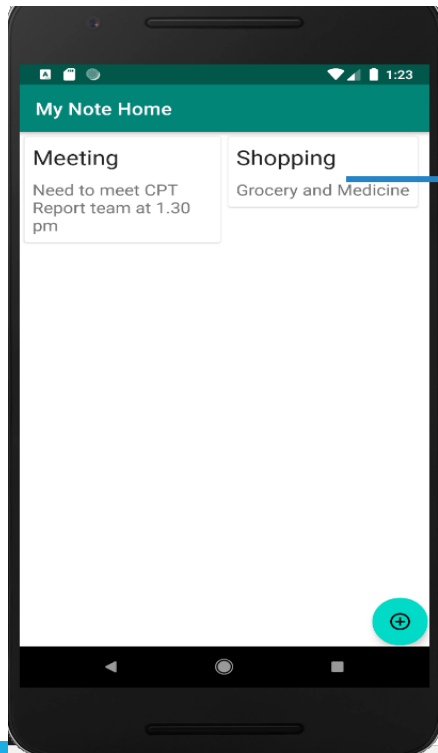


Saved Note is displayed
on the Home Screen
Recycling View

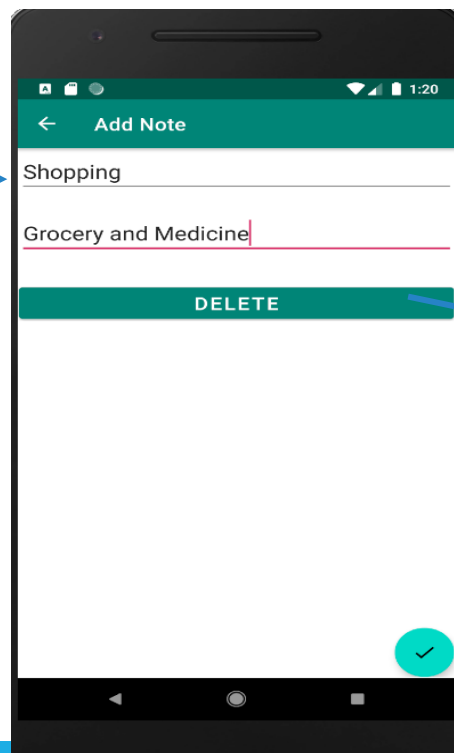


Outcome of the Noteapp – Delete Note

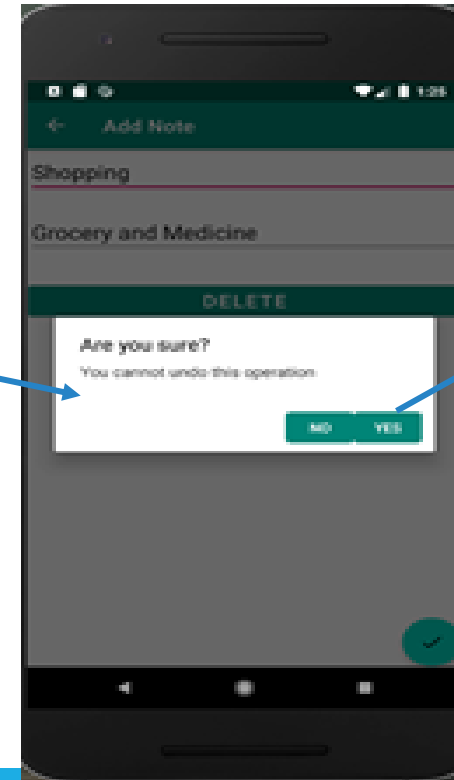
Home Screen
with two Notes



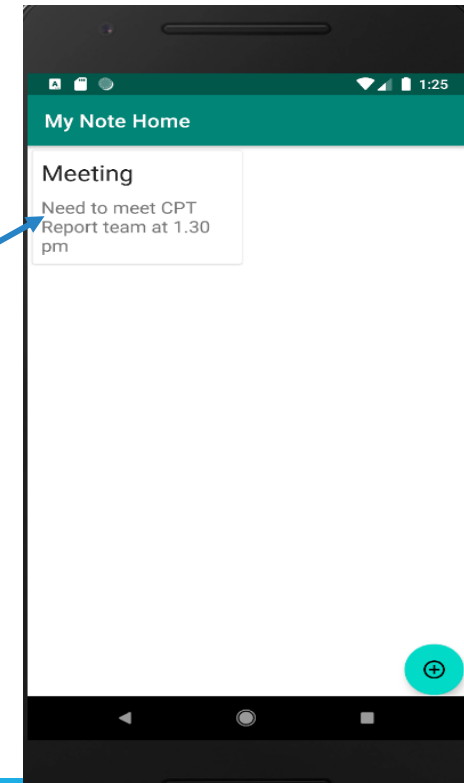
After clicking Shopping
note from Home Screen



