ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

ACTIVITY, FRAGMENTS & NAVIGATION (WEEKS 7-9)

ACTIVITY LIFECYCLE & STATE MANAGEMENT IN ANDROID

CHAPTER 1: UNDERSTANDING ACTIVITY LIFECYCLE

1.1 What is an Activity in Android?

- **An Activity** is a single screen in an Android app that interacts with the user.
- Every Android app has at least one **Activity**, serving as the **entry point** for users.
- Activities can transition between different states, and managing these states is crucial for **performance, efficiency, and user experience**.

✅ **Examples of Activities:**

✓ **Login Screen Activity** – Handles user authentication.

✓ **Home Screen Activity** – Displays main app content.

✓ **Settings Activity** – Allows users to configure preferences.

📌 **Example:**

- **A social media app may have separate activities for login, home feed, and profile pages.**

1.2 Android Activity Lifecycle

🔹 The **Activity Lifecycle** describes the different states an activity goes through from when it is created to when it is destroyed.

🔹 Android provides **lifecycle methods** that allow developers to manage these states efficiently.

✅ Activity Lifecycle Stages & Methods

Lifecycle Stage	Method	Description
Created	onCreate()	Called when the activity is first created. Used for initialization.
Started	onStart()	Activity becomes visible but not interactive.
Resumed	onResume()	Activity is visible and interactive.
Paused	onPause()	Activity is partially visible, but another activity is in the foreground.
Stopped	onStop()	Activity is no longer visible but still exists in memory.
Destroyed	onDestroy()	Activity is completely removed from memory.

📌 Example:

- **When a user opens an app, onCreate() initializes the UI, onStart() makes it visible, and onResume() makes it**

interactive. If the user receives a call, the activity goes into onPause().

CHAPTER 2: LIFECYCLE METHODS & IMPLEMENTATION

2.1 Overriding Activity Lifecycle Methods in Java/Kotlin

📌 **Example: Java Implementation**

```
import android.os.Bundle;
import android.util.Log;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d("Activity Lifecycle", "onCreate() called");
  }

  @Override
  protected void onStart() {
    super.onStart();
    Log.d("Activity Lifecycle", "onStart() called");
```

```
}

@Override
protected void onResume() {
    super.onResume();
    Log.d("Activity Lifecycle", "onResume() called");
}

@Override
protected void onPause() {
    super.onPause();
    Log.d("Activity Lifecycle", "onPause() called");
}

@Override
protected void onStop() {
    super.onStop();
    Log.d("Activity Lifecycle", "onStop() called");
}

@Override
protected void onDestroy() {
```

```
        super.onDestroy();
        Log.d("Activity Lifecycle", "onDestroy() called");
    }
}
```

📌 **Example: Kotlin Implementation**

```
import android.os.Bundle
import android.util.Log
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        Log.d("Activity Lifecycle", "onCreate() called")
    }

    override fun onStart() {
        super.onStart()
        Log.d("Activity Lifecycle", "onStart() called")
    }

    override fun onResume() {
```

```
        super.onResume()
        Log.d("Activity Lifecycle", "onResume() called")
    }

    override fun onPause() {
        super.onPause()
        Log.d("Activity Lifecycle", "onPause() called")
    }

    override fun onStop() {
        super.onStop()
        Log.d("Activity Lifecycle", "onStop() called")
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.d("Activity Lifecycle", "onDestroy() called")
    }
}
```

📌 **Output in Logcat:**

```
D/Activity Lifecycle: onCreate() called
D/Activity Lifecycle: onStart() called
```

D/Activity Lifecycle: onResume() called

(When user presses the home button:)

D/Activity Lifecycle: onPause() called

D/Activity Lifecycle: onStop() called

CHAPTER 3: STATE MANAGEMENT IN ANDROID

3.1 What is State Management?

🔹 **State Management** refers to **handling UI data** when an activity is recreated due to configuration changes (e.g., screen rotation) or system interruptions (e.g., memory constraints).

🔹 Android provides mechanisms to **save and restore** activity state efficiently.

✅ **Common State Management Scenarios:**

✓ **Screen Rotation** – Prevent UI reset when the device is rotated.

✓ **Process Kill & Restart** – Save user progress in case the app is removed from memory.

✓ **Activity Recreation** – Maintain UI state when switching between activities.

📌 **Example:**

- **If a user enters text in an input field and rotates the screen, the text should remain visible.**

3.2 Saving & Restoring State in Android

✅ **Use onSaveInstanceState() & onRestoreInstanceState()**

📌 **Java Implementation**

```
@Override
protected void onSaveInstanceState(Bundle outState) {
   super.onSaveInstanceState(outState);
   outState.putString("username", editText.getText().toString());
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState)
{
   super.onRestoreInstanceState(savedInstanceState);
   editText.setText(savedInstanceState.getString("username"));
}
```

📌 Kotlin Implementation

```
override fun onSaveInstanceState(outState: Bundle) {
   super.onSaveInstanceState(outState)
   outState.putString("username", editText.text.toString())
}

override fun onRestoreInstanceState(savedInstanceState: Bundle) {
   super.onRestoreInstanceState(savedInstanceState)
   editText.setText(savedInstanceState.getString("username"))
}
```

✅ Using ViewModel for State Management

📌 ViewModel Kotlin Implementation

```
class MyViewModel : ViewModel() {
  var username: String? = null
}
```

📌 Using ViewModel in Activity

```
val model: MyViewModel by viewModels()
editText.setText(model.username)
```

CHAPTER 4: HANDLING CONFIGURATION CHANGES

4.1 What Are Configuration Changes?

- **Configuration changes** occur when the device's properties change, causing the activity to restart.
- Examples include:

✓ Screen rotation (portrait ⇄ landscape)
✓ Changing system language
✓ Switching dark mode on/off
✓ Multi-window mode on tablets

4.2 Preventing Activity Restart

✅ Option 1: Handle Configuration Changes Manually

📌 Declare in AndroidManifest.xml

```
<activity
  android:name=".MainActivity"
  android:configChanges="orientation|screenSize" />
```

