Experiment No: 1

Aim: Download Install and Configure Android Studio on Windows/ Linux environment.

Theory:

Introduction

Android Studio is the official Integrated Development Environment (IDE) recommended by Google for Android application development. It provides powerful features such as a code editor, code completion, debugging tools, an Android Emulator, and a flexible build system. To begin developing Android applications on a Windows system, Android Studio must be properly installed and configured.

Downloading Android Studio

To get started, the user must download the latest version of Android Studio from the official Android developer website:

- https://developer.android.com/studio

The downloadable .exe file includes Android Studio, Android SDK, Android Virtual Device (AVD), and necessary command-line tools.

Installing Android Studio on Windows

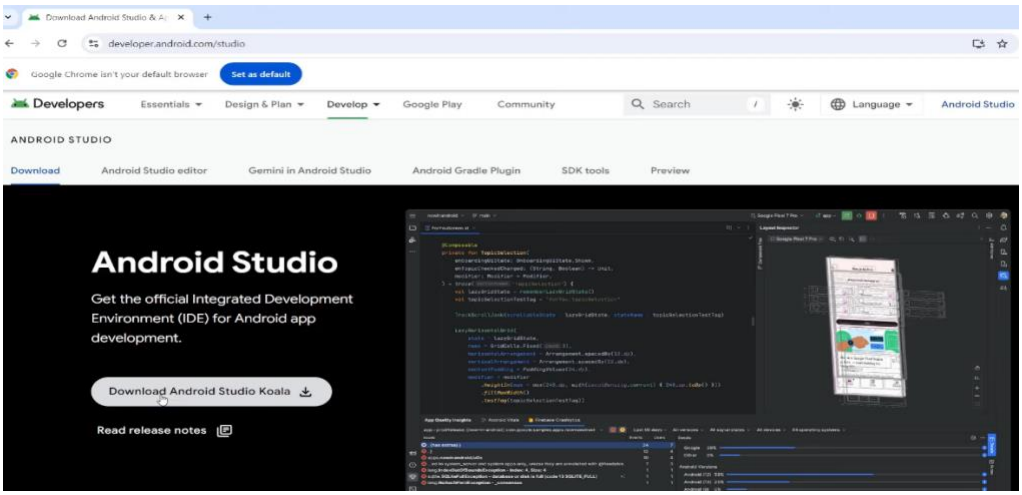
1. **Run the Installer**: Double-click the downloaded .exe file to launch the Android Studio setup wizard.
2. **Follow Setup Wizard**: The wizard guides users through the installation process. It is recommended to choose the **Standard** installation option for beginners.
3. **Install SDK and Tools**: During installation, the wizard automatically downloads and installs the Android SDK, emulator, and essential build tools.
4. **Complete Setup**: Once all components are installed, Android Studio is ready to be launched.

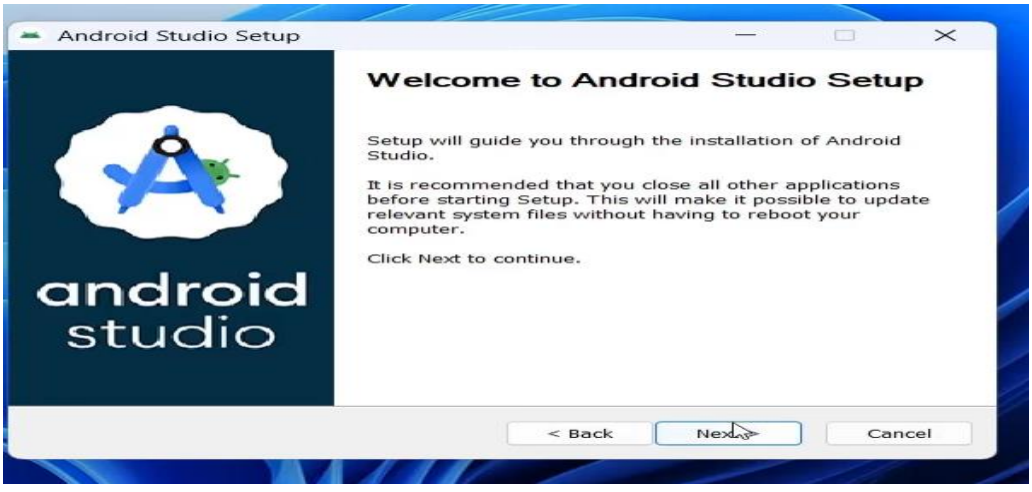
Configuring Android Studio

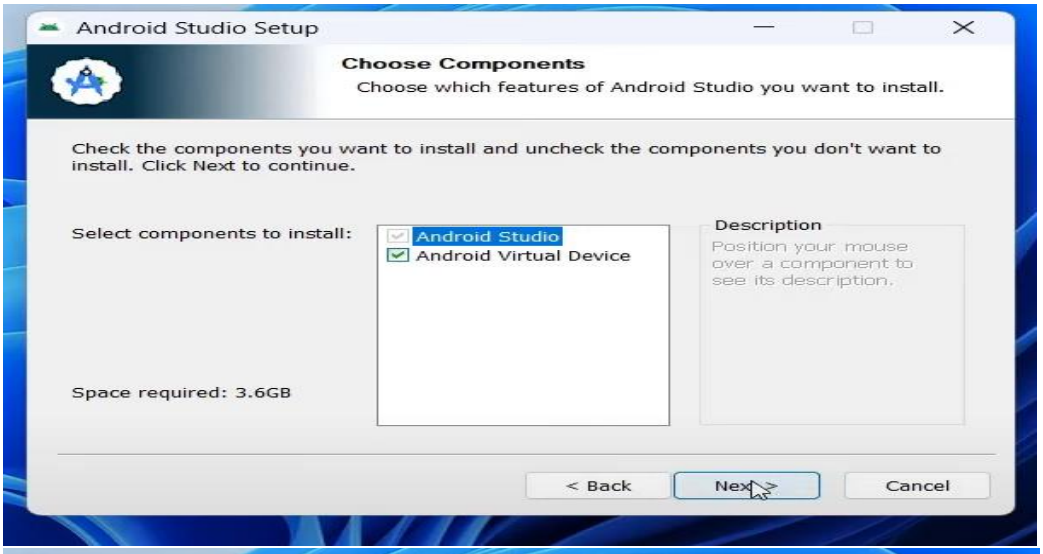
After the installation is complete, Android Studio needs to be configured for the first time:

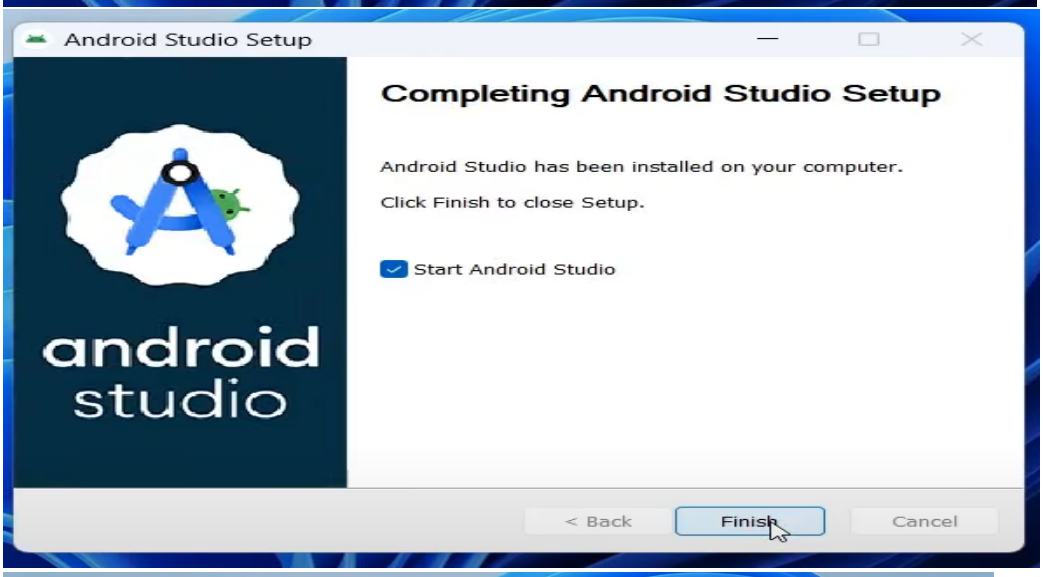
- **SDK Configuration**:
    - Go to **File > Settings > Appearance & Behavior > System Settings > Android SDK**.
    - Select and install the desired SDK platforms and tools.
- **AVD Manager (Optional)**:
    - Tools > Device Manager
    - Create a virtual Android device for testing apps without a physical device.
- **Appearance and Theme**:
    - Customize the theme (light/dark), font size, and layout based on user preferences.

