MOBILE APPLICATION DEVELOPMENT

As per the New NEP Syllabus for BCA 6th Semester Course of

Bengaluru City University and Bangalore University

Authored by

Srikanth S

Manoj Kumar N
Assistant Professor
Dept. of Computer Science
Soundarya Institute of Management and Science, Bengaluru

Anuradha Sindhia
Assistant Professor
Dept. of Computer Science
Soundarya Institute of Management and Science, Bengaluru

Skyward Publishers
#157, 7th Cross, 3rd Main Road, Chamarajpet,
Bengaluru-18. Phone : 080-43706620 / 080-26603535
Mob: 9611185999
E-mail: skyward.publishers@gmail.com
Website: www.skywardpublishers.co.in

A Text Book of **"Mobile Application Development"** - by Srikanth S , Manoj Kumar N & Anuradha Sindhia as per the New NEP Syllabus for VI Semester BCA, Bengaluru City University & Bangalore University.

© Authors

Copy Right : *All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Unauthorized reproduction or distribution of this book, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law. This includes the transmission of this book in digital form such as images or PDF files. Any such activities will be considered a violation of Indian Copyright Laws and are highly punishable.*

Every effort has been made to avoid errors or omissions in this publication. In spite of this, some errors might have crept in. Any mistake, error or discrepancy noted may be brought to our notice which shall be taken care in the next edition. The publisher shall not verify the originality, authenticity, ownership, non-infringement of the data, content, and information. The Authors are the sole owners of the copyrights of the Work. It shall be Authors sole responsibility to ensure the lawfulness of the content and publisher is not responsible for any copyright issues. It is notified that publisher will not be responsible for any damage or loss of action any one, of any kind, in any manner, there from all disputes are subject to Bengaluru jurisdiction only.

Disclaimer: *Skyward Publishers has exercised due care and caution in collecting all the data before publishing the book. In spite of this, if any omission, inaccuracy or printing error occurs with regards to the data contained in this book, Skyward Publishers will not be held responsible or liable. Skyward Publishers will be grateful for your suggestions which will be of great help for other readers.*

First Edition : 2024

Price : ₹ 550.00

Published by:

Skyward Publishers

#157, 7th Cross, 3rd Main Road, Chamarajpet
Bangalore-18. Phone : 080-26603535 / 43706620,
Mob: 9611185999
E-mail: skyward.publishers@gmail.com
Website: www.skywardpublishers.co.in

DTP By

Mary & Nirmala, Skyward Team

PREFACE

Welcome to the comprehensive guide on Mobile Application Development for the BCA 6th Semester Course, meticulously crafted as per the NEP syllabus prescribed by Bengaluru City University and Bangalore University. This book is designed to serve as a fundamental reference for students and aspiring developers keen on mastering the art and science of mobile app development.

The field of mobile application development is evolving at a breakneck pace, driven by the continuous advancements in mobile technologies. With this book, we aim to bridge the gap between theoretical knowledge and practical application, providing readers with a solid foundation in both the principles and practices of mobile app development.

The content is structured to facilitate a step-by-step learning process, with each chapter building upon the previous one. We emphasize a hands-on approach, ensuring that readers not only understand the concepts but also apply them through practical exercises and real-world projects. The book includes:

- **Sample Applications :** Each concept is paired with practical applications, allowing students and learners to see the theory in action.
- **Step-by-Step Implementation :** Detailed instructions guide the reader through the process of building functional mobile applications from scratch.
- **Source Code and Output :** Complete source code examples and actual outputs are provided to enhance understanding and facilitate debugging.
- **Model Question Papers :** Four model question papers are included to help students prepare for their exams effectively.
- **Lab Programs:** The lab programs are covered to reinforce learning and provide practical experience.

This book is primarily intended for BCA students of Bengaluru City University and Bangalore University. However, it is also an invaluable resource for anyone interested in mobile app development, whether you are a student, educator, or professional developer.

Don't get scared about the size of the book. The size is huge because we have covered the practical approach of creating apps for every concept. Without this, it's no use of studying this subject.

I welcome your suggestions and feedback to continuously improve this book. Please feel free to reach out at skyward.publishers@gmail.com.

Happy Coding!

Srikanth.S

SYLLABUS

UNIT – 1 : **[15 Hours]**

Introduction : Brief History of mobile technologies, Different mobile technologies Key Mobile Application Services-Introducing Android, The Android Application Components, Exploring the Development Environment, -Obtaining the Required Tools-Launching Your First Android Application-Exploring the IDE-Debugging Your Application-Publishing Your Application

Using Activities - Fragments and Intents in Android : Working with activities, Using Intents, Fragments, Using the Intent Object to Invoke Built-in Application

UNIT – II : **[15 Hours]**

Working with the User Interface Using views

Understanding the Components of a Screen-Adapting to Display Orientation-Managing Changes to Screen Orientation- Utilizing the Action Bar-Creating the User Interface Programmatically Listening for UI Notification

Using Basic Views-Using Picker Views -Using List Views to Display Long Lists-Understanding Specialized Fragments - Using Image Views to Display Pictures -Using Menus with Views Using WebView- Saving and Loading User Preferences-Persisting Data to Files-Creating and Using Databases.

UNIT – III : **[15 Hours]**

Designing User interface Designing by declaration, creating the opening screen, using alternate resources, implementing an about box, applying a theme, adding a menu, adding settings, debugging with log messages, debugging with debugger

UNIT – IV : **[15 Hours]**

Creating Your Own Content Providers -Using the Content Provider, SMS Messaging - Sending Email-Displaying Maps- Getting Location Data- Monitoring a Location

Putting SQL to work Introducing SQLite, In and Out of SQLite, Hello Database, Data Binding, using content provider, implementing content provider. Reading/writing local data, Accessing the Internal File system, Accessing the SD card. Preparing app for publishing, Deploying APK files, uploading in Market , Consuming Web Services Using HTTP-Consuming JSON Services- Creating Your Own Services Binding Activities to Services -Understanding Threading

CONTENTS

Unit - I

Chapter - 1 Introduction to Mobile Technologies and Key Services 1.1 - 1.18

1.1 Introduction 1.2
1.2 Brief History of Mobile Technologies 1.4
1.3 Different Mobile Technologies 1.6
1.3.1 Cellular Networks 1.6
1.3.2 Mobile Operating Systems (OS) 1.7
1.3.3 Mobile Development Frameworks 1.7
1.3.4 Mobile Web Technologies 1.10
1.3.5 Connectivity and Communication 1.10
1.3.6 Sensors and Hardware Innovations 1.11
1.3.7 Location-Based Services (LBS) 1.12
1.3.8 Cloud Storage 1.13
1.3.9 Emerging Technologies 1.13
1.3.10 Artificial Intelligence (AI) in Mobile 1.14
1.4 Key Mobile Application Services 1.15
1.5 Review Questions 1.17

Chapter - 2 Introducing Android 2.1 - 2.52

2.1 Introduction 2.2
2.1.1 Brief History of Android 2.2
2.1.2 What is Android ? 2.3
2.1.3 Key Features of Android 2.3
2.1.4 Mobile Application Development Using Android 2.6
2.1.5 Architecture of Android 2.6
2.1.6 Native Android Application 2.7
2.1.7 Advantages and Disadvantages of Android 2.8
2.2 The Android Application Components 2.9
2.2.1 Primary (or) Core Components 2.9
2.2.2 Secondary (or) Non-Core Components 2.10
2.3 Exploring the Development Environment 2.13
2.3.1 Android SDK 2.14
2.3.2 Android Studio 2.14
2.3.3 Android Emulators 2.15
2.3.4 ADB (Android Debug Bridge) 2.16
2.3.5 Gradle 2.16
2.4 Obtaining the Required Tools 2.16
2.5 Installing Android Studio 2.17
2.6 Your First Android Application - Hello World 2.20
2.6.1 Understanding the Flow of Execution 2.25
2.7 Exploring the IDE (Android Studio) 2.26

2.7.1 Exploring the Integrated Development Environment (IDE) in Android Studio 2.26
2.7.2 Android Studio Developer Workflow Guide 2.31
2.7.3 Understanding the Project Structure in Android Development 2.32
2.7.4 Android Manifest File 2.33
2.7.5 Creating a Virtual Device 2.38
2.7.6 Using Lightweight Emulator (LDPlayer9) for Enhanced Android Development 2.40
2.7.7 Testing Using a Physical Device 2.42
2.7.8 Creating Global gradle.properties File 2.43
2.8 Debugging your Android Application in Android Studio 2.44
2.8.1 Overview of Debugging Tools in Android Studio 2.44
2.8.2 Steps to Debug an Android Application in Android Studio: 2.44
2.9 Publishing your Android Application 2.46
2.9.1 Important Terms and Key Elements in Publishing an Android Application 2.46
2.9.2 Steps in Publishing Android Application 2.48
2.10 Review Questions 2.51