📌 Java Code to Handle Rotation Manually

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
  super.onConfigurationChanged(newConfig);
  if (newConfig.orientation ==
Configuration.ORIENTATION_LANDSCAPE) {
    Log.d("ConfigChange", "Landscape Mode");
  } else {
    Log.d("ConfigChange", "Portrait Mode");
  }
}
```

✅ Option 2: Use ViewModel for Persisting Data

📌 Example:

- **A timer app should keep counting even if the screen rotates.**

Exercise: Test Your Understanding

- **What are the key lifecycle methods of an activity?**
- **Explain the difference between onPause() and onStop().**
- **Why do we use onSaveInstanceState()?**
- **How does ViewModel help in state management?**
- **What happens when an activity is killed due to low memory?**

Conclusion

✅ **Understanding the Android Activity Lifecycle is crucial for building efficient apps.**

✅ **Lifecycle methods allow developers to manage UI transitions and resource cleanup.**

✅ **State management ensures a smooth user experience during configuration changes.**

✅ **ViewModel and onSaveInstanceState() help in handling data persistence.**

PASSING DATA BETWEEN ACTIVITIES IN ANDROID

CHAPTER 1: INTRODUCTION TO PASSING DATA BETWEEN ACTIVITIES

1.1 What is Activity Communication?

🔹 In Android, an **Activity** represents a single screen with a user interface.

🔹 Often, **multiple activities** work together to create a complete user experience.

🔹 **Passing data between activities** is essential for sharing information such as:

- User inputs (e.g., username, email).
- Selected items from a list (e.g., selected product in an e-commerce app).
- Results from one activity to another.

✅ **Why Pass Data Between Activities?**

✓ Maintain state across screens.

✓ Share user input or selection.

✓ Enable interaction between app components.

📌 **Example:**

- **A user selects a product from a list in ProductListActivity and sees the details in ProductDetailActivity.**

CHAPTER 2: USING INTENT TO PASS DATA

2.1 What is an Intent?

- **Intent** is a messaging object used to **navigate between activities** and pass data.
- Android supports two types of intents:
  - **Explicit Intent** – Used to start a specific activity within the same app.
  - **Implicit Intent** – Used to request an action from another app (e.g., opening a webpage).

2.2 Passing Data Using Intent (Explicit Intent)

✅ Step 1: Create the First Activity (MainActivity)

```
// MainActivity.java

package com.example.datapassing;


import android.content.Intent;

import android.os.Bundle;

import android.widget.Button;

import android.widget.EditText;

import androidx.appcompat.app.AppCompatActivity;


public class MainActivity extends AppCompatActivity {


  @Override

  protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        EditText nameInput = findViewById(R.id.editTextName);
        Button sendButton = findViewById(R.id.buttonSend);

        sendButton.setOnClickListener(v -> {
            String name = nameInput.getText().toString();
            Intent intent = new Intent(MainActivity.this,
SecondActivity.class);
            intent.putExtra("USER_NAME", name); // Passing data
            startActivity(intent);
        });
    }
}
```

✅ **Step 2: Retrieve Data in Second Activity (SecondActivity)**

```
// SecondActivity.java
package com.example.datapassing;

import android.os.Bundle;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
```

```
public class SecondActivity extends AppCompatActivity {


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);


        TextView nameDisplay = findViewById(R.id.textViewName);


        // Retrieve data
        String userName = getIntent().getStringExtra("USER_NAME");
        nameDisplay.setText("Hello, " + userName);
    }
}
```

📌 **Example:**

- **User enters their name in MainActivity, and SecondActivity displays "Hello, [UserName]".**

CHAPTER 3: PASSING COMPLEX DATA USING PARCELABLE

3.1 Why Use Parcelable?

🔹 **Parcelable** is an efficient way to **pass objects between activities**.

🔹 It is **faster than Serializable** since it is optimized for Android.

✅ Step 1: Create a Model Class Implementing Parcelable

```
// User.java
package com.example.datapassing;

import android.os.Parcel;
import android.os.Parcelable;

public class User implements Parcelable {
    String name;
    int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    protected User(Parcel in) {
        name = in.readString();
        age = in.readInt();
    }

    public static final Creator<User> CREATOR = new Creator<User>()
{
```

```
    @Override
    public User createFromParcel(Parcel in) {
      return new User(in);
    }

    @Override
    public User[] newArray(int size) {
      return new User[size];
    }
  };

  @Override
  public int describeContents() {
    return 0;
  }

  @Override
  public void writeToParcel(Parcel dest, int flags) {
    dest.writeString(name);
    dest.writeInt(age);
  }
}
```

✅ Step 2: Pass Parcelable Object in Intent

```
// MainActivity.java

Intent intent = new Intent(MainActivity.this, SecondActivity.class);

User user = new User("John Doe", 25);

intent.putExtra("USER_DATA", user);

startActivity(intent);
```

✅ Step 3: Retrieve Parcelable Object in Second Activity

```
// SecondActivity.java

User user = getIntent().getParcelableExtra("USER_DATA");

textView.setText("Name: " + user.name + "\nAge: " + user.age);
```

📌 Example:

- **Pass a User object containing name and age between activities.**

CHAPTER 4: RECEIVING DATA BACK USING STARTACTIVITYFORRESULT() (DEPRECATED) AND ACTIVITY RESULT API

4.1 When Do We Need to Receive Data Back?

✅ Common Use Cases:

✓ Returning selected values (e.g., user selects an item from a list).
✓ Returning a success message after an operation (e.g., user updates profile).
✓ Capturing user input from another screen (e.g., image selection).

4.2 Using Activity Result API (Modern Approach)

✅ Step 1: Register the Activity Result in MainActivity

```
// MainActivity.java

private final ActivityResultLauncher<Intent> resultLauncher =

  registerForActivityResult(new
ActivityResultContracts.StartActivityForResult(),

    result -> {

      if (result.getResultCode() == Activity.RESULT_OK &&
result.getData() != null) {

        String message =
result.getData().getStringExtra("RESULT_MESSAGE");

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

      }

    });


@Override

protected void onCreate(Bundle savedInstanceState) {

  super.onCreate(savedInstanceState);

  setContentView(R.layout.activity_main);