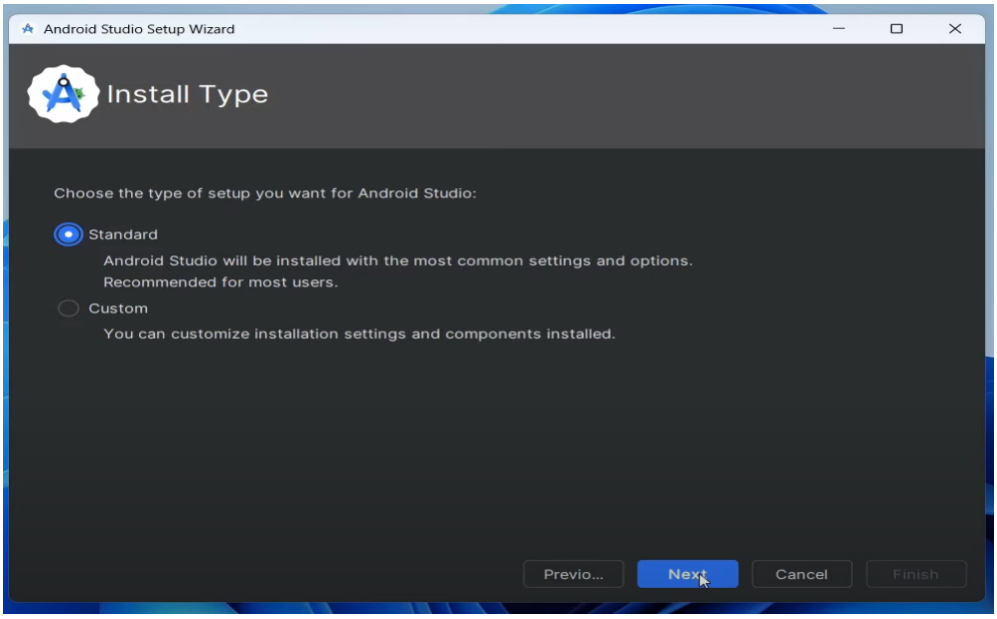
- **Output:**

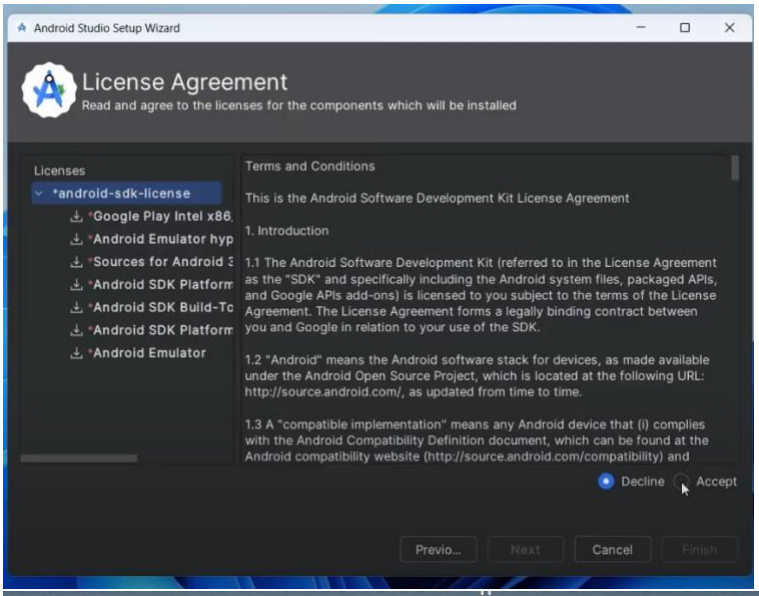


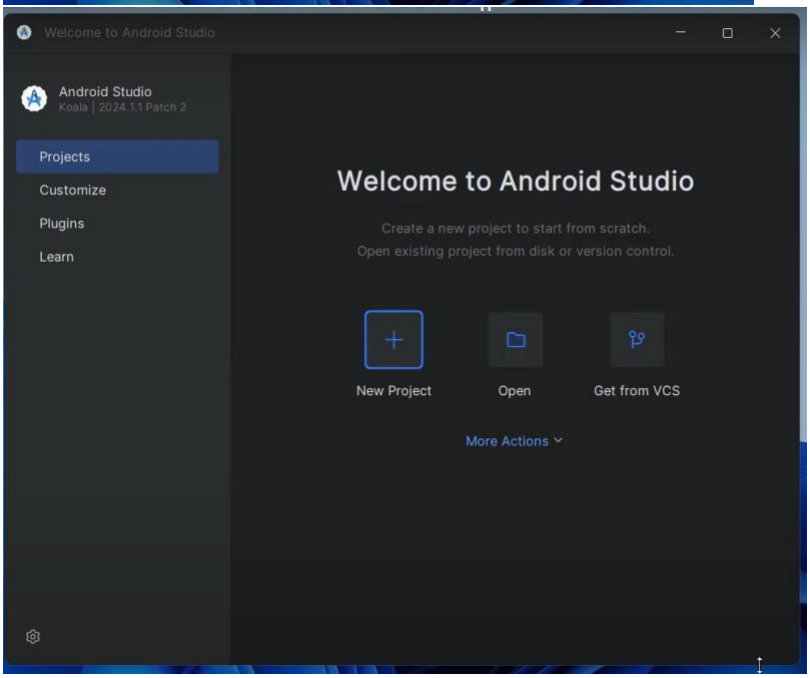



Android Studio Setup
Choose Components
Choose which features of Android Studio you want to install.
Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.
Select components to install:
Android Studio
Android Virtual Device
Description
Position your mouse over a component to see its description.
Space required: 3.6GB
< Back
Next >
Cancel


Android Studio Setup
android studio
Completing Android Studio Setup
Android Studio has been installed on your computer.
Click Finish to close Setup.
Start Android Studio
< Back
Finish
Cancel


Android Studio Setup Wizard
Install Type
Choose the type of setup you want for Android Studio:
Standard
Android Studio will be installed with the most common settings and options.
Recommended for most users.
Custom
You can customize installation settings and components installed.
Previo...
Next
Cancel
Finish





Conclusion

Installing and configuring Android Studio on Windows is a straightforward process, thanks to its user-friendly setup wizard and integrated tools. With Android Studio properly set up, developers can begin building, testing, and deploying Android applications efficiently on a Windows environment.

Experiment No :2

Aim: Building Simple User Interface using UI Widgets, Layouts and Adapters. Use Material Design Pattern.

Theory:

Building Simple User Interface using UI Widgets, Layouts, and Adapters (Material Design Pattern)

In Android application development, a **User Interface (UI)** plays a crucial role in interacting with users. A simple yet efficient UI is typically built using **UI widgets**, **layouts**, and **adapters**, following **Material Design principles** to ensure a consistent, intuitive, and visually appealing experience.

1. UI Widgets

Widgets are the building blocks of an Android UI. They represent interactive UI components that users can see and interact with. Common examples include:

- **TextView**: Displays text to the user.
- **EditText**: Allows the user to enter text.
- **Button**: Triggers an action when clicked.
- **ImageView**: Displays an image.
- **CheckBox, RadioButton, Switch**: Collect user choices or preferences.

Widgets follow Material Design styling using **Material Components** such as Button, OutlinedTextField, Card, etc., available via the androidx.compose.material3 or com.google.android.material libraries.

2. Layouts

Layouts define how widgets are arranged on the screen. Android provides several layout containers:

- **LinearLayout / Column / Row (Compose)**: Arranges widgets linearly, either vertically or horizontally.
- **ConstraintLayout**: Allows complex, responsive UIs by defining constraints between widgets.

- **FrameLayout**: Useful for stacking views on top of each other.
- **Box (Compose)**: Provides flexible overlapping view arrangement in Jetpack Compose.

In Jetpack Compose, layouts are managed using composable functions like Column, Row, Box, with modifiers like padding(), fillMaxSize(), and align() for fine-tuning positioning and appearance.

3. Adapters

Adapters act as a bridge between data sources and UI components like **ListView**, **RecyclerView**, or **LazyColumn** (in Jetpack Compose). They are responsible for:

- Providing data to the UI
- Creating and recycling views for efficient performance
- Binding the data to individual UI elements

For example, in Compose:

kotlin

CopyEdit

```
LazyColumn {
   items(userList) { user ->
      Text("Name: ${user.name}")
   }
}
```

In traditional Android (View system), ArrayAdapter or RecyclerView.Adapter are used.

4. Material Design Pattern

Material Design is a design language developed by Google, aiming for consistency, meaningful motion, and bold visual style. It offers guidelines for:

- **Theming and Typography**: Use of MaterialTheme, consistent fonts, and colors.
- **Components**: Pre-built components like TopAppBar, NavigationDrawer, Snackbar, and FAB.
- **Elevation and Shadows**: To emphasize hierarchy and interactions.
- **Motion and Transitions**: Animations for smoother navigation and feedback.

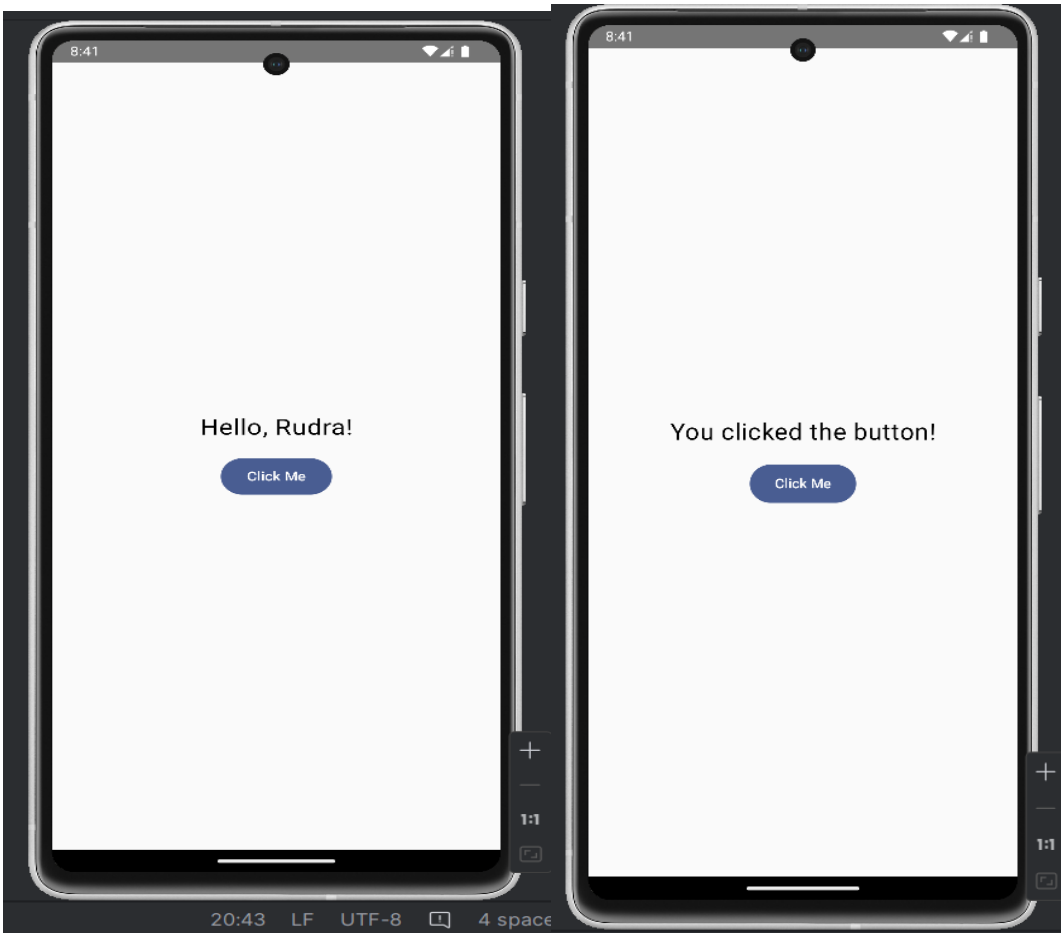
Jetpack Compose provides out-of-the-box support for Material 3 (Material You) with powerful theming and customizable components.