Chapter - 3 Using Activities, Fragments and Intents in Android 3.1 - 3.72

3.1 Introduction 3.2
3.2 Working with Activities 3.2
3.2.1 Creating an Activity 3.3
3.2.2 Understanding Activity Class 3.5
3.2.3 Activity Life Cycle 3.6
3.2.4 Features and Functionalities of an Activity 3.10
3.2.5 Applying Styles and Themes to an Activity 3.12
3.2.6 Hiding the Activity Title 3.15
3.2.7 Displaying a Dialog Window Using an Activity 3.16
3.2.8 ProgressBar in Android 3.19
3.3 Using Intents 3.22
3.3.1 Features and Functionalities of Intents 3.24
3.3.2 Creating an Intent 3.25
3.3.3 Navigating Between Activities (or) Linking Activities Using Intents 3.25
3.3.4 Types of Intents 3.29
3.3.5 Intent Class Methods 3.31
3.3.6 Intent Class Attributes 3.33
3.3.7 Intent Filters 3.33
3.3.8 Returning Results from an Intent 3.36
3.3.9 Passing Data with Intents 3.40
3.3.10 Using the Intent Object to Invoke Built-in Application 3.40
3.4 Fragments 3.43
3.4.1 Features and Benefits of Fragments 3.45
3.4.2 Fragment Lifecycle 3.46
3.4.3 Implementing Fragments in Android Apps 3.49
3.4.4 Creating and Using Fragments 3.51
3.4.5 Adding Fragments Dynamically 3.56

3.5 Understanding the Back Stack in Android 3.61
3.6 Managing Activity Navigation and Back Stack 3.61
3.7 Implementing a New Activity Using Explicit Intent, Implicit Intent and Content Provider 3.66
3.8 Review Questions 3.71

Unit - II

Chapter - 4 Understanding The android User Interface 4.1 - 4.84

4.1 Introduction 4.2
4.2 Compilation Process of XML Layout Files in Android 4.2
4.3 Understanding the Components of a Screen 4.3
4.3.1 View 4.4
4.3.2 Viewgroup 4.8
4.3.3 Layouts 4.11
4.3.3.1 Linear Layout 4.14
4.3.3.2 RelativeLayout 4.18
4.3.3.3 Constraint Layout 4.21
4.3.3.4 Frame Layout 4.26
4.3.3.5 Table Layout 4.29
4.3.3.6 Scroll View 4.33
4.4 Adapting to Display Orientation 4.37
4.4.1 Anchoring Views 4.37
4.4.2 Resizing and Repositioning Views 4.40
4.4.3 Creating Separate Layouts for Different Orientations 4.43
4.5 Managing Changes To Screen Orientation 4.48
4.5.1 What Happens to an Activity's State When Orientation Changes? 4.48
4.5.2 Persisting State Information during Changes in Configuration 4.49
4.5.3 Detecting Orientation Changes 4.53
4.5.4 Controlling the Orientation of the Activity 4.56
4.6 Utilizing The Action Bar 4.57
4.6.1 Creating an ActionBar 4.59
4.6.2 Adding Action Items to the Action Bar 4.62
4.7 Creating The User Interface Programmatically 4.66
4.8 Listening For UI Notifications 4.70
4.8.1 Listening for UI Notifications at the Activity Level 4.71
4.8.2 Registering Views for Notifications 4.76
4.9 Review Questions 4.81

Chapter - 5 Designing User Interface with Views 5.1 - 5.142

5.1 Introduction 5.2
5.1.1 View Attributes in Android 5.2
5.1.2 Defining View Attributes in XML Layout Files 5.3
5.1.3 Setting Attributes Programmatically 5.3
5.1.4 Common Attributes for All Views 5.4

5.2 Using Basic Views 5.6
5.2.1 TextView 5.6
5.2.2. EditText View 5.9
5.2.3 Button 5.13
5.2.4 ImageButton 5.16
5.2.5 CheckBox 5.19
5.2.6 ToggleButton 5.23
5.2.7 RadioButton 5.27
5.2.8 ProgressBar 5.31
5.2.9 AutoCompleteTextView 5.35
5.3 Using Picker Views 5.38
5.3.1 Date Picker 5.39
5.3.2 TimePicker 5.40
5.3.3 NumberPicker 5.41
5.4 Using List Views to Display Long Lists 5.46
5.4.1 ListView 5.47
5.4.2 RecyclerView 5.53
5.4.3 SpinnerView 5.58
5.5 Understanding Specialized Fragments 5.62
5.5.1 Using a ListFragment 5.62
5.5.2 Using a DialogFragment 5.66
5.5.3 Using a PreferenceFragment 5.70
5.6 Using Image Views to Display Pictures 5.75
5.6.1 Using ImageView 5.75
5.6.2 Using GridView 5.79
5.6.3 Using ImageSwitcher 5.84
5.7 Using Menus With Views 5.89
5.8 Using WebView 5.94
5.9 Data Persistence 5.97
5.9.1 Saving and Loading User Preferences 5.99
5.9.2 Persisting Data to Files 5.106
5.9.3 Creating and Using Databases 5.115
5.10 Sample Applications 5.122
5.10.1 Developing a Login Window Using UI Controls 5.122
5.10.2 Developing Login Window and Validating User Credentials. 5.125
5.10.3 Creating an UI with All Views 5.128
5.10.4 Creating an Application with a Custom Designed Opening Screen 5.134
5.11 Review Questions 5.137

Unit - III

Chapter - 6 **Designing User Interface (Sudoku Game app)** **6.1 - 6.38**

6.1 Introduction 6.2
6.2 Understanding Sudoku Game 6.2

6.3 Designing by Declaration 6.4
6.4 Creating the Opening Screen 6.5
6.5 Using Alternate Resources 6.12
6.5.1 Using ScrollView 6.13
6.5.2 Using Different Layout for Landscape Mode 6.15
6.6 Implementing an About Box 6.16
6.7 Applying a Theme 6.20
6.8 Adding a Menu 6.22
6.9 Adding Settings 6.26
6.10 Debugging 6.30
6.10.1 Debugging with Log Messages 6.31
6.10.2 Debugging with Debugger 6.34
6.11 Review Questions 6.37

Unit - IV

Chapter - 7 SMS Messaging, Email and Location based Services 7.1 - 7.48

7.1 SMS Messaging 7.2
7.1.1 Basic Steps to Send an SMS in Android 7.3
7.1.2 Step-by-Step Implementation of Sending SMS 7.4
7.2 Sending Email 7.9
7.2 1 Email Related Classes and Methods in Android 7.10
7.2.2 Basic Steps to Send an Email in Android 7.10
7.2.3 Step-by-Step Implementation of Sending an Email 7.10
7.3 Location-Based Services (LBS) 7.14
7.3.1 Displaying Maps 7.17
7.3.2 Getting Location Data 7.34
7.3.3 Monitoring a Location 7.40
7.4 Review Questions 7.46

Chapter - 8 Putting SQL to Work 8.1 - 8.34

8.1 Introduction to SQLite 8.2
8.1.1 Features of SQLite 8.2
8.1.2 Common Applications of SQLite Database 8.3
8.1.3 SQLite in Android 8.3
8.1.4 Advantages and Disadvantages of SQLite 8.4
8.2 In and Out of SQLite 8.5
8.3 Hello, Database 8.9
8.4 Data Binding 8.23
8.5 Review Questions 8.33

Chapter - 9 Content Providers 9.1 - 9.46

9.1 Content Providers 9.2
9.2 Features of Content Provider 9.3
9.3 Advantages and Disadvantages of Content Provider 9.5
9.4 Content Provider URI 9.6
9.5 Content Resolver 9.9
9.6 Content Values 9.13
9.7 Cursor 9.16
9.8 Permissions in Content Providers 9.18
9.9 Using Android Built-in Content Providers (Content Provider Demo Application) 9.21
9.10 Creating Your Own Content Provider (Custom Content Provider) 9.27
9.11 Using Custom Content Provider 9.42
9.12 Review Questions 9.45

Chapter - 10 Networking & Creating Services 10.1 - 10.46

10.1 Introduction to Networking 10.2
10.2 Consuming Web Services Using HTTP 10.5
10.2.1 Importance of Web Services in Mobile Apps 10.5
10.2.2 Basic Steps in Consuming Web Services Using HTTP 10.7
10.2.3 Downloading Text 10.7
10.2.4 Downloading Image 10.12
10.2.5 Consuming and Parsing XML Web Services in Android 10.15
10.2.6 Consuming JSON Web Services in Android 10.24
10.3 Creating Your Own Services and Binding Activities to Service 10.32
10.4 Understanding Threads 10.41
10.5 Local Data Storage in Android - Read and Write Local Data, Accessing Internal File System Accessing SD Card 10.44
10.6 Preparing App for Publishing 10.45
10.7 Deploying APK Files 10.45
10.8 Uploading in Market 10.45
10.3 Review Questions 10.45