  Button openSecondActivity = findViewById(R.id.buttonOpen);

  openSecondActivity.setOnClickListener(v -> {
```

```
    Intent intent = new Intent(MainActivity.this,
SecondActivity.class);
    resultLauncher.launch(intent);
  });
}
```

✅ **Step 2: Send Data Back in SecondActivity**

```
// SecondActivity.java
Button sendBack = findViewById(R.id.buttonSendBack);
sendBack.setOnClickListener(v -> {
  Intent resultIntent = new Intent();
  resultIntent.putExtra("RESULT_MESSAGE", "Data received
successfully!");
  setResult(RESULT_OK, resultIntent);
  finish();
});
```

📌 **Example:**

- **User submits a form in SecondActivity, and MainActivity receives a confirmation message.**

CHAPTER 5: BEST PRACTICES FOR PASSING DATA BETWEEN ACTIVITIES

✅ **Use Intent.putExtra() for primitive data types (String, int, boolean).**

✅ **Use Parcelable for passing complex objects instead of**

Serializable (better performance).

✅ Use Activity Result API instead of startActivityForResult() (deprecated).

✅ Avoid passing large data directly; store it in a database and pass the reference.

✅ Use SharedPreferences or ViewModel for storing session-related data.

📌 Example:

- Instead of passing a list of 1000 items, store them in a local database and pass only the selected item ID.

📌 Exercise: Test Your Understanding

🔹 What is the difference between Serializable and Parcelable?

🔹 How do you pass a list of objects between activities?

🔹 Why is Activity Result API recommended over startActivityForResult()?

🔹 How can you share user login details across multiple activities?

🔹 Write code to pass an image URI from one activity to another.

📌 Conclusion

✅ Passing data between activities is crucial for building interactive Android applications.

✅ Intents allow passing simple data like Strings, numbers, and Booleans.

✅ Parcelable is preferred for complex objects due to better performance.

✅ **Activity Result API makes receiving data back from another activity easier and more efficient.**

✅ **Following best practices ensures optimized memory usage and better app performance.**

Fragment Lifecycle & Fragment Transactions in Android

Chapter 1: Introduction to Fragments in Android

1.1 What is a Fragment?

🔹 **A Fragment is a modular, reusable UI component** that represents a portion of a screen in an Android app.

🔹 Fragments allow developers to create **flexible and scalable** UI designs, especially for tablets and multi-pane layouts.

🔹 A fragment must be **hosted within an Activity** and cannot exist independently.

✅ **Why Use Fragments?**

✓ **Reusable UI components** that can be used across multiple activities.

✓ **Modular design** improves maintainability and scalability.

✓ **Supports flexible UI layouts** for different screen sizes.

✓ **Reduces code duplication** by reusing fragment components.

📌 **Example Use Cases:**

- A **news app** displays headlines in one fragment and article details in another.
- A **shopping app** uses fragments to show product lists and product details separately.

Chapter 2: Fragment Lifecycle

2.1 Understanding the Fragment Lifecycle

- The **Fragment lifecycle** consists of **seven key callback methods,** each representing a stage in the fragment's existence.
- These methods help manage fragment behavior **when added, removed, paused, or resumed**.

✅ Fragment Lifecycle Flow:

Lifecycle Method	Description
onAttach()	Called when the fragment is attached to its parent Activity.
onCreate()	Initializes essential components (non-UI related work).
onCreateView()	Inflates the fragment's UI layout.
onViewCreated()	Called after the view is created to set up UI elements.
onStart()	Fragment becomes visible to the user.
onResume()	The fragment is active and can interact with the user.
onPause()	Fragment is partially visible (another activity is in front).
onStop()	The fragment is no longer visible.
onDestroyView()	Cleans up views to prevent memory leaks.
onDestroy()	Cleans up resources and background tasks.
onDetach()	The fragment is completely removed from the Activity.

📌 Example Flow:

- **When an app starts,** onAttach() → onCreate() → onCreateView() → onViewCreated() → onStart() → onResume().
- **When the user navigates away,** onPause() → onStop().
- **When the fragment is destroyed,** onDestroyView() → onDestroy() → onDetach().

2.2 Fragment Lifecycle Example in Code

✅ **Implementing a Simple Fragment Lifecycle in Kotlin**

📌 **MyFragment.kt**

```
class MyFragment : Fragment() {


  override fun onAttach(context: Context) {
    super.onAttach(context)
    Log.d("Fragment Lifecycle", "onAttach called")
  }


  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d("Fragment Lifecycle", "onCreate called")
  }


  override fun onCreateView(
```

```
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
  ): View? {
    Log.d("Fragment Lifecycle", "onCreateView called")
    return inflater.inflate(R.layout.fragment_my, container, false)
  }

  override fun onViewCreated(view: View, savedInstanceState:
Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    Log.d("Fragment Lifecycle", "onViewCreated called")
  }

  override fun onStart() {
    super.onStart()
    Log.d("Fragment Lifecycle", "onStart called")
  }

  override fun onResume() {
    super.onResume()
    Log.d("Fragment Lifecycle", "onResume called")
  }
```

```
override fun onPause() {
    super.onPause()
    Log.d("Fragment Lifecycle", "onPause called")
}

override fun onStop() {
    super.onStop()
    Log.d("Fragment Lifecycle", "onStop called")
}

override fun onDestroyView() {
    super.onDestroyView()
    Log.d("Fragment Lifecycle", "onDestroyView called")
}

override fun onDestroy() {
    super.onDestroy()
    Log.d("Fragment Lifecycle", "onDestroy called")
}

override fun onDetach() {
    super.onDetach()
```

```
        Log.d("Fragment Lifecycle", "onDetach called")
    }
}
```

📌 **Example:**

- When the fragment is created and removed, the **log messages track lifecycle changes**.

CHAPTER 3: FRAGMENT TRANSACTIONS

3.1 What is a Fragment Transaction?

🔹 A **FragmentTransaction** allows adding, removing, replacing, or hiding fragments dynamically in an activity.

🔹 It is managed using the **FragmentManager**.

✅ **Common Fragment Transactions:**

✓ add() – Add a new fragment.
✓ replace() – Replace an existing fragment.
✓ remove() – Remove a fragment.
✓ hide() – Hide a fragment temporarily.
✓ show() – Show a hidden fragment.
✓ addToBackStack() – Allows back navigation between fragments.

3.2 Adding a Fragment Dynamically

✅ **Step 1: Define the Fragment Layout (fragment_example.xml)**

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:gravity="center"
    android:orientation="vertical">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is a Fragment"
        android:textSize="20sp"
        android:textStyle="bold"/>
</LinearLayout>
```