After Pressing Delete
will ask dialogs for
delete confirmation

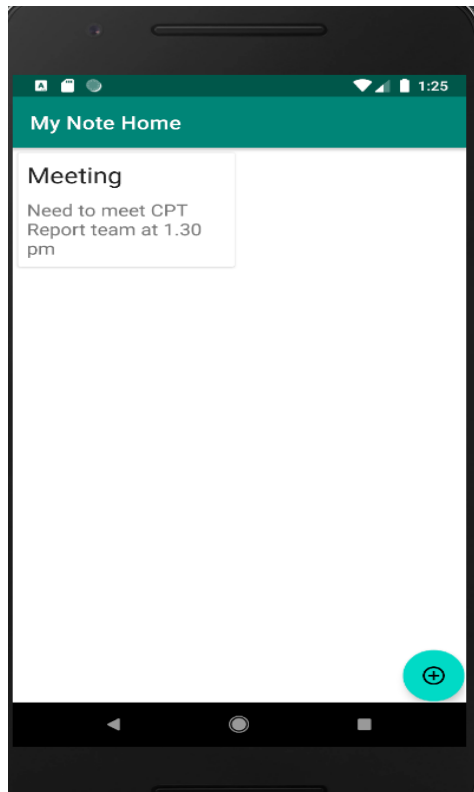


Home Screen after
deleting Shopping
Note

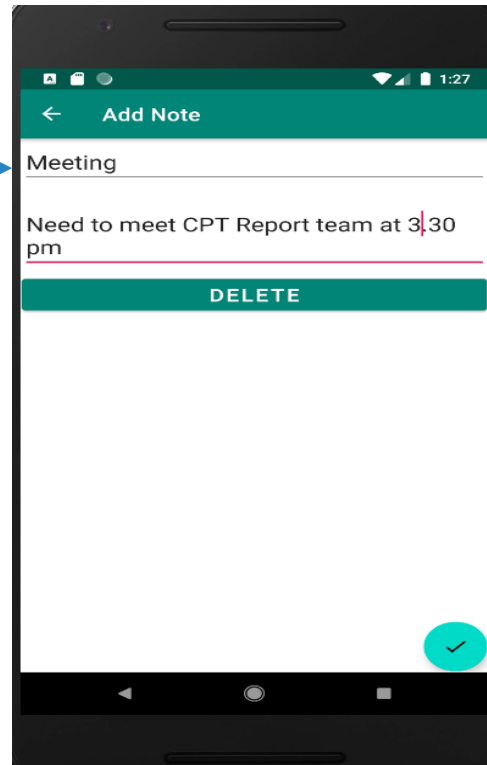


Outcome of the Noteapp –Update Note

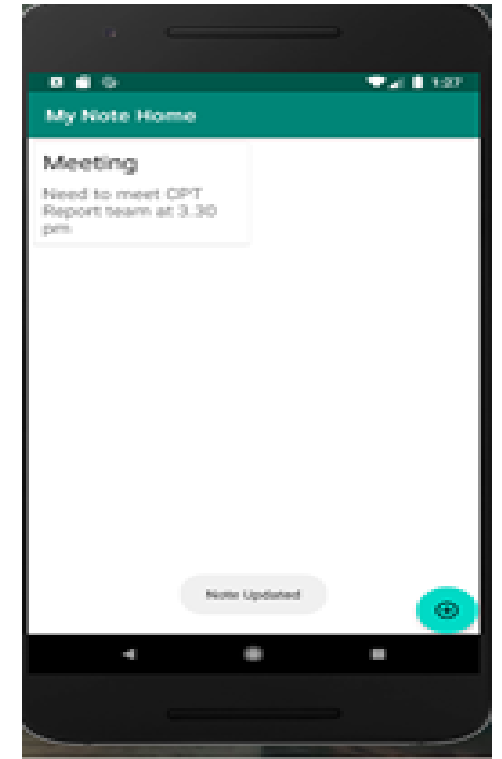
**Home Screen
with one Note**



**Update data and
press tick FAP**



**Home Screen after
updating Meeting Note**



Project Structure and nav-graph

NotesAppRoomDB > app > src > main > res > navigation > nav_graph.xml

app Nexus 6P API 27

Android

1: Project

Resource Manager

2: Structure

Build Variants

2: Favorites

Device File Explorer

Code Split Design

Destinations

HOST

activity_main (fragment)

GRAPH

homeFragment - Start

addNoteFragment

homeFragment

addNoteFragment

Attributes

actionAddNote action

id actionAddNote

destination addNoteFragm

Animations

enterAnim

exitAnim

popEnterAnim

popExitAnim

Argument Default Values

Note com.exampl default value

Pop Behavior

popUpTo none

popUpToIncl...