Code:

```
package com.example.sample


import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.*
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.example.sample.ui.theme.SampleTheme

class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContent {
         SampleTheme {
            // Calling our composable function to display the UI
            GreetingScreen() } }}}
```

```
@Composable
fun GreetingScreen() {
    var messageText by remember { mutableStateOf("Hello, User!") }
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        // Text View with the current message
        Text(
            text = messageText,
            color = Color.Black,
            fontSize = 24.sp,
            modifier = Modifier.padding(bottom = 16.dp)
        )
        Button(onClick = { messageText = "You clicked the button!" }) {
            Text(text = "Click Me") }}}
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    SampleTheme {
        GreetingScreen()
    }
}
```

Output:



Conclusion:

Building a simple user interface in Android involves combining widgets, layouts, and adapters efficiently, and enhancing them with Material Design principles. This ensures that the UI is not only functional but also visually consistent, accessible, and user-friendly across all devices and screen sizes.

Experiment No: 3

Aim: Develop an application having animation on views.

Theory:

Introduction to View Animations

Animations enhance user experience by making interactions more intuitive and engaging. In Android, view animations are used to transition between UI states smoothly. With Jetpack Compose, animations are declarative, concise, and integrated into the UI framework.

🧩 Types of Animations in Jetpack Compose

1. **State-based animations:**
    - animate*AsState functions (e.g., animateDpAsState, animateColorAsState) allow smooth transitions between state values.
2. **Visibility transitions:**
    - AnimatedVisibility is used to animate the appearance or disappearance of a composable with fadeIn, fadeOut, expandIn, and shrinkOut.
3. **Infinite animations:**
    - infiniteTransition creates continuously running animations (e.g., loading indicators).
4. **Custom transitions:**
    - updateTransition is used when multiple properties depend on the same state change.

Components Used

- **Box** or **Surface**: For displaying and animating background color and size.
- **Buttons**: For triggering animations.
- **State variables (remember)**: To track UI state changes (e.g., expanded or visible).
- **Modifiers**: For applying animated properties like size, background, or visibility.

Material Design and Animation

Material Design recommends the use of **motion** to:

- Express changes in state clearly.
- Provide feedback on user interactions.
- Guide user attention.

Jetpack Compose follows Material 3 principles, offering built-in APIs that automatically adopt Material transitions and responsiveness.

Practical Example

In a typical app:

- A user taps a button.
- A card expands with animated size and color.
- Another button toggles visibility using fade and scale animations.

These interactions reflect real-world physics and enhance user satisfaction.

Benefits of Using Compose for Animation

- **Declarative syntax** makes animations easier to write and maintain.

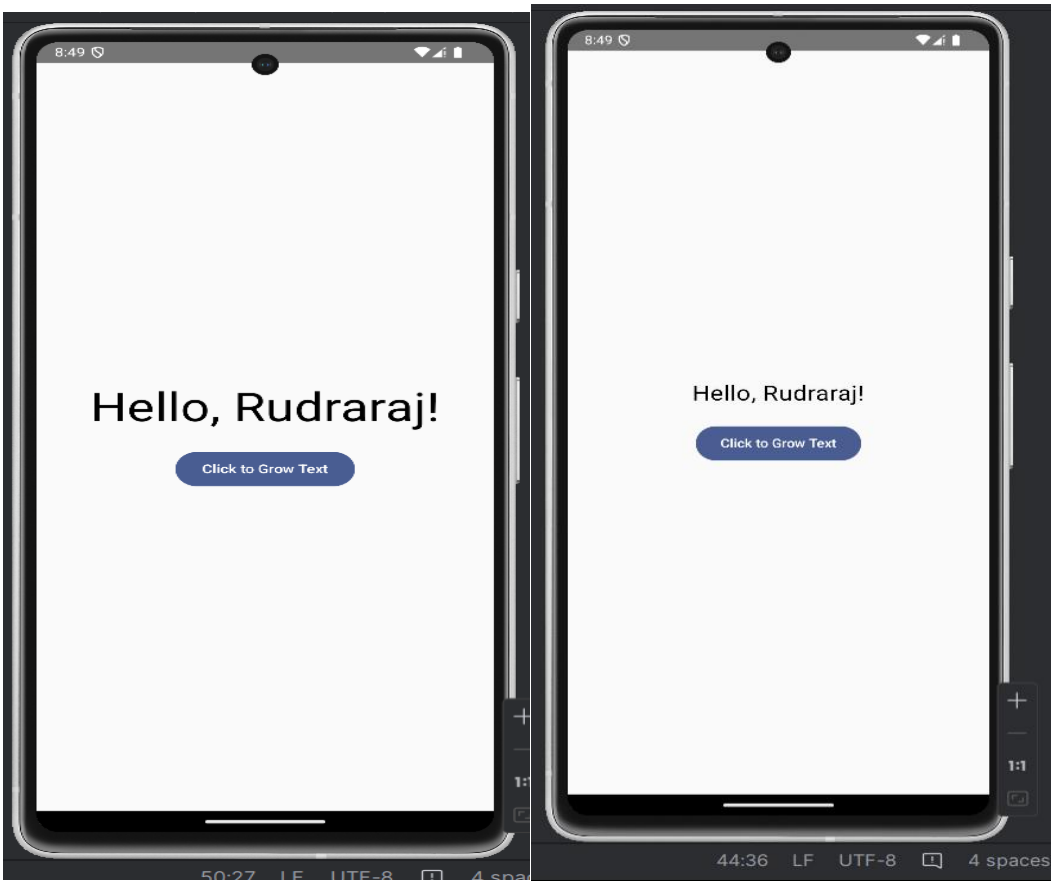
- **Less boilerplate** compared to XML + Animator APIs.
- **Smooth and performant**, especially on modern Android devices.
- **Composable integration** allows animation logic to stay close to UI code

Code:

```
package com.example.sample


import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.*
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.example.sample.ui.theme.SampleTheme


class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState); setContent { SampleTheme {
GreetingScreen() } } }
```

```
}

@Composable
fun GreetingScreen() {
    var messageText by remember { mutableStateOf("Hello, Rudra!") }
    Column(
        modifier = Modifier.fillMaxSize().padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(text = messageText, color = Color.Black, fontSize = 24.sp, modifier =
Modifier.padding(bottom = 16.dp))
        Button(onClick = { messageText = "You clicked the button!" }) { Text("Click
Me") }
    }
}

@Preview(showBackground = true)
@Composable
fun DefaultPreview() { SampleTheme { GreetingScreen()
}
 }
```

Output:



Conclusion:

Animating views in Android using Jetpack Compose enables the creation of modern, interactive UIs with minimal effort. Leveraging tools like animate*AsState and AnimatedVisibility, developers can enrich the user interface while adhering to Material Design guidelines, ensuring a consistent and engaging user experience.

Experiment No :4

Aim: Develop an Android App in which a user can register. After registration user can login with the credentials supplied for registration using Firebase.

Theory:

Android User Registration and Login using Firebase

Introduction

Developing an Android app with user registration and login functionality is a common requirement in modern mobile applications. By integrating **Firebase Authentication**, developers can manage user credentials securely without building custom back-end infrastructure. Firebase supports authentication using email/password, phone number, and third-party providers like Google, Facebook, etc.

Firebase Overview

Firebase is a mobile and web application development platform developed by Google. It provides backend services such as:

- **Authentication**
- **Realtime Database**
- **Cloud Firestore**
- **Cloud Storage**
- **Hosting**

Firebase Authentication specifically allows developers to manage users, handle secure sign-ins, and track sessions.

Components Involved

- **Android Studio**: IDE used to develop Android applications.

- **Firebase Console**: Web interface to configure Firebase services.
- **Firebase Authentication**: Handles user registration and login.
- **Email/Password Provider**: Used for simple authentication in this app.

Functional Requirements

- **User Registration Page**: Allows users to create a new account using email and password.
- **Login Page**: Enables users to login using previously registered credentials.
- **Authentication Logic**: Uses Firebase Auth SDK to validate credentials.
- **Navigation**: After successful login, user is redirected to the main activity/dashboard.

Working Process

1. **User Registration**
    a. User enters email and password.
    b. App calls `FirebaseAuth.createUserWithEmailAndPassword()`.
    c. On success, Firebase creates a user and stores it in the authentication database.
    d. On failure, appropriate error messages (e.g., weak password, email already in use) are shown.
2. **User Login**
    a. User enters email and password.
    b. App calls `FirebaseAuth.signInWithEmailAndPassword()`.
    c. Firebase validates the credentials.
    d. If valid, user is logged in and navigated to the home screen.
    e. If invalid, error is shown (e.g., user not found, wrong password).
3. **Session Handling**
    a. Firebase maintains user sessions.
    b. `FirebaseAuth.getCurrentUser()` can be used to check if a user is logged in.