✅ **Step 2: Create a Fragment Class (ExampleFragment.kt)**

```
class ExampleFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_example, container,
false)
    }
}
```

✅ Step 3: Add Fragment to an Activity Dynamically

📌 MainActivity.kt

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)


        val fragmentButton =
findViewById<Button>(R.id.buttonLoadFragment)
        fragmentButton.setOnClickListener {
            val fragment = ExampleFragment()
            supportFragmentManager.beginTransaction()
                .replace(R.id.fragmentContainer, fragment) // Replace
existing fragment
                .addToBackStack(null) // Add to back stack for navigation
                .commit()
        }
    }
}
```

✅ Step 4: Add FragmentContainerView in activity_main.xml

```
<Button
    android:id="@+id/buttonLoadFragment"
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:text="Load Fragment" />

<androidx.fragment.app.FragmentContainerView
    android:id="@+id/fragmentContainer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

📌 **Example:**

- **When the button is clicked, the fragment is dynamically loaded inside the container.**

CHAPTER 4: FRAGMENT NAVIGATION & BACK STACK

4.1 Using Back Stack to Navigate Between Fragments

🔹 The addToBackStack() method allows users to navigate back to the previous fragment.

✅ **Example: Replacing a Fragment and Allowing Back Navigation**

```
supportFragmentManager.beginTransaction()
    .replace(R.id.fragmentContainer, NewFragment())
    .addToBackStack(null)
    .commit()
```

📌 **Example:**

- When the user clicks back, the previous fragment is restored instead of closing the activity.

Exercise: Test Your Understanding

- What are the major lifecycle methods of a Fragment?
- How does FragmentTransaction work in Android?
- How do you prevent memory leaks when using Fragments?
- Implement a simple app with two fragments and a button to switch between them.
- What is the purpose of addToBackStack() in Fragment Transactions?

Conclusion

✅ Fragments help create flexible and reusable UI components in Android applications.

✅ Understanding the Fragment lifecycle is crucial for managing UI components efficiently.

✅ Fragment Transactions enable adding, replacing, and removing fragments dynamically.

✅ Using back stack ensures proper navigation between fragments without restarting activities.

BOTTOM NAVIGATION & DRAWER NAVIGATION IN ANDROID

CHAPTER 1: INTRODUCTION TO ANDROID NAVIGATION COMPONENTS

1.1 What is Navigation in Android?

🔹 **Navigation** in Android refers to the **interactions that allow users to move between different screens, activities, or fragments within an app**.

🔹 Android provides multiple navigation components to enhance user experience and ensure smooth app usability.

✅ **Types of Navigation in Android:**

✓ **Bottom Navigation** – Used for primary navigation at the bottom of the screen.

✓ **Drawer Navigation (Navigation Drawer)** – A side menu that slides in from the left/right.

✓ **Tab Navigation** – Allows switching between pages using tabs.

✓ **Navigation Component** – A Jetpack library for managing navigation.

📌 **Example:**

- **Instagram uses Bottom Navigation for Home, Search, Reels, and Profile, while Gmail uses Drawer Navigation for folders and labels.**

CHAPTER 2: BOTTOM NAVIGATION IN ANDROID

2.1 What is Bottom Navigation?

🔹 **Bottom Navigation** provides quick access to **primary destinations** in an app using a bar at the bottom of the screen.

🔹 Each item in the bottom navigation bar is represented by an **icon and an optional text label**.

✅ **Key Features of Bottom Navigation:**

✓ Supports **3-5 primary destinations**.

✓ Uses **icons and labels** for better usability.

✓ Enhances **user experience** with easy access to core features.

📌 **Example:**

- **YouTube uses Bottom Navigation for Home, Shorts, Subscriptions, and Library.**

2.2 Adding Bottom Navigation in an Android App

✅ **Step 1: Add Bottom Navigation to activity_main.xml**

```
<com.google.android.material.bottomnavigation.BottomNavigatio
nView
  android:id="@+id/bottomNavigation"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:background="?android:attr/windowBackground"
  app:menu="@menu/bottom_nav_menu"
  app:layout_constraintBottom_toBottomOf="parent"/>
```

✅ **Step 2: Create bottom_nav_menu.xml in res/menu/**

```
<menu
xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/nav_home"
        android:icon="@drawable/ic_home"
        android:title="Home"
        app:showAsAction="ifRoom"/>

    <item
        android:id="@+id/nav_search"
        android:icon="@drawable/ic_search"
        android:title="Search"/>

    <item
        android:id="@+id/nav_profile"
        android:icon="@drawable/ic_profile"
        android:title="Profile"/>
</menu>
```

✅ Step 3: Handle Navigation Selection in MainActivity

```
BottomNavigationView bottomNavigation =
findViewById(R.id.bottomNavigation);

bottomNavigation.setOnNavigationItemSelectedListener(item -> {
```

```
    switch (item.getItemId()) {
        case R.id.nav_home:
            // Load Home Fragment
            return true;
        case R.id.nav_search:
            // Load Search Fragment
            return true;
        case R.id.nav_profile:
            // Load Profile Fragment
            return true;
    }
    return false;
});
```

📌 **Example:**

- **A social media app switches between Home, Search, and Profile screens using Bottom Navigation.**

CHAPTER 3: DRAWER NAVIGATION (NAVIGATION DRAWER)

3.1 What is Drawer Navigation?

🔹 **Drawer Navigation** (Navigation Drawer) is a **side panel** that slides in from the left (or right) to provide access to app destinations and settings.

🔹 It is commonly used in apps with **multiple sections** or where additional navigation options are needed.

✅ **Key Features of Drawer Navigation:**

✓ Provides access to **secondary destinations**.

✓ Works well for apps with **many sections**.

✓ Supports **icons and labels** for better readability.