Appendix - A Lab Programs A.1 - A.2

Appendix - B Model Question Papers B.1 - B.4

Model Question Paper - 1 B.1
Model Question Paper - 2 B.2
Model Question Paper - 3 B.3
Model Question Paper - 4 B.4

UNIT - I

CHAPTER 1

INTRODUCTION TO MOBILE TECHNOLOGIES AND KEY SERVICES

CONTENTS

- Introduction to Mobile Application
- Brief History of Mobile Technologies
- Different Mobile Technologies
- Key Mobile Application Services
- Review Questions

1.1 Introduction

Mobile application development has revolutionized the way we interact with technology and each other. With the advent of smartphones and tablets, mobile apps have become integral to our daily lives, by providing us with instant access to information, communication, entertainment, and services. This chapter aims to introduce the fundamental concepts of mobile technologies that drive this dynamic field. Understanding the basics is crucial to creating innovative and user-friendly applications.

Mobile technologies include a broad spectrum of hardware, software, and communication protocols that enable the development and deployment of applications on mobile devices. These technologies include operating systems such as iOS (using Swift) and Android (Using Kotlin or Java), development frameworks like React Native and Flutter, and communication standards such as 4G and 5G networks. Mobile applications leverage these technologies to deliver seamless user experiences, leveraging features like geolocation, push notifications, and in-app purchases. Understanding and effectively utilizing these technologies are essential for crafting engaging and responsive mobile applications that cater to the evolving needs of users in today's mobile-driven world.

Mobile application development requires expertise in mobile technologies, programming languages, design principles, and an understanding of the mobile ecosystem. Developers need to stay updated with the latest trends and best practices in mobile app development to create successful and competitive applications that meet user needs and expectations.

What is Mobile App ?

A mobile application, commonly referred to as a mobile app, is a software application designed to run on mobile devices such as smartphones and tablets. These applications are developed specifically for mobile platforms and are typically downloaded and installed from app stores or marketplaces like the Apple App Store for iOS devices and Google Play Store for Android devices.

Example : An example of a mobile application is Instagram, a popular app for sharing photos and videos with friends and followers. Users can upload images and videos, apply filters, and share them on their profile or with a selected audience.

What is Mobile Application Development?

Mobile application development is the process of creating software applications that run on mobile devices like smartphones and tablets.

Who is the father of Android?

The "father of Android" is considered to be Andy Rubin. He co-founded Android Inc. in 2003, which was later acquired by Google in 2005. Andy Rubin played a key role in the development of the Android operating system, which has since become the most widely used mobile operating system in the world.

What is Web App ?

A web app, or web application, is a software program that runs on a web server and is accessed through a web browser over the internet.

Example : Popular examples of web apps include email services like Gmail, social media platforms like Facebook, online shopping websites like Amazon, and productivity tools like Google Docs.

Web Apps Vs Mobile Apps

Web apps and mobile apps are tools for delivering digital experiences, but they cater to different platforms and user needs. Web apps are accessible through web browsers and use standard web technologies. Mobile apps, are developed specifically for mobile operating systems like Android and iOS. It offers enhanced performance and deeper integration with device features. Each type has its strengths and limitations, which influence their suitability for different use cases.

This table outlines the key differences and characteristics of web apps and mobile apps, highlighting their respective advantages and limitations.

Aspect	Web Apps	Mobile Apps
Platform	Accessed through web browsers	Developed for specific mobile OS (Android, iOS)
Technologies Used	HTML, CSS, JavaScript	Native languages (Java, Kotlin, Swift, Objective-C)
Installation	No installation required, accessed via URL	Must be downloaded and installed from app stores
Accessibility	Platform-independent, accessible on any device with a browser	Platform-specific, limited to compatible devices
Performance	May have limitations in accessing device features, potentially slower	Can fully leverage device capabilities, generally faster
User Experience	Consistent across different devices, limited to browser capabilities	Optimized for specific devices, provides a richer user experience
Device Features	Limited access to device-specific features	Full access to device-specific features (camera, GPS, etc.)
Offline Availability	Limited, relies on browser caching	Can work offline using local storage and native capabilities
Updates	Updates are immediate and handled server-side	Updates require user action to download and install
Distribution	Easily accessible via URL, no app store restrictions	Distributed through app stores (Google Play, App Store), subject to approval
Development Cost	Generally lower due to single codebase	Higher due to the need for platform-specific versions
Security	More vulnerable to security issues via the browser	More secure due to platform-specific security measures
Integration with System	Limited integration with device's operating system and other apps	Deep integration with the device's OS and other apps

Stages of Mobile Application Development

The Mobile application development is the process of creating software applications that run on mobile devices like smartphones and tablets. These apps are designed to provide specific functions or services to users. Developing a mobile app involves several stages-

1. **Planning:** Define app purpose, target audience, and goals. Conduct market research and create a project plan.
2. **Design:** Create UI/UX design for intuitive user experience. Use wireframes and prototypes for feedback.
3. **Development:** Write code using languages like Java, Swift, or Kotlin. Use tools like Android Studio or Xcode for frontend and backend development.
4. **Testing:** Conduct rigorous testing for bugs and issues. Test functionality, performance, and compatibility on different devices.
5. **Deployment:** Publish app to app stores like Apple App Store or Google Play Store following platform guidelines.
6. **Maintenance and Updates:** Monitor app performance, gather user feedback, and release updates for improvements and new features.

To create a mobile app, developers use programming languages like Java, Swift, or Kotlin, along with development tools and platforms like Android Studio or Xcode. They also consider factors like user interface (UI) design, performance, security, and compatibility with different devices and operating systems. Mobile apps can offer a wide range of functionalities, from simple games and social media platforms to complex productivity tools and e-commerce platforms. The popularity of mobile devices has led to a high demand for mobile apps, making mobile application development an important and rapidly evolving field in the tech industry.

1.2 Brief History of Mobile Technologies

The history and progress of mobile technologies have been defined by significant milestones that have revolutionized how we communicate and engage with the world. From the early stages of mobile communication to the advanced smartphones of today, mobile technologies have consistently progressed, driven by innovation and the increasing needs of users.

- **Early Beginnings:**
  - The journey of mobile technology began in the early 20th century with the invention of radio communication. The first real step towards mobile communication came with the development of mobile radio telephones in the 1940s. It was primarily used in vehicles with limited coverage and functionality.
- **First Generation (1G):**
  - The introduction of the first-generation (1G) mobile networks in the 1980s marked the true beginning of mobile telephony. 1G networks were analog and allowed for voice communication through mobile phones.
  - The launch of the Motorola DynaTAC 8000X in 1983, the first commercially available mobile phone, was a landmark event despite its large size and limited battery life.

- **Second Generation (2G):**
  - The 1990s saw the advent of the second-generation (2G) mobile networks, which were digital and offered significant improvements in voice quality and capacity.
  - 2G networks introduced the Global System for Mobile Communications (GSM) and it became the standard for mobile communication worldwide.
  - This era also saw the introduction of Short Message Service (SMS) by enabling text messaging between mobile devices.
- **Third Generation (3G):**
  - The early 2000s brought the third-generation (3G) mobile networks to enable faster data transmission and improved internet access on mobile devices.
  - 3G networks supported multimedia messaging, video calls, and mobile internet browsing,. It has shown the way for the development of more advanced mobile applications and services.
  - Technologies like Universal Mobile Telecommunications System (UMTS) and CDMA2000 expanded mobile capabilities beyond voice calls.
- **Fourth Generation (4G) and the Rise of Smartphones:**
  - The introduction of fourth-generation (4G) mobile networks in the late 2000s revolutionized mobile technology with high-speed internet access to enable seamless streaming of high-definition video, faster downloads, and enhanced mobile gaming experiences.
  - Long-Term Evolution (LTE) became the dominant 4G standard by offering significantly faster data speeds and lower latency.
  - This period also witnessed the rise of smartphones, integrating powerful processors, high-resolution displays, and a wide array of sensors, transforming them into versatile tools for communication, entertainment, and productivity.
- **Fifth Generation (5G) and Beyond:**
  - 5G represents the latest advancement in mobile technology, promising ultra-fast speeds, low latency, and massive connectivity to support emerging technologies like Internet of Things (IoT), augmented reality (AR), and autonomous vehicles.
  - 5G is set to revolutionize industries by enabling smart cities, autonomous vehicles, and advanced healthcare solution.