Advantages of Using Firebase Authentication

- **Security**: Google-backed authentication system.
- **Simplicity**: Quick integration with Firebase SDK.
- **Scalability**: Handles large number of users.
- **Cross-Platform Support**: Can be used in iOS, Android, and web apps.

Security Considerations

- Firebase uses secure HTTPS communication.
- Passwords are stored securely using hashing.
- Developers should validate inputs on both client and server side.
- It is recommended to use **Firebase Realtime Database Rules** or **Firestore Security Rules** to protect data access.

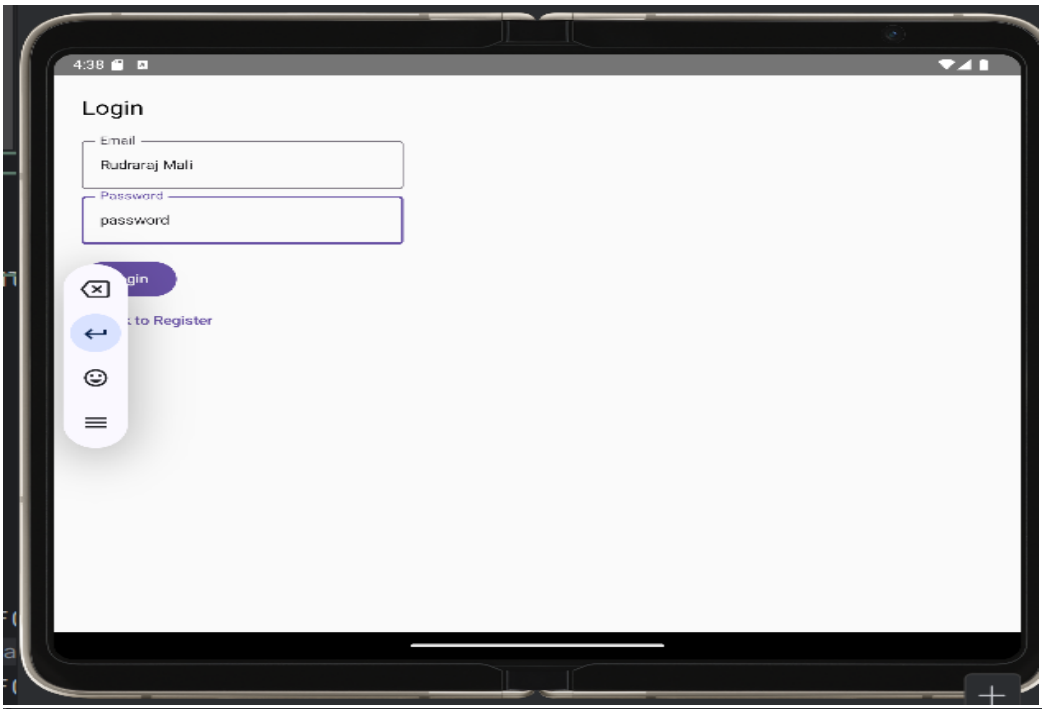
Code:

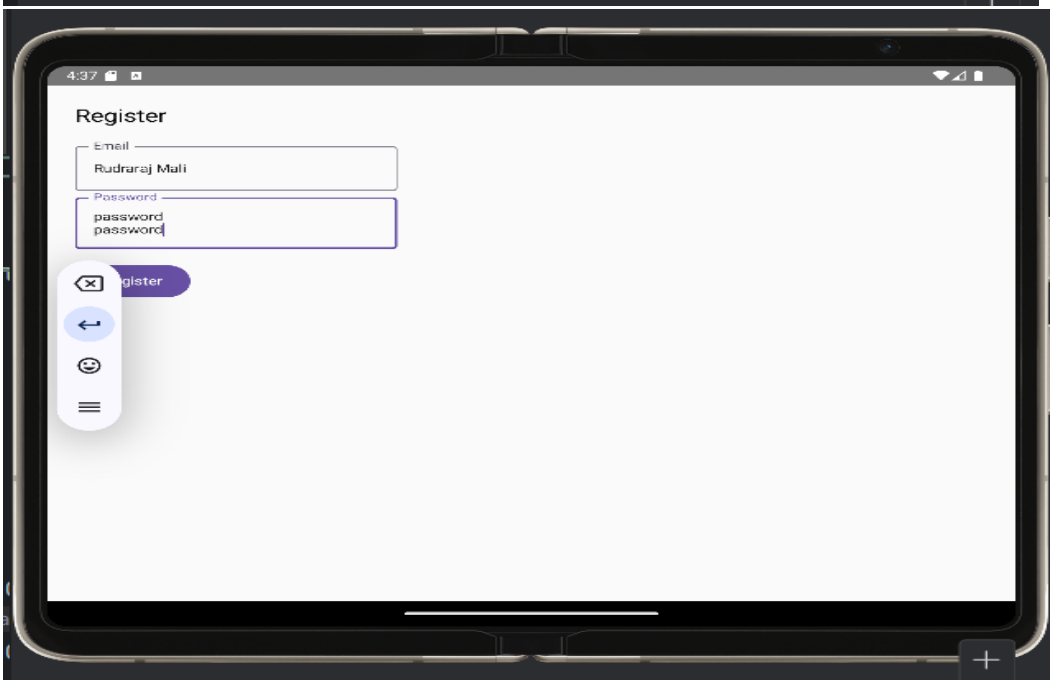
```
package call


import android.os.Bundle

import android.widget.Toast

import androidx.activity.ComponentActivity

import androidx.activity.compose.setContent

import androidx.compose.foundation.layout.*

import androidx.compose.material3.*

import androidx.compose.runtime.*

import androidx.compose.ui.Modifier

import androidx.compose.ui.platform.LocalContext

import androidx.compose.ui.unit.dp


class MainActivity : ComponentActivity() {

   override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState);
setContent { SimpleAuthApp() } }

}
```

```
@Composable
fun SimpleAuthApp() {
    var isLoginScreen by remember { mutableStateOf(false) }
    var savedEmail by remember { mutableStateOf("") }
    var savedPassword by remember { mutableStateOf("") }
    if (isLoginScreen) {
        LoginScreen(
            onLogin = { email, password, context ->
                if (email == savedEmail && password == savedPassword)
                    Toast.makeText(context, "Login successful", Toast.LENGTH_SHORT).show()
                else
                    Toast.makeText(context, "Login failed", Toast.LENGTH_SHORT).show()
            },
            onBack = { isLoginScreen = false }
        )
    } else {
        RegisterScreen(
            onRegister = { email, password, context ->
                savedEmail = email; savedPassword = password
                Toast.makeText(context, "Registered successfully", Toast.LENGTH_SHORT).show()
                isLoginScreen = true
            }
        )
    }
}

@Composable
fun RegisterScreen(onRegister: (String, String, android.content.Context) -> Unit) {
```

```
    val context = LocalContext.current
    var email by remember { mutableStateOf("") }
    var password by remember { mutableStateOf("") }
    Column(Modifier.padding(24.dp)) {
        Text("Register", style = MaterialTheme.typography.titleLarge)
        Spacer(Modifier.height(16.dp))
        OutlinedTextField(email, { email = it }, label = { Text("Email") })
        OutlinedTextField(password, { password = it }, label = { Text("Password") })
        Spacer(Modifier.height(16.dp))
        Button(onClick = { onRegister(email, password, context) }) { Text("Register") }
    }
}

@Composable
fun LoginScreen(onLogin: (String, String, android.content.Context) -> Unit, onBack: () -> Unit)
{
    val context = LocalContext.current
    var email by remember { mutableStateOf("") }
    var password by remember { mutableStateOf("") }
    Column(Modifier.padding(24.dp)) {
        Text("Login", style = MaterialTheme.typography.titleLarge)
        Spacer(Modifier.height(16.dp))
        OutlinedTextField(email, { email = it }, label = { Text("Email") })
        OutlinedTextField(password, { password = it }, label = { Text("Password") })
        Spacer(Modifier.height(16.dp))
        Button(onClick = { onLogin(email, password, context) }) { Text("Login") }
        TextButton(onClick = onBack) { Text("Back to Register") }
    }
}
```

Output:


4:38
Login
Email
Rudraraj Mali
Password
password
to Register


4:37
Register
Email
Rudraraj Mali
Password
password
password

Conclusion: Firebase Authentication provides a robust and secure method for managing user login and registration in Android applications. By leveraging Firebase, developers can focus more on app functionality and user experience without worrying about backend implementation and security complexities.

Experiment No:5

Aim: Develop an Android App to your college display a Navigation Drawer with Menus like About Us, Departments, Student Section, Contact Us, etc.

Theory:

Developing a College Android App with Navigation Drawer

Introduction

An Android app for a college can serve as a centralized information platform for students, faculty, and visitors. One of the most user-friendly interfaces for organizing multiple sections in an app is the **Navigation Drawer**. It is a side panel that displays navigation options, allowing users to switch between different fragments or activities easily.

Objective

The main goal is to develop a structured and user-friendly Android application that includes:

- A **Navigation Drawer** interface
- Menu options like **About Us**, **Departments**, **Student Section**, **Contact Us**, etc.
- Fragment-based navigation for a smooth and modular design

Tools and Technologies

- **Android Studio**: Primary development environment
- **Java** or **Kotlin**: Programming language
- **XML**: For UI design
- **Material Design Components**: For modern UI/UX
- **Fragments**: For dynamic and modular content loading within the same activity

Key Components

- **Navigation Drawer Layout**: Implemented using `DrawerLayout`, `NavigationView`, and `Toolbar`
- **Menu Items**: Defined in a `menu.xml` file and inflated inside the `NavigationView`
- **Fragments**: Each menu option loads a different fragment (e.g., AboutUsFragment, DepartmentFragment)
- **Toolbar**: Replaces the ActionBar and syncs with the Drawer for consistent navigation

Functional Workflow

1. **App Launches** and displays the main activity with a Toolbar and Navigation Drawer.
2. User **opens the Drawer** by swiping from the left or clicking the menu icon.
3. Menu items like **About Us**, **Departments**, **Student Section**, **Contact Us**, etc., are displayed.
4. When the user selects a menu item:
    a. The drawer closes.
    b. The corresponding **fragment is loaded** in the main activity.
5. Each fragment displays relevant static or dynamic content.