📌 **Example:**

- **Gmail uses a Navigation Drawer for switching between folders (Inbox, Sent, Drafts).**

3.2 Implementing a Navigation Drawer in an Android App

✅ **Step 1: Modify activity_main.xml**

```
<androidx.drawerlayout.widget.DrawerLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/drawerLayout"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <!-- Main Content -->
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <!-- Toolbar -->
```

```
    <androidx.appcompat.widget.Toolbar
      android:id="@+id/toolbar"
      android:layout_width="match_parent"
      android:layout_height="?attr/actionBarSize"
      android:background="?attr/colorPrimary"
      app:title="My App"/>

    <!-- Content Frame -->
    <FrameLayout
      android:id="@+id/contentFrame"
      android:layout_width="match_parent"
      android:layout_height="match_parent"/>
  </LinearLayout>

  <!-- Navigation Drawer -->
  <com.google.android.material.navigation.NavigationView
    android:id="@+id/navigationView"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    app:menu="@menu/drawer_menu"/>
</androidx.drawerlayout.widget.DrawerLayout>
```

✅ Step 2: Create drawer_menu.xml in res/menu/

```
<menu
xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/nav_home"
    android:icon="@drawable/ic_home"
    android:title="Home"/>

  <item
    android:id="@+id/nav_settings"
    android:icon="@drawable/ic_settings"
    android:title="Settings"/>

  <item
    android:id="@+id/nav_logout"
    android:icon="@drawable/ic_logout"
    android:title="Logout"/>
</menu>
```

✅ Step 3: Handle Navigation Item Clicks in MainActivity.java

```
DrawerLayout drawerLayout = findViewById(R.id.drawerLayout);

NavigationView navigationView =
findViewById(R.id.navigationView);
```

```
ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(this, drawerLayout, toolbar, R.string.open, R.string.close);

drawerLayout.addDrawerListener(toggle);

toggle.syncState();


navigationView.setNavigationItemSelectedListener(item -> {
   switch (item.getItemId()) {
      case R.id.nav_home:
         // Load Home Fragment
         return true;
      case R.id.nav_settings:
         // Open Settings
         return true;
      case R.id.nav_logout:
         // Logout logic
         return true;
   }
   return false;
});
```

📌 **Example:**

- **A file manager app uses Drawer Navigation to switch between file categories like Images, Videos, and Documents.**

CHAPTER 4: BOTTOM NAVIGATION VS DRAWER NAVIGATION – WHEN TO USE?

Feature	Bottom Navigation	Drawer Navigation
Best for	Primary destinations	Secondary destinations
Visibility	Always visible	Hidden until swiped open
Number of items	3-5 options	Can contain many items
Usage	Frequent navigation	Less frequent navigation
Example Apps	Instagram, YouTube	Gmail, Google Drive

📌 **Guideline:**

- **Use Bottom Navigation for quick access to key features.**
- **Use Drawer Navigation for secondary options or settings.**
- **You can combine both in the same app (e.g., Gmail).**

Exercise: Test Your Understanding

🔹 **What is the primary difference between Bottom Navigation and Drawer Navigation?**

🔹 **How does NavigationView help in creating a Drawer Menu?**

🔹 **Write a basic implementation of Bottom Navigation with 3 menu items.**

🔹 **When should you use both Bottom Navigation and Drawer Navigation together?**

- **How can you animate the opening and closing of a Navigation Drawer?**

Conclusion

✅ **Bottom Navigation provides quick access to core app destinations.**
✅ **Drawer Navigation is useful for secondary destinations and settings.**
✅ **Both navigation methods enhance user experience when used correctly.**
✅ **Understanding when and how to use these navigation components improves app usability and design.**

CREATING CUSTOM ALERTS, SNACKBARS & DIALOG FRAGMENTS IN ANDROID

CHAPTER 1: INTRODUCTION TO CUSTOM ALERTS, SNACKBARS, AND DIALOG FRAGMENTS

1.1 Why Use Custom Alerts & Dialogs?

🔹 **Alerts and dialogs** improve user interaction by providing **important messages, confirmations, or actions** without navigating away from the screen.

🔹 They are used for:

✅ **User confirmations** (e.g., "Are you sure you want to delete this item?")

✅ **Errors & warnings** (e.g., "Incorrect password. Try again.")

✅ **Information messages** (e.g., "Update available!")

📌 **Example:**

- **A banking app shows a confirmation dialog before processing a transaction.**

CHAPTER 2: CREATING CUSTOM ALERT DIALOGS IN ANDROID

2.1 What is an AlertDialog?

🔹 **AlertDialog** is a built-in Android UI component for displaying alerts, confirmations, or choices to users.

🔹 It includes:

✓ **Title** – Describes the purpose of the alert.

✓ **Message** – The main content of the alert.

✓ **Buttons** – To accept or dismiss the dialog.

2.2 Creating a Simple AlertDialog in Android

✅ Step 1: Show an AlertDialog with OK & Cancel Buttons

📌 Java Implementation

```
new AlertDialog.Builder(this)
   .setTitle("Exit App")
   .setMessage("Are you sure you want to exit?")
   .setPositiveButton("Yes", (dialog, which) -> {
      finish(); // Close the app
   })
   .setNegativeButton("No", (dialog, which) -> dialog.dismiss())
   .show();
```

📌 Kotlin Implementation

```
AlertDialog.Builder(this)
   .setTitle("Exit App")
   .setMessage("Are you sure you want to exit?")
   .setPositiveButton("Yes") { _, _ ->
      finish() // Close the app
   }
   .setNegativeButton("No") { dialog, _ ->
      dialog.dismiss()
   }
```

.show()

📌 **Example:**

- **A logout confirmation alert appears when a user clicks the logout button.**

2.3 Creating a Custom Alert Dialog with a Custom Layout

✅ **Step 1: Create a Custom Layout for the AlertDialog**

📌 **res/layout/custom_alert_dialog.xml**

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"

  android:layout_width="wrap_content"

  android:layout_height="wrap_content"

  android:orientation="vertical"

  android:padding="20dp">


  <TextView

    android:id="@+id/customMessage"