Do You Know?

- **First Mobile Phone Call (1973) :** Martin Cooper made the first mobile phone call using a prototype of the Motorola DynaTAC 8000X in 1973, leading to the commercial release of the device in 1983.
- **Nokia 3310 (2000) :** The Nokia 3310, released in 2000, gained fame for its durability and long battery life, selling over 126 million units worldwide and remaining a beloved choice for users.
- **First Smartphone - IBM Simon Personal Communicator (1992) :** The IBM Simon Personal Communicator, introduced in 1992, is considered the first smartphone with touchscreen capabilities, email and fax features, and basic applications.
- **Android Operating System (2003 - Present) :** Originally developed by Android Inc. and acquired by Google in 2005, the Android OS debuted in 2008 with the HTC Dream (T-Mobile G1), becoming a dominant force in the global smartphone market.

1.3 Different Mobile Technologies

Mobile technologies include a broad spectrum of innovations, ranging from fundamental communication networks to sophisticated software and hardware solutions. A comprehensive understanding of these technologies is essential for creating modern mobile applications that prioritize efficiency, security, and user-friendliness. The future of mobile technology appears promising, with ongoing advancements facilitating deeper integration and unlocking new possibilities across diverse fields. The overview of the key mobile technologies are listed below.

1.3.1 Cellular Networks

Cellular networks are the foundation of modern mobile communication. It enabled us to make phone calls, send text messages, and access the internet from our mobile devices. Cellular networks have undergone significant transformations since their inception. It has evolved through multiple generations to meet the increasing demands for better performance, connectivity, and new services. Each generation has introduced groundbreaking technologies and standards that have revolutionized mobile communication to enable everything from simple voice calls to high-speed internet and immersive digital experiences.

1. **1G (First Generation) :** Introduced in the 1980s, 1G was the first analog cellular network standard that enabled basic voice communication. Despite its limitations such as poor sound quality and limited capacity, it laid the foundation for mobile telephony. A notable technology of 1G was AMPS (Advanced Mobile Phone System), which used analog signals and FDMA to facilitate communication.
2. **2G (Second Generation) :** Rolled out in the early 1990s, 2G networks marked the shift from analog to digital communication, offering improved security, capacity, and sound quality. This generation introduced services like SMS and MMS, enhancing user communication. Key technologies included GSM (Global System for Mobile Communications) and CDMA (Code Division Multiple Access), which improved network efficiency and encryption.
3. **3G (Third Generation) :** Emerged in the early 2000s, 3G networks provided faster data transmission speeds, enabling a broader range of mobile services beyond voice and text. Users could now access mobile internet, video calls, and multimedia content, setting the stage for the smartphone era. Technologies like UMTS and HSPA enhanced data rates and spectral efficiency.
4. **4G (Fourth Generation) :** Introduced in the late 2000s, 4G networks revolutionized mobile broadband with high-speed internet browsing, HD video streaming, and low latency. LTE became the dominant 4G technology, offering significant improvements in data rates and connectivity. WiMAX also played a role in providing high-speed wireless internet access.
5. **5G (Fifth Generation) :** The latest advancement in cellular technology, 5G offers ultra-fast data speeds, minimal latency, and massive device connectivity. With speeds exceeding 1 Gbps and latency as low as 1 millisecond, 5G supports a wide range of applications such as IoT, autonomous vehicles, and smart cities. NR (New Radio) is the key technology for 5G, designed to deliver high performance for advanced applications like real-time remote surgery and immersive AR experiences.

1.3.2 Mobile Operating Systems (OS)

Mobile Operating Systems (OS) are the software platforms that manage the operations of mobile devices such as smartphones, tablets, and smartwatches. Mobile operating systems serve as the backbone of smartphones by providing the essential software framework that enables devices to function, run applications, and connect to services. These OS provide the basic functions necessary for the device to operate and for users to interact with the device and its applications.

Some Key Mobile Operating Systems:

- **Android:** Developed by Google, Android is the most widely used mobile operating system globally. It is based on the Linux kernel and is known for its open-source nature, allowing manufacturers to customize it for their devices. Android has a vast ecosystem of apps available through the Google Play Store.
- **iOS:** Developed by Apple, iOS is the operating system used exclusively on Apple's mobile devices including the iPhone, iPad, and iPod Touch. iOS is known for its smooth performance, security features, and tight integration with Apple's hardware and software ecosystem. Apps for iOS devices are available through the Apple App Store.
- **Others :** While Android and iOS dominate the mobile operating system globally, there are other operating systems that have been used in mobile devices. These include now-discontinued platforms like Windows Mobile and BlackBerry OS, as well as niche platforms like Tizen and HarmonyOS.
  - **Windows Mobile:** Once a popular choice for business users due to its integration with Microsoft Office and Exchange Server, Windows Mobile has been discontinued. Microsoft shifted its focus to developing applications for Android and iOS.
  - **BlackBerry OS:** Known for its strong security features and physical keyboards, BlackBerry OS was a favorite among enterprise or business users. However, it has been largely phased out, with BlackBerry devices now running on Android.
  - **Tizen:** An open-source operating system developed by the Linux Foundation and supported by Samsung. Tizen is used in a range of Samsung devices including smartwatches, TVs, and some smartphones, though it has not achieved widespread adoption.
  - **HarmonyOS:** Developed by Huawei, HarmonyOS is designed to work across a variety of devices, including smartphones, tablets, wearables, and IoT devices. It represents Huawei's effort to reduce its dependency on Android amid geopolitical challenges.

1.3.3 Mobile Development Frameworks

Mobile Development Frameworks play a crucial role in the creation of mobile applications, offering developers a variety of tools and approaches to build high-quality apps. These frameworks can be broadly categorized into Native Development, Cross-Platform Development and Hybrid Development.

1. Native Development

Native development involves creating applications for a specific platform using the official programming languages and tools provided by the platform. This approach typically results in better performance, tighter integration with the device's hardware and operating system, and a more refined user experience.

- **Languages and Tools:**
  - **Swift (iOS):** A powerful programming language developed by Apple for building apps for iOS, macOS, watchOS, and tvOS. Swift offers modern syntax, safety features, and high performance.
  - **Kotlin/Java (Android):** Kotlin is a modern and user-friendly programming language that works seamlessly with Java for Android development. It is increasingly popular for Android apps, although Java is still commonly used and well-supported.
- **Development Environments:**
  - **Xcode (iOS):** Xcode is Apple's integrated development environment (IDE) for macOS. It includes a suite of tools for developing software for iOS, macOS, watchOS, and tvOS. Xcode provides a robust set of features, including a source code editor, debugging tools, and performance analysis.
  - **Android Studio (Android):** Android Studio is the official IDE for Android development, based on IntelliJ IDEA. It offers a comprehensive set of tools for building, testing, and debugging Android apps. Features include a visual layout editor, APK analyzer, and real-time profilers.
- **Advantages of Native Development:**
  - **Performance:** Native apps generally perform better as they are optimized for the specific platform they are built for.
  - **Access to Device Features:** Native development allows developers to fully leverage device-specific features such as the camera, GPS, and sensors.
  - **User Experience:** Native apps offer a more seamless and intuitive user experience, as they adhere to the platform's design guidelines and standards.
- **Examples:**
  - **iOS Apps:** Instagram, WhatsApp (iOS versions)
  - **Android Apps:** Spotify, Twitter (Android versions)

2. Cross-Platform Development

Cross-platform development frameworks allow developers to build applications for multiple platforms using a single codebase. This approach can significantly reduce development time and costs as the same code can be deployed on both iOS and Android.

- **Popular Frameworks:**
  - **React Native:** Developed by Facebook, React Native enables developers to build mobile apps using JavaScript and React. It allows for the creation of rich, native-like user interfaces and offers a robust ecosystem of libraries and tools.
  - **Flutter:** Developed by Google, Flutter uses the Dart programming language to build natively compiled applications for mobile, web, and desktop from a single codebase. Flutter is known for its fast development cycles, expressive UI, and high performance.

- **Xamarin:** Owned by Microsoft, Xamarin allows developers to build cross-platform apps using C# and the .NET framework. Xamarin.Forms, a part of Xamarin, provides a way to build user interfaces that can be shared across platforms while allowing access to native APIs.