Advantages of Using Navigation Drawer

- **Improved Usability**: Easy access to multiple sections
- **Efficient Layout**: Keeps UI clean and organized
- **Responsive Design**: Works well on tablets and phones
- **Modular Architecture**: Each section can be developed and maintained independently using fragments

User Interface Design Considerations

- Use **consistent icons** and labels for clarity.
- Apply **Material Design** for modern look and feel.
- Ensure **accessibility and responsiveness** across screen sizes.
- Keep content in each section relevant and easy to navigate.

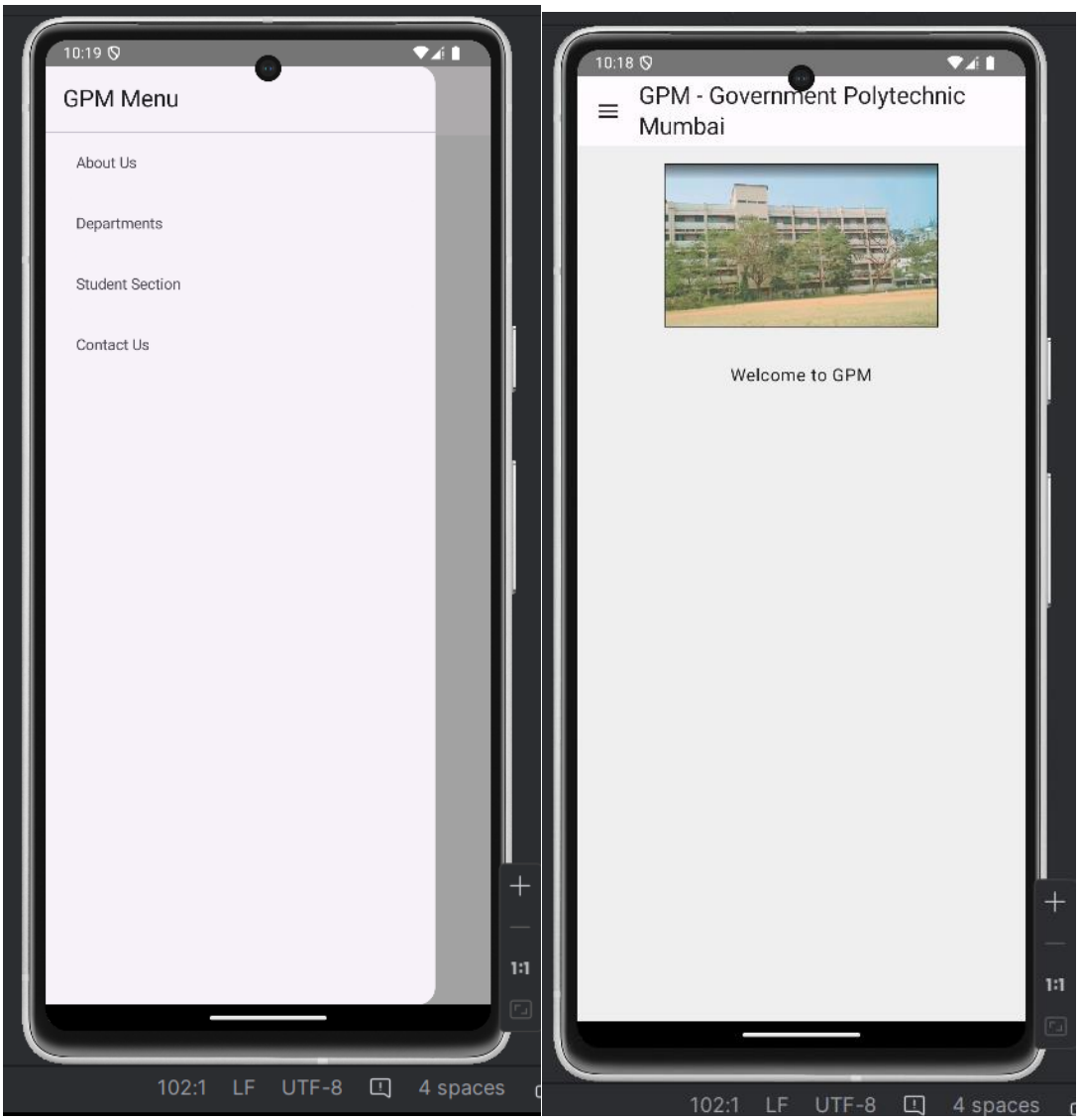
Code:

```
package com.example.sample


import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.Image
import androidx.compose.foundation.layout.*
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Menu
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.unit.dp
import kotlinx.coroutines.launch


class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState);
setContent { GPMApp() } }
}


@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun GPMApp() {
   val drawerState = rememberDrawerState(initialValue = DrawerValue.Closed)
```

```
    val scope = rememberCoroutineScope()
    val menuItems = listOf("About Us", "Departments", "Student Section", "Contact Us")
    var selectedScreen by remember { mutableStateOf("Welcome to GPM") }


    ModalNavigationDrawer(
        drawerState = drawerState,
        drawerContent = {
            ModalDrawerSheet {
                Text("GPM Menu", modifier = Modifier.padding(16.dp), style =
MaterialTheme.typography.titleLarge)
                Divider()
                menuItems.forEach { item ->
                    NavigationDrawerItem(
                        label = { Text(item) },
                        selected = item == selectedScreen,
                        onClick = {
                            selectedScreen = item; scope.launch { drawerState.close() }
                        },
                        modifier = Modifier.padding(NavigationDrawerItemDefaults.ItemPadding)
                    )
                }
            }
        }
    ) {
        Scaffold(
            topBar = {
                TopAppBar(
                    title = { Text("GPM - Government Polytechnic Mumbai") },
                    navigationIcon = {
```

```
            IconButton(onClick = { scope.launch { drawerState.open() } }) {
                Icon(Icons.Filled.Menu, contentDescription = "Menu") }})}
) { padding ->
    Surface(
        modifier = Modifier.padding(padding).fillMaxSize(),
        color = Color(0xFFEFEFEF)
    ) {
        Column(
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Top,
            modifier = Modifier.fillMaxSize()
        ) {
            Image(
                painter = painterResource(id = R.drawable.gpm),
                contentDescription = "GPM Logo",
                modifier = Modifier.padding(16.dp).height(150.dp)
            )
            Text(
                text = selectedScreen,
                modifier = Modifier.padding(16.dp),
                style = MaterialTheme.typography.bodyLarge
            )
        }}} }}
```

Output:



Conclusion:

Using a Navigation Drawer in a college app allows structured access to various informational sections such as About Us, Departments, and Student Section. It enhances user experience by providing a clean, consistent, and organized interface for navigating multiple parts of the app seamlessly.

Experiment No :6

Aim: Design an android based application to display contact list in RecyclerView.

Theory:

Designing an Android App to Display Contact List in RecyclerView

Introduction

Displaying a list of items in an Android app is a common use case, and `RecyclerView` is the recommended widget for displaying large or dynamic datasets efficiently. In this application, we aim to develop a simple Android app that **displays a list of contacts (names, phone numbers, etc.)** using `RecyclerView`.

Objective

- Develop a responsive and efficient contact list interface.
- Use `RecyclerView` for smooth scrolling and memory efficiency.
- Create a custom layout for individual contact items.
- Populate the list either statically or dynamically (e.g., from device contacts or a database).

Tools and Technologies

- **Android Studio**: IDE for Android development.

- **Java** or **Kotlin**: Programming language.
- **RecyclerView**: Android UI widget for efficient list display.
- **ViewHolder Pattern**: To reuse views and improve performance.
- **Custom Adapter**: To bind data to the RecyclerView.

Functional Workflow

1. **Design UI Layouts**:
    a. Main layout includes `RecyclerView`.
    b. Item layout defines how each contact is displayed (e.g., with TextViews for name and number).

2. **Create Contact Model Class**:
    a. Class with fields like `name`, `phoneNumber`, `email`, etc.

3. **Build RecyclerView Adapter**:
    a. Extends `RecyclerView.Adapter`.
    b. Overrides methods like `onCreateViewHolder`, `onBindViewHolder`, and `getItemCount`.

4. **Setup RecyclerView in Activity/Fragment**:
    a. Initialize `RecyclerView`.
    b. Set layout manager (e.g., `LinearLayoutManager`).
    c. Attach the adapter with a contact list.

Advantages of RecyclerView

- **Efficient Memory Use**: Reuses views with `ViewHolder`.
- **Flexible Layouts**: Supports grid, list, or staggered layouts.
- **Animation Support**: In-built support for item animations.
- **Large Dataset Handling**: Suitable for thousands of items.

Optional Enhancements

- Fetch contacts from **device's Contacts Content Provider**.
- Add **click listeners** to make a call or open a detailed view
- Implement **search or filter** functionality.