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="Enter your name"

    android:textSize="16sp"

    android:paddingBottom="10dp" />
```

```
<EditText
    android:id="@+id/inputName"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Your Name" />
</LinearLayout>
```

✅ Step 2: Show Custom AlertDialog in Java/Kotlin

📌 Java Implementation

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
View view =
getLayoutInflater().inflate(R.layout.custom_alert_dialog, null);
builder.setView(view);

EditText inputName = view.findViewById(R.id.inputName);
builder.setPositiveButton("Submit", (dialog, which) -> {
  String name = inputName.getText().toString();
  Toast.makeText(this, "Hello, " + name,
Toast.LENGTH_SHORT).show();
});

builder.setNegativeButton("Cancel", (dialog, which) ->
dialog.dismiss());
```

```
builder.show();
```

📌 **Kotlin Implementation**

```
val builder = AlertDialog.Builder(this)
val view = layoutInflater.inflate(R.layout.custom_alert_dialog, null)
builder.setView(view)

val inputName = view.findViewById<EditText>(R.id.inputName)

builder.setPositiveButton("Submit") { _, _ ->
   val name = inputName.text.toString()
   Toast.makeText(this, "Hello, $name",
Toast.LENGTH_SHORT).show()
}

builder.setNegativeButton("Cancel") { dialog, _ -> dialog.dismiss() }
builder.show()
```

📌 **Example:**

- **A custom login dialog prompting the user to enter their name.**

CHAPTER 3: USING SNACKBARS FOR USER NOTIFICATIONS

3.1 What is a Snackbar?

🔹 **Snackbars** provide brief messages at the bottom of the screen **with an optional action button**.

🔹 They automatically disappear after a few seconds, making them **less intrusive than Toast messages**.

✅ **Why Use Snackbars?**

✓ **Better than Toasts** – Can have actions (e.g., "Undo").

✓ **Auto-dismiss feature** – Disappears after a set time.

✓ **Customizable UI** – Can change colors, styles, and add icons.

📌 **Example:**

- **When an item is deleted from a list, a Snackbar shows a message with an "Undo" button.**

3.2 Implementing a Simple Snackbar

📌 **Java Implementation**

```
Snackbar.make(findViewById(android.R.id.content), "Item deleted", Snackbar.LENGTH_LONG).show();
```

📌 **Kotlin Implementation**

```
Snackbar.make(findViewById(android.R.id.content), "Item deleted", Snackbar.LENGTH_LONG).show()
```

3.3 Adding an Action Button to a Snackbar

📌 **Java Implementation**

```
Snackbar.make(findViewById(android.R.id.content), "Item deleted", Snackbar.LENGTH_LONG)
    .setAction("Undo", v -> {
```

```
    Toast.makeText(getApplicationContext(), "Undo action", Toast.LENGTH_SHORT).show();
  })
  .show();
```

📌 **Kotlin Implementation**

```
Snackbar.make(findViewById(android.R.id.content), "Item deleted", Snackbar.LENGTH_LONG)
  .setAction("Undo") {
    Toast.makeText(applicationContext, "Undo action", Toast.LENGTH_SHORT).show()
  }
  .show()
```

📌 **Example:**

- **An "Undo" button appears in a Snackbar when an item is deleted from a list.**

CHAPTER 4: USING DIALOG FRAGMENTS IN ANDROID

4.1 What is a Dialog Fragment?

🔹 **Dialog Fragments** allow **modular and reusable dialogs** that persist across configuration changes.

🔹 Unlike AlertDialog, **DialogFragments are lifecycle-aware** and provide better flexibility.

✅ **Why Use Dialog Fragments?**

✓ **More reusable than AlertDialogs**

✓ **Persists across screen rotations**

✓ **Supports complex UI elements**

4.2 Creating a Dialog Fragment

✅ Step 1: Create a New Java/Kotlin Class for the DialogFragment

📌 Java Implementation

```
public class MyDialogFragment extends DialogFragment {
   @NonNull
   @Override
   public Dialog onCreateDialog(Bundle savedInstanceState) {
      AlertDialog.Builder builder = new
AlertDialog.Builder(getActivity());
      builder.setTitle("Dialog Fragment")
         .setMessage("This is a Dialog Fragment example.")
         .setPositiveButton("OK", (dialog, which) -> dismiss());
      return builder.create();
   }
}
```

📌 Kotlin Implementation

```
class MyDialogFragment : DialogFragment() {
   override fun onCreateDialog(savedInstanceState: Bundle?): Dialog
{
```

```
        val builder = AlertDialog.Builder(requireActivity())
        builder.setTitle("Dialog Fragment")
            .setMessage("This is a Dialog Fragment example.")
            .setPositiveButton("OK") { _, _ -> dismiss() }
        return builder.create()
    }
}
```

✅ **Step 2: Show the DialogFragment**

```
new MyDialogFragment().show(getSupportFragmentManager(), "DialogFragment");
```

```
MyDialogFragment().show(supportFragmentManager, "DialogFragment")
```

📌 **Example:**

- **A date picker or time picker can be implemented using DialogFragment.**

Exercise: Test Your Understanding

🔹 **What is the difference between an AlertDialog and a DialogFragment?**

🔹 **How can you add a custom layout to an AlertDialog?**

🔹 **What is the advantage of using Snackbars over Toast messages?**

🔹 **How do you persist a DialogFragment across screen rotations?**

Conclusion

✅ **AlertDialogs are used for confirmations, warnings, and user inputs.**

✅ **Snackbars are ideal for quick user feedback with actions.**

✅ **DialogFragments offer better lifecycle management for complex dialogs.**

Deep Linking & Navigation Graph in Android Development

Chapter 1: Introduction to Deep Linking & Navigation Graph

1.1 What is Deep Linking?

🔹 **Deep Linking** allows users to **navigate directly to a specific screen within an app** using a URL.

🔹 It enables seamless user experience by integrating web links with in-app navigation.

✅ **Types of Deep Linking:**

✓ **Traditional Deep Linking** – Opens a specific app screen if the app is installed.

✓ **Deferred Deep Linking** – Works even if the app is not installed (redirects to Play Store first).

✓ **App Links (Android App Links)** – Secure deep links that open directly in the app if installed.