- **Advantages of Cross-Platform Development:**
  - **Cost Efficiency:** Developing a single codebase for multiple platforms reduces development and maintenance costs.
  - **Faster Time to Market:** Cross-platform frameworks enable quicker deployment across multiple platforms, allowing for faster time to market.
  - **Consistency:** A single codebase ensures a consistent user experience across different platforms.
- **Examples:**
  - **React Native Apps:** Facebook, Instagram, Airbnb
  - **Flutter Apps:** Google Ads, Alibaba, Reflectly
  - **Xamarin Apps:** Slack (some parts), Insightly, FreshDirect

3. Hybrid Development

Hybrid development frameworks combine elements of both native and web applications. Hybrid apps are typically built using web technologies such as HTML, CSS, and JavaScript, and then wrapped in a native container that allows them to run on both iOS and Android devices.

- **Popular Frameworks:**
  - **Ionic:** A framework that uses web technologies to build cross-platform mobile applications. Ionic apps are built using HTML, CSS, and JavaScript, and can be wrapped in a native container using tools like Cordova or Capacitor.
  - **Cordova:** An open-source mobile development framework that enables developers to use web technologies to build mobile applications. Cordova provides a set of APIs that allow web apps to interact with device hardware.
- **Advantages of Hybrid Development:**
  - **Ease of Development:** Developers can leverage their existing web development skills to build mobile apps.
  - **Single Codebase:** Similar to cross-platform frameworks, hybrid development allows for a single codebase to be deployed across multiple platforms.
  - **Access to Device Features:** Hybrid frameworks provide plugins and APIs that enable access to native device features.
- **Examples:**
  - **Ionic Apps:** MarketWatch, Amtrak, Sworkit
  - **Cordova Apps:** Adobe PhoneGap, Untappd, Localeur

Choosing the right mobile development framework depends on various factors, including project requirements, budget, development timeline, and the desired user experience. Native development offers the best performance and integration with platform-specific features, while cross-platform and hybrid development frameworks provide cost efficiency and faster time to market. Understanding the strengths and limitations of each approach helps in making informed decisions to meet the specific needs of your mobile application project.

1.3.4 Mobile Web Technologies

Mobile Web Technologies encompass a range of tools and standards that enable the development of web-based applications optimized for mobile devices. Mobile web technologies play a crucial role in delivering content and functionality to users on mobile devices. These technologies enable developers to create responsive and interactive web applications that provide a seamless user experience across different devices and screen sizes. Two key technologies in this domain are HTML5 and Progressive Web Apps (PWAs):

1. **HTML5 :** HTML5 is the latest version of the Hypertext Markup Language (HTML) used for structuring and presenting content on the web. It introduces numerous features and improvements that enhance the capabilities of web applications, especially on mobile devices.

   Examples:

   - **Multimedia Websites:** Websites like YouTube and Vimeo utilize HTML5 to stream video content seamlessly on mobile devices.
   - **Interactive Applications:** HTML5 is used in various interactive applications such as online games and data visualization tools.

2. **Progressive Web Apps (PWAs):** Progressive Web Apps (PWAs) are web applications that offer a native app-like experience while being accessible through a web browser. PWAs combine the best features of web and mobile apps by providing functionalities such as offline access, push notifications, and home screen installation.

   Examples:

   - **Twitter Lite:** Twitter Lite is a PWA that offers a fast, data-friendly version of Twitter, providing offline access and push notifications.
   - **Pinterest:** Pinterest's PWA delivers a smooth browsing experience with fast loading times and offline functionality to ensure users can access content even with limited connectivity.

1.3.5 Connectivity and Communication

Connectivity and communication technologies are fundamental to the functionality and versatility of modern mobile devices. These technologies enable devices to connect and communicate with each other, exchange data, and interact with various networks and services. Let us look at some of the key connectivity and communication technologies:

1. **Bluetooth :** Bluetooth is a wireless technology standard used for exchanging data over short distances. It is designed to provide a secure and reliable communication channel between devices, typically within a range of 10 meters (33 feet). Bluetooth is widely used for connecting peripheral devices, sharing files, and enabling various wireless applications.

   Examples:

   - **Bluetooth 5.0:** This version offers faster data transfer rates (up to 2 Mbps), a longer range (up to 240 meters in open space), and improved capacity for connecting multiple devices simultaneously. It is commonly used in wireless headphones, fitness trackers, and smart home devices.

2. **Wi-Fi :** Wi-Fi is a wireless networking technology that allows devices to communicate with each other and access the internet without the need for direct cable connections. Wi-Fi networks use radio waves to transmit data to enable high-speed internet access and local network communication.

   Examples:

   - **Wi-Fi 6 (802.11ax):** Wi-Fi 6 offers faster speeds (up to 9.6 Gbps), better performance in congested environments, and improved energy efficiency. It is designed to handle the growing number of connected devices and provide a more reliable and consistent internet experience.

3. **NFC (Near Field Communication) :** NFC is a short-range wireless communication technology that enables devices to exchange data when they are brought within a few centimeters of each other. NFC is commonly used for contactless payments, data exchange, and pairing devices.

   Examples:

   - **Google Pay and Apple Pay:** These mobile payment systems use NFC technology to enable secure, contactless transactions. Users can make payments by holding their smartphones near a compatible payment terminal to perform a quick and convenient payment experience.

1.3.6 Sensors and Hardware Innovations

Mobile devices have evolved into powerful tools not only because of their software capabilities but also due to advanced sensors and hardware innovations. These components enhance the functionality, security, and user experience of mobile devices. Let us understand some key sensors and hardware innovations:

1. **Accelerometers and Gyroscopes:** Accelerometers and gyroscopes are essential sensors that detect orientation, movement, and acceleration in mobile devices. They work together to provide detailed information about the device's position and motion, enabling a variety of applications.

   Examples:

   - **Fitness Apps:** Apps like Google Fit and Apple Health use accelerometers and gyroscopes to monitor physical activity and provide fitness metrics.
   - **Gaming Apps:** Games like "Asphalt" and "Temple Run" use these sensors for intuitive and responsive control mechanisms.

2. **Biometric Sensors :** Biometric sensors enhance the security and functionality of mobile devices by using unique biological characteristics for authentication and health monitoring. These sensors provide a convenient and secure way to unlock devices, verify identity, and monitor health parameters.

   Examples:

   - **Apple's Face ID:** Uses a combination of infrared sensors and cameras to create a detailed 3D map of the user's face, allowing for secure and accurate facial recognition.
   - **Samsung's Heart Rate Monitor:** Integrated into Samsung's smartphones and smartwatches, this sensor measures the user's heart rate by detecting blood flow through the skin, providing real-time health data.

3. **Additional Sensors and Innovations**
   - **Proximity Sensors:** Detect the presence of nearby objects without physical contact. Proximity sensors are typically used to detect when the user is holding the device close to their face.
     - **Examples:** Used in smartphones to disable the touchscreen while on a call.
   - **Ambient Light Sensors:** Measure the amount of light in the surrounding environment. These sensors adjust the screen brightness automatically based on ambient lighting conditions.
     - **Examples:** Automatic brightness adjustment in smartphones and tablets.
   - **Infrared Sensors:** Detect infrared light and are often used in remote control functions.
     - **Examples:** Many Android phones have built-in infrared blasters for controlling home electronics.
   - **Thermal Sensors:** Measure the temperature of the device or the user's skin.
     - **Examples:** Used in high-end smartphones to manage thermal performance.

1.3.7 Location-Based Services (LBS)

Location-Based Services (LBS) utilize geographic data and location information to provide a variety of services and applications that enhance user experiences, improve operational efficiencies, and enable new functionalities. These services leverage technologies such as GPS, geofencing, and other location-tracking methods.

1. **Global Positioning System (GPS) :** The Global Positioning System (GPS) is a satellite-based system that provides precise location and time information anywhere on Earth. It consists of a network of satellites that transmit signals to GPS receivers, which then calculate their exact position based on the signals received from multiple satellites.

   Examples:

   - **Google Maps:** Uses GPS to provide turn-by-turn navigation, traffic updates, and location-based services like finding nearby restaurants and businesses.
   - **Apple Maps:** Offers similar navigation and location services, integrated with Apple's ecosystem, including Siri and Apple Watch.

2. **Geofencing :** Geofencing is a technology that uses GPS, RFID (Radio Frequency Identification), Wi-Fi, or cellular data to create virtual boundaries around a specific geographic area. When a device equipped with location services enters or exits these predefined boundaries, specific actions can be triggered.

   Examples:

   - **Location-Based Marketing:** Retailers like Starbucks and McDonald's use geofencing to send promotional offers to customers' mobile devices when they are near a store, to encourage visits and purchases.
   - **Smart Home Automation:** Systems like Apple HomeKit and Google Nest use geofencing to automate home settings such as turning on lights when approaching home or adjusting the thermostat based on the user's presence.

1.3.8 Cloud Storage

Cloud storage involves storing data on remote servers that can be accessed via the internet. It provides a convenient and reliable way to back up, synchronize, and share data across multiple devices and users. This service ensures that mobile users can access their files and data from anywhere, at any time.