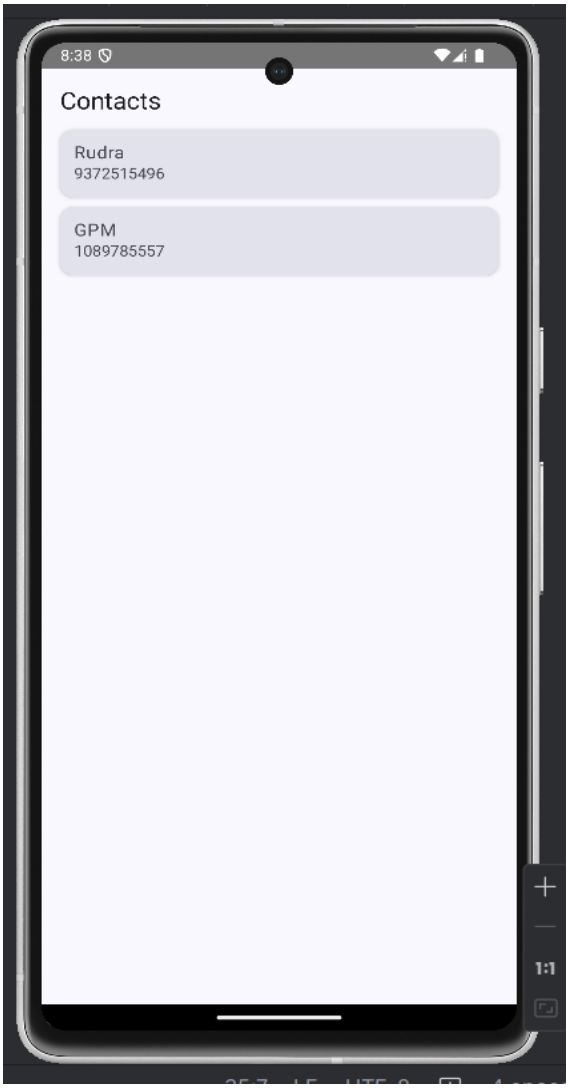
- Use **Room Database** for storing and managing contact data locally.

Code:

```
package com.example.sample


import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import com.example.sample.ui.theme.SampleTheme


class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState);
setContent {
      SampleTheme {
         Surface(modifier = Modifier.fillMaxSize()) { ContactListScreen() }
      }
   }}
}


data class Contact(val name: String, val phone: String)
```

```
@Composable
fun ContactListScreen() {
    val contacts = listOf(
        Contact("Rudra", "9372515496"),
        Contact("GPM", "1089785557")
    )

    Column(modifier = Modifier.padding(16.dp)) {
        Text("Contacts", style = MaterialTheme.typography.titleLarge)
        Spacer(Modifier.height(8.dp))
        LazyColumn {
            items(contacts) { contact -> ContactItem(contact) }
        }
    }
}

@Composable
fun ContactItem(contact: Contact) {
    Card(
        modifier = Modifier.fillMaxWidth().padding(vertical = 4.dp),
        elevation = CardDefaults.cardElevation(2.dp)
    ) {
        Column(modifier = Modifier.padding(12.dp)) {
            Text(contact.name, style = MaterialTheme.typography.bodyLarge)
            Text(contact.phone, style = MaterialTheme.typography.bodyMedium)
        }
    }
}
```

Output:



Conclusion:

By using `RecyclerView` in Android, developers can create a powerful, scalable, and user-friendly contact list application. This component, paired with a custom adapter and efficient data binding, results in a smooth and modern user experience, especially suited for lists that change dynamically or have a large number of items.

Experiment No :7

Aim: Develop an application to make and receive calls on mobile.

Theory:

Develop an Android Application to Make and Receive Calls

Introduction

Mobile communication is a core feature of smartphones. Android allows developers to integrate calling functionality using the **`Telecom` and `Telephony`** APIs. This application enables users to **initiate phone calls** and potentially **respond to incoming calls** (with limitations due to security and privacy restrictions on Android versions 10+).

Objective

- Enable users to **make outgoing phone calls** from within the app.
- (Limited) Monitor or respond to **incoming calls**, where permitted.
- Handle permissions, call intents, and phone state.

Tools and Technologies

- **Android Studio**
- **Java/Kotlin**
- **TelephonyManager**: To listen to call state changes.
- **Intent.ACTION_CALL**: To initiate calls.
- **BroadcastReceiver**: To monitor phone state changes.
- **Manifest Permissions**: For phone-related operations.

Application Workflow

A. Making Calls

1. User inputs a phone number.
2. App triggers an intent.
3. The system dials the number using the phone's dialer.

B. Receiving Calls (Limited)

- Use `TelephonyManager` to monitor incoming call state.
- Optionally, create a `BroadcastReceiver` for `PHONE_STATE`.
- To automatically answer a call, app must be the **default dialer**, which is not allowed for most third-party apps.

Security and Privacy Considerations

- Always request **runtime permissions** for `CALL_PHONE`.
- Never place calls without user knowledge or consent.
- For Android 10+, auto-answering is highly restricted due to user privacy concerns.

Limitations

- **Auto-answering calls** is **not allowed** on most Android versions unless your app is set as the **default dialer**.
- Monitoring call logs or phone numbers requires **special permissions**.
- Background call handling is restricted starting Android 10 (API 29) due to tightened security.

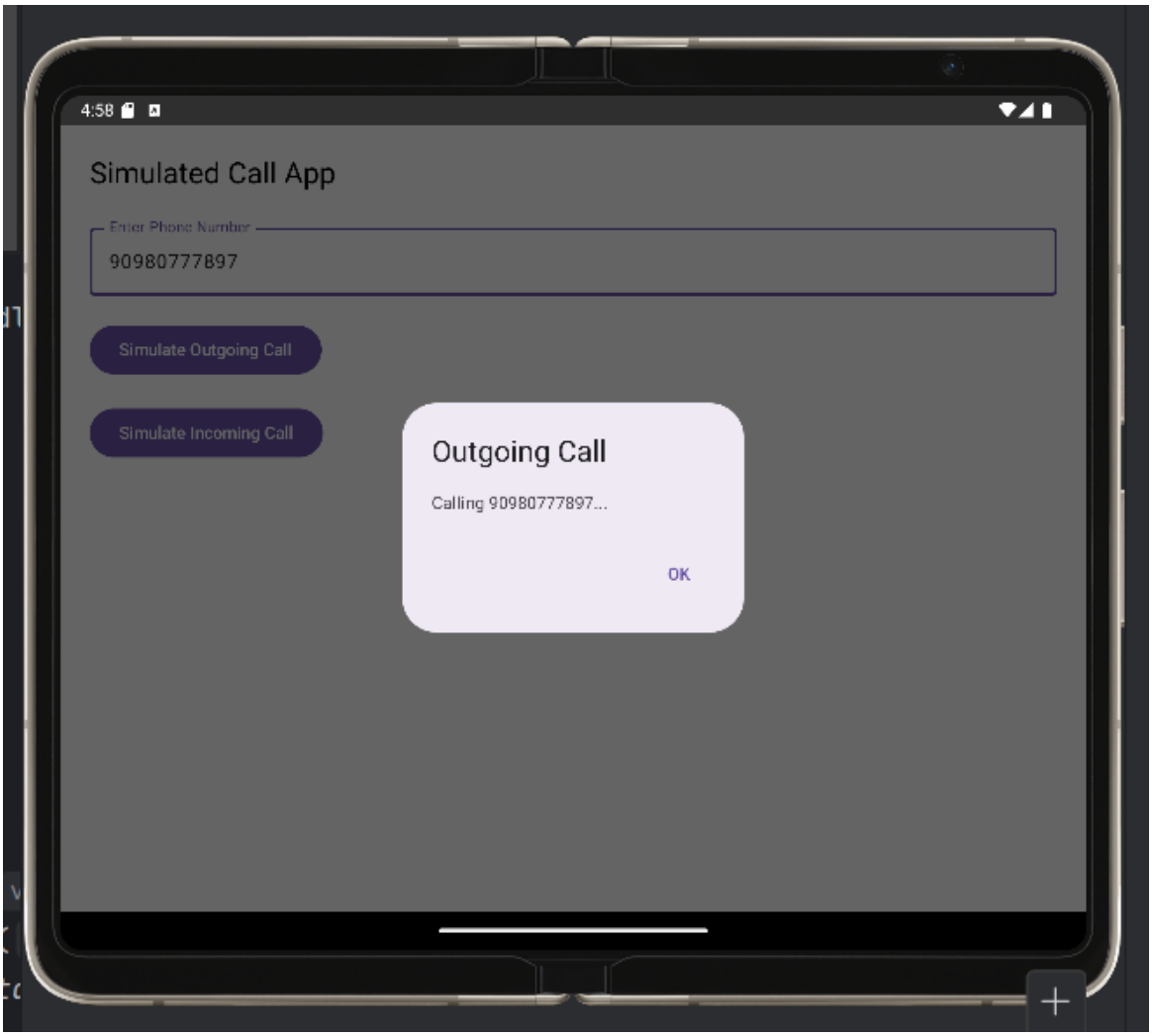
Code:

```
package com.example.call


import android.os.Bundle

import androidx.activity.ComponentActivity

import androidx.activity.compose.setContent
```

```
import androidx.compose.foundation.layout.*
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp

class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState);
setContent {
      MaterialTheme { SimulatedCallApp() }
   }}
}

@Composable
fun SimulatedCallApp() {
   var phoneNumber by remember { mutableStateOf("") }
   var incomingCall by remember { mutableStateOf(false) }
   var showOutgoingDialog by remember { mutableStateOf(false) }

   Column(modifier = Modifier.fillMaxSize().padding(24.dp), verticalArrangement =
Arrangement.spacedBy(20.dp)) {
      Text("Simulated Call App", style = MaterialTheme.typography.headlineSmall)

      OutlinedTextField(value = phoneNumber, onValueChange = { phoneNumber = it }, label =
{ Text("Enter Phone Number") }, modifier = Modifier.fillMaxWidth())

      Button(onClick = {
         if (phoneNumber.isNotBlank()) {
            incomingCall = false
```

```
            showOutgoingDialog = true
        }
    }) { Text("Simulate Outgoing Call") }
    Button(onClick = { incomingCall = true }) { Text("Simulate Incoming Call") }
    if (incomingCall) {
        AlertDialog(
            onDismissRequest = { incomingCall = false },
            title = { Text("Incoming Call") },
            text = { Text("Call from +91 99999 88888") },
            confirmButton = { TextButton(onClick = { incomingCall = false }) { Text("Accept")
}},
            dismissButton = { TextButton(onClick = { incomingCall = false }) { Text("Reject") }}
        )}
    if (showOutgoingDialog) {
        AlertDialog(
            onDismissRequest = { showOutgoingDialog = false },
            title = { Text("Outgoing Call") },
            text = { Text("Calling $phoneNumber...") },
            confirmButton = { TextButton(onClick = { showOutgoingDialog = false }) {
Text("OK") }}    dismissButton = {})}}}
```

Output:



Conclusion:

Developing an app that allows users to **make phone calls** is straightforward using `Intent.ACTION_CALL`, provided proper permissions are handled. Receiving or auto-answering calls is significantly restricted on newer Android versions for security reasons. Developers must respect user privacy and comply with system permission policies.