📌 **Example:**

- **A user clicks a promo link (e.g., https://example.com/discount) and is taken directly to the discount page in the app.**

1.2 What is Navigation Graph?

🔹 **Navigation Graph** is an XML-based system that defines **how different screens (fragments) in an app are connected**.

🔹 It is part of **Jetpack Navigation Component** and simplifies navigation in multi-screen apps.

✅ **Key Features:**

✓ **Centralized Navigation** – Defines all app destinations in a single XML file.

✓ **Simplified Fragment Transactions** – Reduces manual fragment management.

✓ **Deep Linking Support** – Integrates with deep links easily.

📌 **Example:**

- **A shopping app uses Navigation Graph to navigate from the product list to the product details page.**

CHAPTER 2: IMPLEMENTING DEEP LINKING IN ANDROID

2.1 Adding Deep Links to an Activity

✅ **Step 1: Define Deep Link in Manifest**

```
<activity android:name=".DiscountActivity">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category
android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="https" android:host="example.com"
android:path="/discount" />
  </intent-filter>
</activity>
```

✅ **Step 2: Handle Deep Link in the Activity**

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_discount);

   Uri data = getIntent().getData();
   if (data != null) {
      String promoCode = data.getQueryParameter("code");
      Toast.makeText(this, "Promo Code: " + promoCode,
Toast.LENGTH_LONG).show();
   }
}
```

📌 **Example:**

- **Clicking https://example.com/discount?code=SUMMER50 opens DiscountActivity and displays the promo code.**

2.2 Adding Deep Links Using Navigation Component

✅ **Step 1: Define Deep Link in Navigation Graph**

📌 **res/navigation/nav_graph.xml**

```
<fragment
   android:id="@+id/discountFragment"
   android:name="com.example.app.DiscountFragment"
   android:label="Discount">
   <deepLink app:uri="https://example.com/discount" />
```

```
</fragment>
```

✅ **Step 2: Handle Deep Link in MainActivity**

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Intent intent = getIntent();
    Uri data = intent.getData();
    if (data != null) {
        NavController navController =
Navigation.findNavController(this, R.id.nav_host_fragment);
        navController.handleDeepLink(intent);
    }
}
```

📌 **Example:**

- **A user clicks https://example.com/discount, and the app navigates to the DiscountFragment using Navigation Graph.**

CHAPTER 3: ANDROID APP LINKS (ADVANCED DEEP LINKING)

3.1 What are Android App Links?

🔹 **Android App Links** are deep links that **open directly in the app if installed, without asking the user**.

🔹 If the app is **not installed**, the link opens in a browser.

✅ **Benefits of Android App Links:**

✓ **Secure** – Only verified domains can open the app.

✓ **Seamless user experience** – No prompt asking users to choose a browser or the app.

✓ **Indexed by Google** – Enhances SEO ranking.

3.2 Implementing Android App Links

✅ **Step 1: Modify Manifest to Support App Links**

```
<intent-filter android:autoVerify="true">
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  <data android:scheme="https" android:host="example.com" android:pathPrefix="/product" />
</intent-filter>
```

✅ **Step 2: Verify the Domain with a digital_asset_links.json File**

📌 **Upload the following JSON file to:**

👉 https://example.com/.well-known/assetlinks.json

```
[
 {
  "relation": ["delegate_permission/common.handle_all_urls"],
```

```
  "target": {
    "namespace": "android_app",
    "package_name": "com.example.app",
    "sha256_cert_fingerprints": ["AB:CD:EF:12:34:..."]
  }
 }
]
```

📌 **Example:**

- **Clicking https://example.com/product/123 will directly open the product details in the app.**

CHAPTER 4: IMPLEMENTING NAVIGATION GRAPH IN ANDROID

4.1 What is a Navigation Graph?

🔹 A **Navigation Graph** is an **XML resource file** that defines the structure of app navigation.

🔹 It contains **destinations (screens), actions (transitions), and deep link support**.

✅ **Benefits of Navigation Graph:**

✓ **Removes manual FragmentTransaction code.**

✓ **Supports Safe Args for passing data between fragments.**

✓ **Built-in deep link support.**

4.2 Setting Up a Navigation Graph

✅ **Step 1: Add Dependencies**

📌 Add in build.gradle (Module: app)

```
dependencies {
    implementation "androidx.navigation:navigation-fragment-ktx:2.5.3"
    implementation "androidx.navigation:navigation-ui-ktx:2.5.3"
}
```

✅ Step 2: Create a Navigation Graph

📌 res/navigation/nav_graph.xml

```
<?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"
    app:startDestination="@id/homeFragment">

    <fragment
        android:id="@+id/homeFragment"
        android:name="com.example.app.HomeFragment"
        android:label="Home"/>

    <fragment
        android:id="@+id/detailsFragment"
        android:name="com.example.app.DetailsFragment"
```

```
    android:label="Details">
    <argument
      android:name="itemId"
      app:argType="string"/>
  </fragment>

  <action
    android:id="@+id/action_home_to_details"
    app:destination="@id/detailsFragment"
    app:enterAnim="@anim/slide_in_right"
    app:exitAnim="@anim/slide_out_left"/>
</navigation>
```

✅ **Step 3: Navigate Between Fragments**

📌 **Navigate from HomeFragment to DetailsFragment**

```
NavController navController = Navigation.findNavController(view);
navController.navigate(R.id.action_home_to_details);
```

📌 **Example:**

- **A user clicks on an item in HomeFragment, and the app navigates to DetailsFragment with an animation.**

📌 **Exercise: Test Your Understanding**

🔹 **What is the difference between traditional deep linking and deferred deep linking?**

- 🔹 How does Navigation Graph simplify fragment transactions?
- 🔹 What is the purpose of digital_asset_links.json in Android App Links?
- 🔹 Write a deep link intent filter for a profile page (https://example.com/profile).
- 🔹 Implement a deep link using Navigation Graph in an Android app.

📌 Conclusion

✅ Deep Linking allows users to navigate directly to app screens from URLs.
✅ Navigation Graph simplifies fragment navigation in multi-screen apps.
✅ Android App Links provide a secure and seamless deep linking experience.
✅ Understanding deep linking and navigation helps create smooth and user-friendly Android apps.