- **Examples:**
  - **Google Drive:** Offers 15 GB of free storage with integration into Google's suite of services, allowing users to store files, collaborate on documents, and access data from any device.
  - **iCloud:** Apple's cloud storage service provides seamless integration with iOS and macOS devices, offering features like automatic photo backups, document synchronization, and device backups.
  - **Dropbox:** A popular cloud storage service that provides easy file sharing, collaboration tools, and cross-platform access, making it suitable for both personal and professional use.

1.3.9 Emerging Technologies

Emerging technologies are continually reshaping the landscape of mobile and digital experiences. These technologies enhance the capabilities of devices, create new forms of interaction, and open up possibilities for innovative applications. Let us understand some of the key emerging technologies:

1. **Augmented Reality (AR) :** Augmented Reality (AR) adds digital information such as images, videos, and interactive elements onto the real world. AR enhances the user's perception of reality by integrating virtual objects with the physical environment in real time.

   Examples:

   - **Pokémon GO:** This popular AR game overlays virtual Pokémon characters onto real-world locations by allowing players to capture and interact with them using their smartphones.
   - **IKEA Place:** An AR app that enables users to place virtual furniture in their homes to see how it fits and looks before purchasing.

2. **Virtual Reality (VR) :** Virtual Reality (VR) creates immersive experiences by transporting users into a fully digital environment. Using VR headsets, users can explore and interact with virtual worlds that provide a sense of presence and immersion.

   Examples:

   - **Oculus VR:** A leading VR platform that offers a range of VR headsets and experiences, including games, social interactions, and virtual tours.
   - **Google Cardboard:** An affordable VR solution that uses a smartphone and a cardboard viewer to provide basic VR experiences, making VR accessible to a wider audience.

3. **Internet of Things (IoT) :** The Internet of Things (IoT) refers to the network of everyday objects connected to the internet to enable them to send and receive data. IoT devices are embedded with sensors, software, and connectivity, allowing them to interact with each other and their environment.

Examples:

- **Smart Home Devices:** Devices like the Nest Thermostat, Ring Video Doorbell, and Philips Hue smart lights use IoT to provide enhanced home automation and security.
- **Wearable Health Monitors:** Devices like Fitbit and Apple Watch use IoT technology to track fitness metrics, monitor heart rate, and provide health insights, helping users maintain a healthy lifestyle.

Industrial IoT Solutions: Industrial IoT applications include predictive maintenance systems that monitor equipment health and performance, and smart factories that use connected devices to optimize production processes.

1.3.10 Artificial Intelligence (AI) in Mobile

Artificial Intelligence (AI) has become a transformative force in mobile technology to enable smarter, more intuitive, and more personalized user experiences. AI is integrated into mobile devices in various forms, from virtual assistants to machine learning algorithms to enhance the functionality and efficiency of mobile applications. Here's an in-depth look at AI in mobile:

1. **Virtual Assistants :** Virtual assistants are AI-powered applications that perform tasks and provide information based on voice commands. These assistants use natural language processing (NLP) and machine learning to understand and respond to user queries, offering a hands-free way to interact with mobile devices.

   Examples:

   - **Siri:** Apple's virtual assistant integrates seamlessly with iOS devices by offering features like voice-activated commands, smart home control, and personalized suggestions based on user behavior.
   - **Google Assistant:** Available on Android devices and smart speakers, Google Assistant provides comprehensive voice search capabilities, contextual understanding, and integration with Google services.
   - **Amazon Alexa:** Known for its robust smart home integration, Alexa powers Amazon Echo devices and offers a wide range of skills from playing music to managing smart home ecosystems.

2. **Machine Learning :** Machine learning involves implementing algorithms that allow mobile apps to learn from data and improve their performance over time. By analyzing user behavior and data patterns, machine learning models can make predictions, automate tasks, and enhance app functionalities.

   Examples:

   - **Predictive Text:** Mobile keyboards like Gboard and SwiftKey use machine learning to predict and suggest the next word or phrase as users type, speeding up text input and reducing errors.
   - **Image Recognition:** Apps like Google Photos and Apple Photos use machine learning to recognize faces, objects, and scenes in images to enable automatic organization and search capabilities.

- **Voice Recognition:** Voice assistants like Siri and Google Assistant use machine learning to improve their understanding of voice commands to provide more accurate and contextually relevant responses.

1.4 Key Mobile Application Services

Mobile applications offer a wide range of services to enhance user experience and functionality. Understanding these key mobile application services is crucial for developers to create successful and user-friendly apps. Mobile application services are essential components that support the functionality, performance, and user experience of mobile apps. These services provide the necessary infrastructure, tools, and features to build, deploy, and manage mobile applications efficiently. Let's understand each of these services and explore examples of how they are implemented in popular mobile applications.

1. **Push Notifications :** Push notifications are essential for engaging users and delivering timely information. They allow apps to send messages to users even when the app is not actively in use.

   Example: Push notifications are extensively used by social media platforms like Facebook and Instagram. These apps send notifications to users about new messages, likes, comments, and other interactions to ensure users are engaged and informed.

2. **In-App Messaging :** In-app messaging enables users to communicate within the app to enhance engagement and interaction among users.

   Example: Messaging apps like WhatsApp and Messenger provide in-app messaging services that allow users to send text, voice, image, and video messages to their contacts. It creates a seamless communication experience within the app.

3. **Analytics and Tracking :** Analytics services help developers track user behavior, app usage, and performance metrics to optimize the app's performance and user experience.

   Example: Google Analytics for Firebase is a powerful tool that tracks user engagement, retention, demographics, and in-app events. Developers use this data to gain insights into user behavior and make data-driven decisions to improve their apps.

4. **User Authentication :** User authentication services verify user identities to provide secure access to app features and data.

   Example: Google Sign-In and Facebook Login are popular authentication services that allow users to sign in to various apps using their Google or Facebook credentials. It simplifies the login process and enhancing security.

5. **Payment Processing :** Payment processing services enable secure in-app purchases, subscriptions, and financial transactions.

   Example: Apple Pay and Google Pay are widely used payment processing services that allow users to make purchases within apps securely and conveniently, enhancing the overall user experience.

6. **Cloud Storage :** Cloud storage services enable apps to store data on remote servers to facilitate data synchronization and backup across multiple devices.

Example: Google Drive and iCloud offer cloud storage solutions that allow users to store and sync documents, photos, and other files across their devices to ensure data accessibility and backup.

7. **Location Services :** Location services empower apps to provide location-based functionalities such as navigation, geotagging, and location-based notifications.

   Example: Apps like Google Maps and Uber utilize location services to offer navigation, real-time location tracking, and personalized services based on the user's location to enhance user experience and convenience.

8. **Social Media Integration :** Social media integration services enable apps to connect with social platforms, allowing users to share content and interact with their social networks seamlessly.

   Example: Social media apps like Instagram and TikTok integrate with platforms like Facebook and Twitter, enabling users to share photos, videos, and updates directly from the app to their social media accounts, enhancing user engagement and reach.

9. **Cloud Messaging :** Cloud messaging services facilitate sending messages and notifications to users across different devices and platforms.

   Example: Firebase Cloud Messaging (FCM) is a robust cloud messaging service that enables developers to send notifications and messages to users on iOS, Android, and web applications, ensuring effective communication and engagement.

10. **Data Synchronization :** Data synchronization services ensure that user data remains consistent and up-to-date across multiple devices and platforms.

    Example: Apps like Evernote and Microsoft OneNote utilize data synchronization to keep notes and documents updated across smartphones, tablets, and computers, allowing users to access their data seamlessly across devices.

11. **Voice Recognition :** Voice recognition services allow apps to convert spoken language into text, enabling voice commands and voice-activated features.

    Example: Siri (Apple) and Google Assistant use voice recognition to perform tasks and provide information based on user voice commands.

12. **Offline Access :** Offline access services enable users to access certain app features and content without an active internet connection, enhancing usability in low-connectivity environments.

    Example: Apps like Google Maps allow users to download maps for offline use, enabling navigation and location-based services even when users are offline or have limited connectivity, ensuring uninterrupted access to essential features.

13. **Personalization and Customization :** Personalization and customization services tailor the app experience to individual user preferences, providing a personalized and engaging user experience.

    Example: Music streaming apps like Spotify use personalization algorithms to recommend music based on user listening habits, preferences, and history, creating a customized music experience for each user and enhancing user engagement.

14. **Augmented Reality (AR) Integration :** AR integration services enable apps to overlay digital content and interactive experiences onto the real world, enhancing user engagement and interactivity.

    Example: Apps like Pokémon GO leverage AR technology to superimpose virtual creatures and gameplay elements onto the real-world environment, creating an immersive and interactive gaming experience for users.