Experiment No :08

Aim: Design an android based application to take a snapshot by using the Camera in your mobile.

Theory:

Designing an Android Application to Take a Snapshot Using the Camera

Introduction

Modern Android devices are equipped with powerful cameras. Android provides a straightforward way for applications to access the camera and capture images using either the **built-in camera app** (via **Intents**) or by using the **Camera API** for direct control. In this app, the focus is on using the built-in camera app to take a snapshot and display the result.

Objective

- Launch the device's **camera application** from within your app.
- **Capture a photo (snapshot)**.
- **Display the captured image** inside the app.
- Handle **permissions and file storage** securely.

Tools and Technologies

- **Android Studio** – Development environment.
- **Java/Kotlin** – Programming language.
- **Camera Intent (`MediaStore.ACTION_IMAGE_CAPTURE`)** – To launch the camera app.

- **FileProvider** – To securely share image file paths.

Application Workflow

Step 1: Create the UI

- One **Button** to open the camera.
- One **ImageView** to display the captured image.

Step 2: Handle Runtime Permissions

Request camera permission at runtime (Android 6.0+).

Step 3: Launch Camera Intent

Use the `MediaStore.ACTION_IMAGE_CAPTURE` intent:

```
java
CopyEdit
Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
startActivityForResult(intent, REQUEST_IMAGE_CAPTURE);
```

Step 4: Handle Image Result

Override `onActivityResult()` to get the image as a `Bitmap` and display it in the `ImageView`:

```
java
CopyEdit
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode ==
RESULT_OK) {
        Bundle extras = data.getExtras();
        Bitmap imageBitmap = (Bitmap) extras.get("data");
        imageView.setImageBitmap(imageBitmap);
    }
}
```

Note: This returns a **low-resolution thumbnail**. For full-resolution images, you must save the image to a file and pass the URI to the camera intent using `FileProvider`.

Advanced Option: Capture High-Quality Image

To capture a high-quality image:

- Create a file in storage.
- Use `FileProvider` to get a URI.
- Pass this URI with `Intent.putExtra(MediaStore.EXTRA_OUTPUT, uri)`.

Security and Best Practices

- Always **check and request permissions at runtime**.
- Use **`FileProvider`** for sharing file URIs securely.
- Avoid using deprecated or legacy storage access methods on Android 10+.
- Handle different device resolutions and orientations.

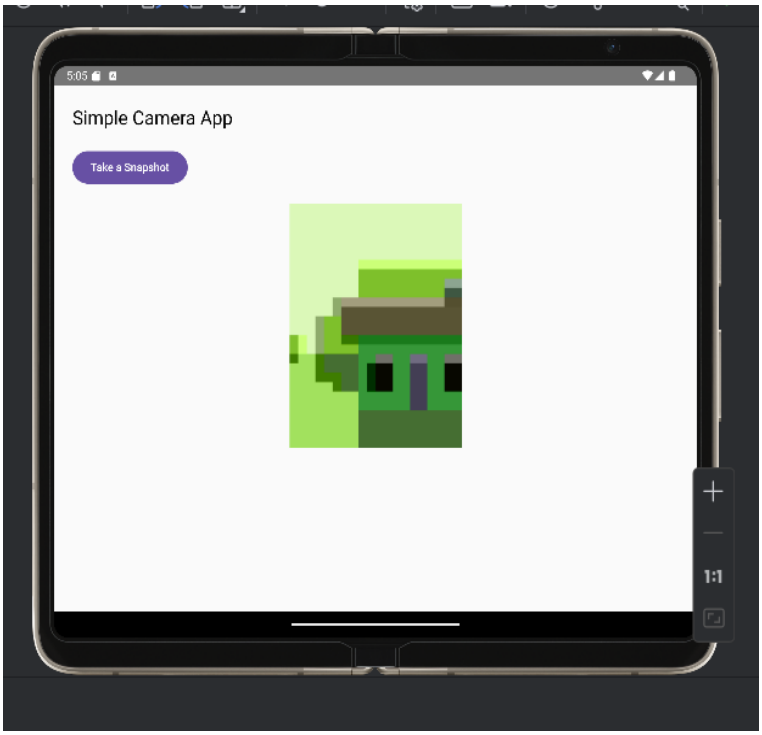
Code:

```
package com.example.call


import android.graphics.Bitmap

import android.os.Bundle

import androidx.activity.ComponentActivity

import androidx.activity.compose.setContent

import androidx.activity.result.contract.ActivityResultContracts

import androidx.activity.compose.rememberLauncherForActivityResult

import androidx.activity.result.launch

import androidx.compose.foundation.Image

import androidx.compose.foundation.layout.*
```

```
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.asImageBitmap
import androidx.compose.ui.unit.dp

class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState)
      setContent {
         var imageBitmap by remember { mutableStateOf<Bitmap?>(null) }

         val launcher =
rememberLauncherForActivityResult(ActivityResultContracts.TakePicturePreview()) { bitmap -
>
            imageBitmap = bitmap
         }

         MaterialTheme {
            Column(
               modifier = Modifier.fillMaxSize().padding(24.dp),
               verticalArrangement = Arrangement.spacedBy(20.dp)
            ) {
               Text("Simple Camera App", style = MaterialTheme.typography.headlineSmall)

               Button(onClick = { launcher.launch() }) { Text("Take a Snapshot") }

               imageBitmap?.let {
                  Image(
                     bitmap = it.asImageBitmap(),
```

```
                contentDescription = null,
                modifier = Modifier.fillMaxWidth().height(300.dp)
            )
        }
    }
  }
 }
}}
```

Output:



Conclusion: Taking a snapshot using the device's camera in an Android app can be efficiently implemented using camera intents. This approach is simple, requires minimal permissions, and ensures compatibility across devices. For more control (e.g., custom camera UI, real-time filters), developers can use the **CameraX** or **Camera2 API**.

Experiment No :09

Aim: Develop an application to access Accelerometer, Gyroscope, Orientation Sensors and to display data received from each sensor.

Theory:

Develop an Android Application to Access and Display Sensor Data

Introduction

Android devices include a variety of built-in sensors that provide information about the device's environment and movement. Three commonly used motion and position sensors are:

- **Accelerometer**: Measures acceleration force (including gravity).
- **Gyroscope**: Measures rate of rotation around the device's axes.
- **Orientation Sensor**: Computes device orientation (deprecated, but can be derived).

This application accesses these sensors using the **Android Sensor Framework** and displays live data.

Objective

- Access built-in motion sensors using the `SensorManager`.
- Capture real-time data from:
    - **Accelerometer**
    - **Gyroscope**
    - **Orientation** (calculated using accelerometer + magnetometer).
- Display sensor data on the screen.

Tools and Technologies

- **Android Studio**
- **Java** or **Kotlin**
- **SensorManager** and **SensorEventListener**
- **UI Components** (e.g., `TextView`)

Key Components

Component	Purpose
`SensorManager`	Manages access to sensors.
`Sensor`	Represents a specific sensor (e.g., accelerometer).
`SensorEventListener`	Receives sensor data updates.

Required Permissions

No special permissions are required to use motion sensors. Just declare in your activity.

Application Workflow

Step 1: UI Design

- Create `TextViews` to display sensor data (e.g., X, Y, Z values for each sensor).

Step 2: Initialize Sensors

- Get the `SensorManager` using:

```
java
CopyEdit
SensorManager sensorManager = (SensorManager)
getSystemService(Context.SENSOR_SERVICE);
```

- Register listeners for each sensor:

```
java
CopyEdit
Sensor accelerometer =
sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
sensorManager.registerListener(sensorListener, accelerometer,
SensorManager.SENSOR_DELAY_NORMAL);
```

Step 3: Handle Sensor Data

Implement `SensorEventListener`:

```
java
CopyEdit
@Override
public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        float x = event.values[0];
        float y = event.values[1];
        float z = event.values[2];
        // Update UI
    }
    // Similar for Gyroscope and Orientation
}
```

Step 4: Unregister on Pause

Unregister listeners to save battery:

```
java
CopyEdit
@Override
protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(sensorListener);
}
```

Sensor Descriptions

Sensor	Data Provided
Accelerometer	Acceleration (m/s²) on X, Y, Z axes (includes gravity).
Gyroscope	Rotation rate (rad/s) around X, Y, Z axes.
Orientation	Deprecated sensor; derive orientation using accelerometer + magnetometer.

For orientation, use `Sensor.TYPE_MAGNETIC_FIELD` with `Sensor.TYPE_ACCELEROMETER` and `SensorManager.getRotationMatrix()` to calculate orientation angles (azimuth, pitch, roll).