ASSIGNMENT: BUILD A MULTI-SCREEN APP WITH FRAGMENTS AND NAVIGATION COMPONENTS.

📌 Step-by-Step Assignment Solution: Build a Multi-Screen App with Fragments and Navigation Components

In this assignment, we will build a **multi-screen Android app** using **Fragments and Navigation Components**. The app will have:

✅ **Three Fragments** (Home, Profile, Settings).

✅ **Navigation Component** for smooth navigation between screens.

✅ **Bottom Navigation Bar** for quick access to fragments.

📌 Step 1: Create a New Android Project

1️⃣ **Open Android Studio** and create a new project.

2️⃣ Choose **Empty Activity** and click **Next**.

3️⃣ Set the project name as **MultiScreenApp**.

4️⃣ Select **Kotlin** as the language.

5️⃣ Click **Finish** to create the project.

📌 Step 2: Add Dependencies for Navigation Component

1️⃣ Open **build.gradle (Module: app)** and add the following dependencies:

```
dependencies {

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

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

}

2️⃣ Click **Sync Now** to apply the changes.

📌 Step 3: Create Fragments for the Multi-Screen App

🔹 **We will create three fragments: HomeFragment, ProfileFragment, and SettingsFragment.**

✅ Step 3.1: Create HomeFragment

1️⃣ **Right-click on com.example.multiscreenapp > New > Fragment > Fragment (Blank).**
2️⃣ Name the fragment **HomeFragment** and click **Finish**.
3️⃣ Modify HomeFragment.kt:

```
class HomeFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_home, container, false)
    }
}
```

✅ Step 3.2: Create ProfileFragment & SettingsFragment

Repeat the same steps for **ProfileFragment** and **SettingsFragment**.

📌 ProfileFragment.kt:

```
class ProfileFragment : Fragment() {
```

```
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_profile, container,
false)
    }
}
```

📌 **SettingsFragment.kt:**

```
class SettingsFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_settings, container,
false)
    }
}
```

📌 **Step 4: Create a Navigation Graph**

🔹 The **Navigation Graph** manages navigation between fragments.

✅ **Step 4.1: Add a Navigation Graph**

1️⃣ **Go to res > Right-click on res > New > Android Resource File.**

2️⃣ Name it **nav_graph,** select **Navigation,** and click **OK**.

✅ Step 4.2: Add Fragments to nav_graph.xml

📌 res/navigation/nav_graph.xml:

```
<?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"
    app:startDestination="@id/homeFragment">

    <fragment
        android:id="@+id/homeFragment"
        android:name="com.example.multiscreenapp.HomeFragment"
        android:label="Home"
        tools:layout="@layout/fragment_home" />

    <fragment
        android:id="@+id/profileFragment"
        android:name="com.example.multiscreenapp.ProfileFragment"
        android:label="Profile"
        tools:layout="@layout/fragment_profile" />
```

```
    <fragment
        android:id="@+id/settingsFragment"
android:name="com.example.multiscreenapp.SettingsFragment"
        android:label="Settings"
        tools:layout="@layout/fragment_settings" />
</navigation>
```

📌 Step 5: Set Up Navigation Host in MainActivity

🔹 The **NavHostFragment** is the container that swaps between fragments dynamically.

✅ Modify activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
android:name="androidx.navigation.fragment.NavHostFragment"
```

```
    android:layout_width="match_parent"
    android:layout_height="odp"
    android:layout_marginBottom="56dp"
    app:defaultNavHost="true"
    app:navGraph="@navigation/nav_graph"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"/>


<com.google.android.material.bottomnavigation.BottomNavigatio
nView
    android:id="@+id/bottom_nav"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?android:attr/windowBackground"
    app:menu="@menu/bottom_nav_menu"
    app:layout_constraintBottom_toBottomOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

📌 Step 6: Create a Bottom Navigation Menu

✅ Step 6.1: Add a menu resource file

1️⃣ **Right-click on res > New > Android Resource Directory.**
2️⃣ Select **Resource type: menu,** and name it **menu**.

5. Right-click on menu > New > Menu Resource File > Name it **bottom_nav_menu.xml**.

✅ Step 6.2: Define Bottom Navigation Items

📌 res/menu/bottom_nav_menu.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">

  <item
    android:id="@+id/homeFragment"
    android:icon="@drawable/ic_home"
    android:title="Home"
    app:showAsAction="ifRoom" />

  <item
    android:id="@+id/profileFragment"
    android:icon="@drawable/ic_profile"
    android:title="Profile"
    app:showAsAction="ifRoom" />

  <item
    android:id="@+id/settingsFragment"
```

```
    android:icon="@drawable/ic_settings"
    android:title="Settings"
    app:showAsAction="ifRoom" />

</menu>
```

📌 Step 7: Connect Bottom Navigation with Navigation Component

✅ Modify MainActivity.kt

```
class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val navController = findNavController(R.id.nav_host_fragment)
    val bottomNav = findViewById<BottomNavigationView>(R.id.bottom_nav)

    // Set up BottomNavigationView with NavController
    bottomNav.setupWithNavController(navController)
  }
}
```

📌 Step 8: Run and Test the Multi-Screen App

1️⃣ **Run the app on an emulator or real device.**
2️⃣ **Click on Bottom Navigation items** to switch between **Home, Profile, and Settings** fragments.
3️⃣ **Ensure smooth transitions** between fragments using the Navigation Component.

📌 Additional Enhancements

- **Add animations** to fragment transitions.
- **Implement ViewModel** to pass data between fragments.
- **Use Safe Args** for better data passing between fragments.
- **Improve UI with Material Design components.**

📌 Exercise: Test Your Understanding

- **What is the role of NavController in Navigation Component?**
- **How does FragmentContainerView replace fragments dynamically?**
- **Why do we use BottomNavigationView in multi-screen apps?**
- **How can we pass data from one fragment to another?**
- **What are the advantages of using Navigation Component over traditional Fragment Transactions?**

📌 Conclusion

✅ **We successfully built a multi-screen Android app using Fragments and Navigation Components.**
✅ **The app supports smooth navigation with a Bottom**

Navigation Bar.

✅ **We used a Navigation Graph to manage fragment transitions efficiently.**