15. **Offline Data Sync :** Offline data sync services synchronize app data with backend servers when the device is back online, ensuring data consistency and seamless user experience across devices.

    Example: Apps like Microsoft OneDrive use offline data sync to automatically update files and documents stored on the cloud when the device reconnects to the internet, allowing users to access the latest data seamlessly across devices.

16. **Biometric Authentication :** Biometric authentication services utilize unique biological characteristics such as fingerprints, facial recognition, or iris scans to verify user identities securely and conveniently.

    Example: Apps like banking applications and mobile wallets integrate biometric authentication features such as Touch ID (fingerprint recognition) or Face ID (facial recognition) on mobile devices to provide users with a secure and seamless login experience, enhancing security and user convenience.

17. **App Distribution and Deployment :** App distribution and deployment services provide platforms for distributing apps to users, managing updates, and tracking app performance.

    Example: Apple App Store and Google Play Store are the primary platforms for distributing iOS and Android apps, respectively.

1.5 Review Questions

Section - A

1. What is a Mobile Application?
2. What is Mobile Application Development?
3. Who is considered the father of Android?
4. What are web apps?
5. What are mobile apps?
6. What are the stages of mobile application development?
7. What is the role of mobile operating systems (OS)?
8. Name two popular cross-platform mobile development frameworks.
9. What is Bluetooth technology?
10. What is the function of GPS in mobile devices?
11. What are Progressive Web Apps (PWAs)?

12. What is a native development framework?
13. What is the primary advantage of native mobile app development?
14. What is a hybrid development framework?
15. What are geofencing technologies?
16. What is the purpose of cloud storage in mobile apps?
17. What are the features of augmented reality (AR) in mobile apps?
18. How does virtual reality (VR) differ from augmented reality (AR)?
19. What is the Internet of Things (IoT)?
20. What is the role of machine learning in mobile apps?
21. What are push notifications used for in mobile apps?
22. What are location-based services (LBS)?
23. What is the function of the accelerometer in mobile devices?
24. What is a virtual assistant?
25. What is biometric authentication in mobile devices?
26. Name any two Key Mobile Application Services
27. What is Cross Platform Mobile Development?
28. Write any two mobile operating systems.

Section - B

1. What is Mobile App? How it is different from Web App?
2. Explain the Stages of Mobile Application Development.
3. Explain the Brief History of Mobile Technologies.
4. Explain Mobile Operating Systems.
5. Write a note on
   (a) Connectivity and Communication Technologies
   (b) Location-Based Services (LES)
6. Write a note on
   (a) Artificial Intelligence (AI) in Mobile
   (b) Emerging Technologies.

Section - C

1. Explain Different Mobile Technologies.
2. Explain Key Mobile Application Services.
3. Explain Mobile Development Frameworks with examples.

UNIT - I

CHAPTER 2

INTRODUCING ANDROID

CONTENTS

- Introduction
- The Android Application Components
- Exploring the Development Environment
- Obtaining the Required Tools
- Installing Android Studio
- Your First Android Application - Hello World
- Exploring the IDE (Android Studio)
- Debugging your Android Application in Android Studio
- Publishing your Android Application
- Review Questions

2.1 Introduction

In the past, when discussing operating systems, the common choices were Windows, Linux, and macOS. However, the mobile phone market saw huge competition and Android quickly emerged as a dominant player. Android is an operating system for mobile devices and it is one of the most popular mobile platforms today. Android platform runs on hundreds of millions of mobile devices throughout the world. It's the largest installed operating system of any mobile operating system and growing rapidly day by day. We can develop apps and games using Android and it gives us an open marketplace for distributing apps and games instantly.

Android is a mobile operating system based on a modified version of the Linux kernel and open-source software. It is tailored for touch screen devices like smartphones and tablets. Google has expanded the Android ecosystem to include Android TV for televisions, Android Auto for cars, and Wear OS for smartwatches, each with a specialized user interface. Variants of Android are also found in game consoles, digital cameras, PCs, and other electronics.

2.1.1 Brief History of Android

- Android Inc. was founded in October 2003 by Andy Rubin, Rich Miner, Nick Sears, and Chris White with the aim of developing smarter mobile devices that are more aware of the user's location and preferences.
- In August 2005, Google acquired Android Inc., and Andy Rubin, along with the team, continued the development of the Android operating system under Google's umbrella with the support of the Open Handset Alliance (OHA).
- The Open Handset Alliance (OHA) was formed in November 2007, comprised of many of the largest and most successful mobile companies on the planet. Its members include chip makers, handset manufacturers, software developers, and service providers. The entire mobile supply chain is well represented.
- The first commercial version of Android, Android 1.0, was released on September 23, 2008, with the HTC Dream (also known as the T-Mobile G1) being the first device to run the new OS. This version introduced basic features like a web browser, camera support, and access to the Android Market (now Google Play Store).
- Android 1.0 laid the foundation for the widespread adoption of the Android platform and marked the beginning of a new era in mobile technology.
- After the initial release of Android 1.0 in 2008, the operating system has continuously evolved through various versions, reaching up to Android 14.0, each bringing new features, enhancements, and improvements to the platform.
- Android 14 is the fourteenth major version of the Android operating system. Google announced Android 14 on February 8, 2023.
- Google hosts the Android open source project and provides online Android documentation, tools, forums, and the Software Development Kit (SDK) for developers. All major Android news originates at Google.

2.1.2 What is Android ?

Android is an open-source operating system developed by Google, designed primarily for touchscreen mobile devices such as smartphones and tablets. Since its inception, Android has become the most widely used mobile operating system globally, powering billions of devices. It is known for its flexibility, extensive app ecosystem, and seamless integration with Google services, Android offers a robust platform for both users and developers. It supports a wide range of devices including smartwatches, TVs, and car infotainment systems, making it a versatile and comprehensive mobile operating system. Android's open-source nature allows for significant customization and innovation, fostering a diverse and dynamic mobile technology landscape.

What is Android ?

Android is described as "the **first complete, open, and free mobile platform**":

- **Complete:** It allows for rich application development opportunities.
- **Open:** It is provided through open-source licensing.
- **Free:** Android applications are free to develop, and they can be distributed and commercialized in various ways.

Android is an open-source software stack that includes the operating system, middleware, and key mobile applications. It provides a platform for developing innovative applications for various devices.

Different Definitions of Android

- **Andy Rubin's Definition:**
  Andy Rubin, the co-founder of Android Inc. and one of the creators of the Android operating system, describes Android as a platform that aims to bring the power of the internet into the hands of more people. He envisions Android as an open-source platform that allows developers to create innovative applications that can enhance the user experience on mobile devices.
- **Google's Description:**
  Google, the current owner of the Android platform, describes Android as a mobile operating system that is based on a modified version of the Linux kernel and other open-source software. It is designed primarily for touchscreen mobile devices such as smartphones and tablets.

2.1.3 Key Features of Android

Android, the popular operating system developed by Google, offers a wide range of key features and benefits that make it a preferred choice for mobile device manufacturers, developers, and users. In this detailed explanation, we will explore some of the key features and benefits of Android, along with examples to illustrate their significance.

1. **Open-Source Platform:** Android is based on the Linux kernel and is released under the Apache License, allowing manufacturers and developers to modify and distribute the software freely.

Example: Custom ROMs like LineageOS and Paranoid Android offer alternative Android experiences with additional features and optimizations.

2. **Customizable User Interface:** Android allows users to customize their home screens, widgets, and themes to personalize their experience.

   Example: Users can add weather widgets, change wallpapers, and use custom launchers like Nova Launcher to modify the look and feel of their device.

3. **Google Services Integration:** Android devices are tightly integrated with Google services such as Gmail, Google Maps, Google Drive, and Google Assistant.

   Example: Users can access Google Maps for navigation, store files in Google Drive, and use Google Assistant for voice commands and smart home control.

4. **Extensive App Ecosystem:** The Google Play Store offers millions of apps covering various categories, including productivity, entertainment, social networking, and gaming.

   Example: Popular apps like WhatsApp, Instagram, Spotify, and Netflix are readily available for download from the Play Store.

5. **Multi-Device Support:** Android powers a variety of devices beyond smartphones and tablets, including smartwatches (Wear OS), smart TVs (Android TV), car infotainment systems (Android Auto), and home appliances.

   Example: Devices like the Samsung Galaxy Watch use Wear OS, while smart TVs from brands like Sony and NVIDIA use Android TV.

6. **Google Assistant and Voice Commands:** Android devices come with Google Assistant, a virtual assistant that can perform tasks, answer questions, and control smart home devices through voice commands.

   Example: Users can say "Hey Google" to set reminders, send texts, play music, or adjust smart thermostats and lights.