Code:

package com.example.call

import android.hardware.*

import android.os.Bundle

import androidx.activity.ComponentActivity

import androidx.activity.compose.setContent

import androidx.compose.foundation.layout.*

import androidx.compose.material3.*

import androidx.compose.runtime.*

```
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.core.content.getSystemService

class MainActivity : ComponentActivity(), SensorEventListener {
    private lateinit var sensorManager: SensorManager
    private var accelerometerData = mutableStateOf(Triple(0f, 0f, 0f))
    private var gyroscopeData = mutableStateOf(Triple(0f, 0f, 0f))
    private var orientationData = mutableStateOf(Triple(0f, 0f, 0f))

    override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState)
        sensorManager = getSystemService()!!
        registerSensor(Sensor.TYPE_ACCELEROMETER)
        registerSensor(Sensor.TYPE_GYROSCOPE)
        registerSensor(Sensor.TYPE_ROTATION_VECTOR)
        setContent {
            MaterialTheme {
                SensorDisplayUI(
                    accelerometerData.value,
                    gyroscopeData.value,
                    orientationData.value
                )
            }
        }
    }

    private fun registerSensor(sensorType: Int) {
        sensorManager.getDefaultSensor(sensorType)?.also { sensor ->
```

```
            sensorManager.registerListener(this, sensor, SensorManager.SENSOR_DELAY_UI)
        }
    }

    override fun onSensorChanged(event: SensorEvent?) {
        event ?: return
        when (event.sensor.type) {
            Sensor.TYPE_ACCELEROMETER -> accelerometerData.value =
Triple(event.values[0], event.values[1], event.values[2])
            Sensor.TYPE_GYROSCOPE -> gyroscopeData.value = Triple(event.values[0],
event.values[1], event.values[2])
            Sensor.TYPE_ROTATION_VECTOR -> {
                val orientationAngles = FloatArray(3)
                val rotationMatrix = FloatArray(9)
                SensorManager.getRotationMatrixFromVector(rotationMatrix, event.values)
                SensorManager.getOrientation(rotationMatrix, orientationAngles)
                orientationData.value = Triple(
                    Math.toDegrees(orientationAngles[0].toDouble()).toFloat(),
                    Math.toDegrees(orientationAngles[1].toDouble()).toFloat(),
                    Math.toDegrees(orientationAngles[2].toDouble()).toFloat()
                )
            }
        }
    }

    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {}
    override fun onDestroy() { super.onDestroy(); sensorManager.unregisterListener(this) }
}
```

```
@Composable
fun SensorDisplayUI(
    accel: Triple<Float, Float, Float>,
    gyro: Triple<Float, Float, Float>,
    orient: Triple<Float, Float, Float>
) {
    Column(modifier = Modifier.padding(24.dp)) {
        Text("Accelerometer:\nX=${accel.first} Y=${accel.second} Z=${accel.third}")
        Spacer(Modifier.height(16.dp))
        Text("Gyroscope:\nX=${gyro.first} Y=${gyro.second} Z=${gyro.third}")
        Spacer(Modifier.height(16.dp))
        Text("Orientation
(degrees):\nAzimuth=${orient.first}\nPitch=${orient.second}\nRoll=${orient.third}")
    }
}
```

Output:

Conclusion: Using the Android Sensor API, we can access real-time data from accelerometer, gyroscope, and orientation sensors. This is useful in applications like fitness tracking, game development, and AR. Best practices include efficient sampling, UI updates on the main thread, and resource management (unregistering sensors when not in use).

Experiment No :10

Aim: Publish all the above apps on your own website.

Theory:

Publishing Android Applications on a Personal Website

1. Introduction

Distributing Android applications through your own website is an alternative to publishing on the Google Play Store. This approach is particularly useful for sharing apps within a closed group (e.g., students, beta testers) or showcasing portfolio projects. Users can **download the APK files directly** from your website and install them on their devices.

2. Objective

To make Android apps available for download through a personal website by:

- Building the APK files.
- Hosting them on a web server.
- Providing proper download links and app information.

3. Prerequisites

- A working Android app (.apk file).
- A registered domain and hosting space or a platform like GitHub Pages.
- Basic knowledge of web development (HTML/CSS).
- Signed APKs for release.

4. Process Overview

A. Build APK File

- Use **Android Studio** to build a signed APK.
- Navigate to `Build > Build Bundle(s) / APK(s) > Build APK(s)` to generate it.

B. Host the APK File

- Upload the `.apk` file to your website's **hosting directory** or a public cloud drive with direct access.

C. Create Webpage with Download Link

- Design an HTML page with information about the app, a brief description, and a **download button** linking to the APK file.

Example:

```
html
CopyEdit
<a href="yourserver.com/apps/myapp.apk" download>Download App</a>
```

D. User Side – Installation

- Users need to enable **"Install from Unknown Sources"** on their Android device.
- After downloading the APK, they install it manually.

5. Advantages

- **Full control** over app distribution.
- **No approval process** or Play Store restrictions.
- Ideal for testing, prototypes, or private use.

6. Limitations

- **Security warnings**: Devices may warn about installing apps from unknown sources.
- **No automatic updates**: You must notify users of new versions.
- **Trust and visibility**: Not as trusted or visible as Play Store apps.
- **APK compatibility**: APKs must be compatible with the target Android versions.

7. Security and Best Practices

- Sign the APK with a **release key** to prevent tampering.
- Use HTTPS for secure downloads.
- Mention **permissions** and **app details** clearly on the site.
- Respect user data and privacy if your app handles sensitive information.

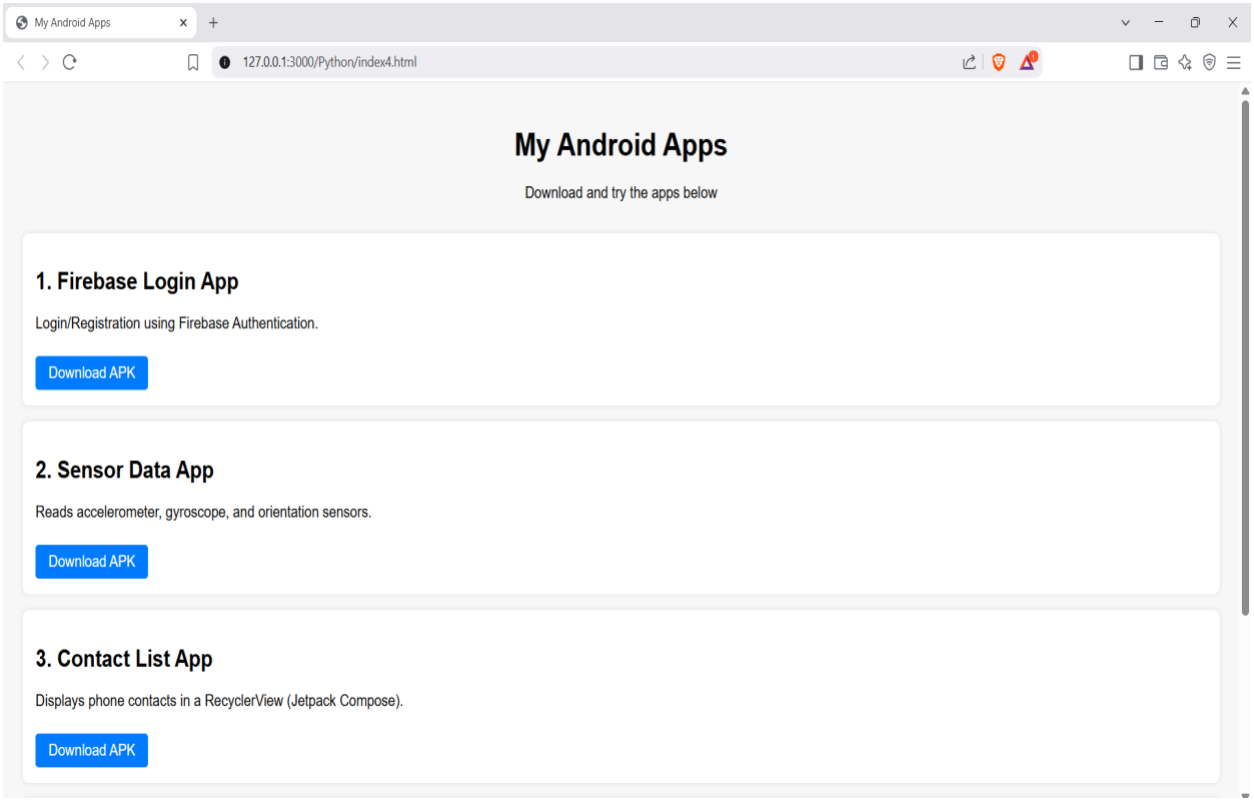
Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>My Android Apps</title>
  <link rel="stylesheet" href="styles.css" />
</head>
<body>
  <header>
    <h1>My Android Apps</h1>
    <p>Download and try the apps below</p>
  </header>

  <main>
```

```
  <section>
    <h2>1. Firebase Login App</h2>
    <p>Login/Registration using Firebase Authentication.</p>
    <a href="apps/firebase-login.apk" download>Download APK</a>
  </section>

  <section>
    <h2>2. Sensor Data App</h2>
    <p>Reads accelerometer, gyroscope, and orientation sensors.</p>
    <a href="apps/sensors-app.apk" download>Download APK</a>
  </section>

  <section>
    <h2>3. Contact List App</h2>
    <p>Displays phone contacts in a RecyclerView (Jetpack Compose).</p>
    <a href="apps/contact-list.apk" download>Download APK</a>
  </section>

  <section>
    <h2>4. Simple Login App (No Auth)</h2>
    <p>Basic register and login UI without Firebase.</p>
    <a href="apps/simple-login.apk" download>Download APK</a>
  </section>
</main>

<footer>
  <p>&copy; 2025 My Android Projects</p>
</footer>
```

```
</body>
</html>
```

Output:



Conclusion: Publishing Android apps on your own website is a practical and flexible method for distributing apps outside the Play Store. It provides full control over the distribution process, but it also requires handling security, updates, and installation instructions manually. It's well-suited for internal tools, demos, and educational purposes.