Launch Options

launchSingle...

java

com.example.notesapp

db

Note

NoteDao

NoteDatabase

ui

AddNoteFragment

BaseFragment

Helpers.kt

HomeFragment

MainActivity

NotesAdapter

com.example.notesapp (androidTest)

com.example.notesapp (test)

java (generated)

res

drawable

layout

activity_main.xml

fragment_add_note.xml

fragment_home.xml

note_layout.xml

mipmap

navigation

nav_graph.xml

values

g.java

DialogInterface.java

NotesAdapter.kt

fragment_add_note.xml

HomeFragment.kt

NoteDao.kt

nav_graph.xml

Building Room Database

- You need to use Kotlin companion object similar like static block in Java to build a database in a singleton pattern.
- Room DB will not work on main thread, if you try to run on main thread, will throw run time exception.
- Here, Kotlin Co-routines is used to maintain the database operations on the separate thread without blocking Main thread.
- operator fun invoke(context: Context) is used here to get the Database instance.
 - An interesting feature of the Kotlin language is the ability to define an "invoke operator".
 - When you specify an invoke operator on a class, it can be called on instances of the class
 - without a method name! This trick seems especially useful for classes that really have one method to be used.
 - If it is static, can directly call with the class name
- You can follow the same pattern for any database creation.

Code to build Database

```
abstract class NoteDatabase():RoomDatabase() {  
    abstract fun getNoteDao() : NoteDao  
  
    companion object {  
        @Volatile private var instance : NoteDatabase? = null  
        private val LOCK = Any()  
  
        operator fun invoke(context: Context) = instance ?: synchronized(LOCK){  
            instance ?: buildDatabase(context).also {  
                instance = it  
            }  
        }  
  
        // Function to build database  
        private fun buildDatabase(context: Context) = Room.databaseBuilder(  
            context.applicationContext,  
            NoteDatabase::class.java,  
            "notedatabase"  
        ).build()  
    }  
}
```

Build database - Code explanation

- @Volatile means that this field is immediately made visible to other threads
@Volatile private var instance : NoteDatabase? = null
- Keyword Any → Every Kotlin class has [Any] as a superclass. The root of the Kotlin class hierarchy like Object in Java
private val LOCK = Any()
- Help of ?: elvis operator check if the instance is not null return the instance, if it is null then synchronized block will work, inside the block do the nullability check and call the function buildDatabase if database instance is null.

```
operator fun invoke(context: Context) = instance ?: synchronized(LOCK){  
    instance ?: buildDatabase(context).also { it:NoteDatabase  
        instance = it  
    }  
}
```

Kotlin having higher order scoping functions apply, with, let, also, and run. Here **also** passes an object as a parameter(it:NoteDatabase) and returns the same object(it).

Build database - Code explanation

```
private fun buildDatabase(context: Context) = Room.databaseBuilder(  
    context.applicationContext,  
    NoteDatabase::class.java,  
    "notedatabase"  
).build()
```

- Room.databaseBuilder, creates a persistent database by accepting three arguments.
 - arg1: Context context → Here you can pass **context.applicationContext**
 - arg2: Class<T> kclass → the abstract class which is annotated with @Database and extends RoomDatabase. Here you can pass **NoteDatabase::class.java**
 - arg3 : String name → Name of the database. Here you can pass **"notedatabase"**

Fragments Code Explanation

- Two Fragments are used in this code
 - HomeFragment
 - AddNoteFragment
- Both are inherited from the abstract class BaseFragment extends Fragment() and implements CoroutineScope.
- Kotlin Coroutines
 - A new way of managing background threads that can simplify code by reducing the need for callbacks. Coroutines are a Kotlin feature that convert async callbacks for long-running tasks, such as database or network access.
 - Network request cannot run through Main thread, we need to run it on a separate thread using Coroutines. Helps to improve the app performance.

Kotlin Coroutine concepts and code

Below concepts and codes are introduced in the BaseFragment class

abstract class BaseFragment : Fragment(), CoroutineScope

- **CoroutineScope(Interface)**

- In Kotlin, all coroutines run inside a CoroutineScope. A scope controls the lifetime of coroutines through its job. When you cancel the job of a scope, it cancels all coroutines started in that scope.

- **Declare Job**

- **private lateinit var job: Job**

Kotlin Coroutine concepts and code

- **CoroutineContext(Interface)**

- Persistent context for the coroutine. CoroutineScope needs a property of CoroutineContext. The main elements are Job and Dispatcher.

override val coroutineContext: CoroutineContext
get() = job + Dispatchers.Main

- A dispatcher controls which thread runs a coroutine. For coroutines started by the UI, it is typically correct to start them on Dispatchers.Main which is the main thread on Android. A coroutine started on Dispatchers.Main won't block the main thread while suspended.

Kotlin Coroutine concepts and code

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    // Create an Instance for the Job()  
    job = Job()  
}  
// Cancel the Job in onDestroy()  
override fun onDestroy() {  
    super.onDestroy()  
    // Cancel the Job  
    job.cancel()  
}
```

```

launch { this: CoroutineScope
    context?.let { it: Context
        val mNote = Note(noteTitle, noteBody)
        // note == null means Inserting a new Note
        if (note == null) {
            NoteDatabase(it).getNoteDao().addNote(mNote)
            it.toast("Note Saved")
        } else {
            // Update the note
            mNote.id = note!!.id
            NoteDatabase(it).getNoteDao().updateNote(mNote)
            it.toast("Note Updated")
        }
    }
    // after adding a note need to return to Home_Fragment as per the navigation directions
    val action : NavDirections = AddNoteFragmentDirections.actionSaveNote()
    Navigation.findNavController(view).navigate(action)
}

```

Kotlin Coroutine concepts and launch code

- **launch** is a function that creates a coroutine and dispatches the execution of its function body to the corresponding dispatcher.
- By calling this launch method, can execute the DB tasks under CoroutineScope.

Example : Inside AddNoteFragment class, launch method once you press save FAB. Refer the left side code screenshot.

```

class AddNoteFragment :BaseFragment() {
    private var note: Note? = null
}

```


Kotlin Coroutine concepts and suspend keyword

- **suspend** keyword enforce a function to be called from within a coroutine or another suspend function
- The biggest merit of coroutines is that they can *suspend* without blocking a thread.
- So, we have to mark functions that *may suspend* explicitly in the code.
- All methods in NoteDao declared with the keyword suspend to execute DB functionalities under Coroutine scope.

@Dao

interface NoteDao {

 @Insert

suspend fun addNote(note:Note)

 @Query("SELECT * FROM NOTE ORDER BY id DESC")

suspend fun getAllNotes():List<Note>

 @Update

suspend fun updateNote(note:Note)

 @Delete

suspend fun deleteNote(note: Note)

}

Kotlin Extension Function

This means we can extend any class with new features even if we don't have access to the source code. Kotlin extension function provides a facility to "add" methods to class without inheriting a class

Example : You can define the extended functions in a Kotlin File

```
fun Context.toast(message:String) =
```

```
    Toast.makeText(this, message, Toast.LENGTH_SHORT).show()
```

```
fun String.add(s1:String,s2:String) = this+s1+s2
```

Referring in an Activity

```
btnclick.setOnClickListener {
```

```
    tv.text = "Kotlin".add(" is a Functional " ,"Object Oriented")
```

```
    this.toast("Test Extensions")}
```

```
}
```

Reference

- Refer the complete code from the NotesApp.
- Refer the complete step by implementation using the Lesson-10 RoomDB NotesApp implementation step by step.pdf.
- Refer Kotlin Scope functions :let, also, apply, with and run.
<https://kotlinlang.org/docs/reference/scope-functions.html>
- Accessing data using Room DAO to write Different queries
<https://developer.android.com/training/data-storage/room/accessing-data>
Simple example to follow app architecture with Room, LiveData and ViewModel
<https://developer.android.com/codelabs/android-room-with-a-view-kotlin#0>

Main Point 3

Room is a database library, provides an abstraction layer over SQLite to allow for more robust database accesses. Kotlin Coroutines are new way of managing background threads that can simplify code by reducing the need for callbacks. Coroutines are a Kotlin feature that convert async callbacks for long-running tasks, such as database or network access. ***Science of consciousness: With growth of experience, the abstraction of TC becomes more concrete. So, one can easily brought out robust potential effortlessly to access.***

UNITY CHART

Infinite organizing power

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Collapse of infinity to a point embodied in Stacks and Queues

1. Android Jetpack, a collection of software components designed to accelerate Android development and make writing high-quality apps easier. Farseeing visions of truth in the practicalities of daily life.
 2. Android data retrieval and storage methods allow us to adapt to changing user and business needs just as Creative Intelligence is Insightful and Foresightful.
-
3. **Transcendental Consciousness:** Transcendental Consciousness is the unbounded value of awareness.
 4. ***Impulses within the Transcendental field:*** *These impulses are responsible for organizing the whole infinitely diverse universe.*
 5. ***Wholeness moving within Itself:*** *In Unity Consciousness, creation is seen as the interaction of unboundedness and point value: the unbounded collapses to its point value; point value expands to infinity; all within the wholeness of awareness.*