7. **Multi-Tasking and Split-Screen Mode:** Android supports multitasking and split-screen mode, allowing users to run and view two apps simultaneously.

   Example: Users can watch a YouTube video while browsing the web or use a note-taking app while reading an e-book.

8. **Notifications and Quick Settings:** Android provides a rich notification system with actionable notifications and quick settings for easy access to frequently used functions.

   Example: Users can reply to messages directly from the notification shade and toggle settings like Wi-Fi, Bluetooth, and Do Not Disturb.

9. **Advanced Connectivity Options:** Android supports various connectivity options such as NFC, Bluetooth, Wi-Fi Direct, and USB OTG.

   Example: Users can make contactless payments using Google Pay (NFC), share files with nearby devices using Bluetooth, and connect USB peripherals like keyboards and flash drives using USB OTG.

10 **Regular Updates and Security:** Google regularly releases updates to the Android operating system by introducing new features, performance improvements, and security patches.

   Example: The Google Pixel series receives timely updates directly from Google to ensure the latest features and security enhancements are available.

11. **Storage:** Android uses SQLite, a lightweight relational database for local data storage. This allows applications to store and manage complex data structures efficiently on the device.

    Example: A note-taking app can use SQLite to store and retrieve user notes.

12. **Connectivity:** Android supports various connectivity options, including GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth (including A2DP and AVRCP profiles), WiFi, LTE, and WiMAX.

    Example: Users can connect to the internet via WiFi or mobile networks, and pair their devices with Bluetooth accessories like wireless headphones.

13. **Messaging:** SMS and MMS: Android supports both SMS (Short Message Service) and MMS (Multimedia Messaging Service) for text and multimedia communication.

    Example: Users can send and receive text messages, photos, videos, and audio messages.

14. **Media Support:** Android includes support for various media formats, including H.263, H.264, MPEG-4 SP, AMR, AMR-WB, AAC, HE-AAC, MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF, and BMP.

    Example: Users can play music, watch videos, and view images in different formats.

15. **Hardware Support:** Android supports various hardware sensors, including an accelerometer, camera, digital compass, proximity sensor, and GPS.

    Example: Fitness apps can use the accelerometer to track steps, and navigation apps can use GPS for location tracking.

16. **Multi-Touch:** Android supports multi-touch screens, allowing for intuitive gestures like pinch-to-zoom and multi-finger swiping.

    Example: Users can zoom in on photos or maps using two-finger gestures.

17. **Tethering:** Android supports tethering, allowing users to share their internet connection via wired (USB) or wireless (WiFi hotspot) methods.

    Example: Users can turn their Android device into a portable WiFi hotspot to provide internet access to other devices.

18. **Web Browser:** Android's web browser is based on the open-source WebKit engine and incorporates Chrome's V8 JavaScript engine for fast and efficient web browsing.

    Example: The built-in browser allows users to surf the internet, access web applications, and view multimedia content.

19. **Google Play Store:** Android users have access to the Google Play Store, a vast repository of apps, games, movies, music, and books. The Play Store offers a diverse range of content for users to download and enjoy on their devices. Developers can publish their apps on the Play Store to reach a global audience.

    Example : Apps like WhatsApp, Instagram, and Spotify are available on the Google Play Store. Users can download WhatsApp for messaging, Instagram for social networking and photo sharing, and Spotify for music streaming.

20. **Adaptive Battery:** Adaptive Battery uses machine learning to optimize battery usage based on user habits and app usage patterns.

    Example: The system prioritizes battery power for frequently used apps and limits the battery consumption of less frequently used apps, extending overall battery life.

2.1.4 Mobile Application Development Using Android

Mobile application development using Android refers to the process of creating software applications that run on mobile devices by using the Android operating system. Developers use various tools, programming languages, and resources to design, build, test, and deploy Android applications for smartphones, tablets, smartwatches, smart TVs, and other Android-powered devices.

Key aspects of Android application development include:

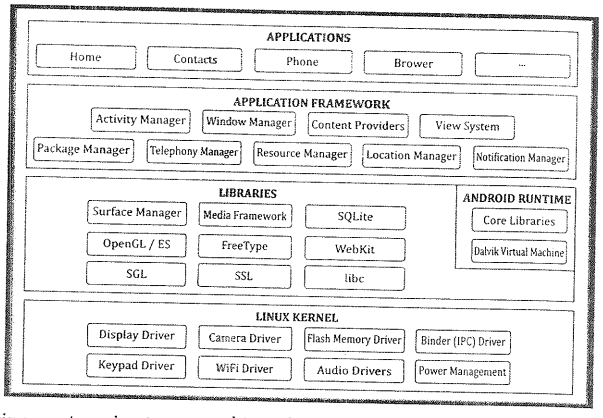
1. **Programming Languages:** Java has traditionally been the primary language for Android app development. However, with the introduction of Kotlin as an official language for Android development, developers now have the option to choose between Java and Kotlin. Kotlin offers modern features, concise syntax, and enhanced safety compared to Java.
2. **Development Tools:** Android Studio is the official Integrated Development Environment (IDE) for Android app development. It provides a comprehensive set of tools for coding, debugging, testing, and deploying Android applications. Android Studio offers features like code templates, visual layout editor, and integration with the Android SDK.
3. **Android SDK:** The Android Software Development Kit (SDK) includes libraries, tools, and APIs necessary for building Android applications. Developers use SDK components to access device features, interact with the operating system, and create engaging user interfaces.
4. **User Interface Design:** Android app developers focus on creating intuitive and visually appealing user interfaces that adhere to Google's Material Design guidelines. Design elements such as layouts, views, animations, and themes play a crucial role in enhancing the user experience.
5. **Testing and Debugging:** Testing is a critical phase in Android app development to ensure the application functions correctly across different devices and screen sizes. Developers use tools like Android Debug Bridge (ADB), Android Emulator, and third-party testing frameworks to test and debug their applications.
6. **Publishing:** Once the app is developed and tested, developers can publish it on the Google Play Store or other app distribution platforms. Publishing involves creating app listings, setting pricing and distribution options, and complying with store guidelines and policies.
7. **Continuous Updates and Maintenance:** Android developers need to regularly update their applications to add new features, fix bugs, enhance performance, and ensure compatibility with the latest Android versions. Continuous maintenance is essential to provide users with a seamless and up-to-date experience.

What are Android Devices ?

Android devices refer to hardware products that run on the Android operating system developed by Google. These devices come in various forms, including smartphones, tablets, smartwatches, smart TVs, Android Automotive Systems and other gadgets.

2.1.5 Architecture of Android

In order to understand how Android works, take a look at below architectural diagram, which shows the various layers that make up the Android operating system (OS).



The Android operating system is structured into five sections across four main layers:

1. **Linux Kernel:** At the base of the Android OS is the Linux kernel, which serves as the foundation for the system. This layer includes low-level device drivers that facilitate communication between the hardware components of an Android device and the software.
2. **Libraries:** The libraries layer contains code that provides essential functionalities of the Android OS. For example, the SQLite library offers database support for storing data, while the WebKit library enables web browsing capabilities on Android devices.
3. **Android Runtime:** Positioned alongside the libraries, the Android runtime consists of core libraries that empower developers to create Android applications using the Java programming language. The Android runtime also incorporates the Dalvik virtual machine, which allows each Android app to operate in its own process with a dedicated instance of the Dalvik virtual machine. Dalvik is optimized for mobile devices with limited resources like memory and CPU.
4. **Application Framework:** This layer exposes the various features and capabilities of the Android OS to application developers. It provides a framework that developers can utilize to access system functions and services for their applications.
5. **Applications:** The top layer of the Android OS comprises pre-installed applications that come with the device, such as Phone, Contacts, and Browser. Additionally, it includes applications that users download and install from sources like the Google Play Store. Any custom applications developed by users or third-party developers also reside in this layer.

This layered architecture of the Android OS ensures a structured and organized approach to system functionality, development, and application management. Each layer plays a crucial role in the overall operation and user experience of Android devices.

2.1.6 Native Android Application

A native Android application refers to an app developed specifically for the Android operating system using the official Android development tools and programming languages. These apps are written in languages like Java or Kotlin and are compiled directly into machine code for the target device's processor architecture.

Characteristics of Native Android Applications

1. **Performance:** Native apps are optimized for the Android platform to provide high performance and responsiveness. They can directly access device-specific features and hardware, resulting in smooth user experiences.
2. **Access to Native APIs:** Native Android apps have full access to the Android SDK and native APIs, allowing developers to leverage the platform's capabilities for features like camera access, GPS, sensors, and more.
3. **User Experience:** Native apps can provide a consistent and intuitive user interface that aligns with Android design guidelines. This leads to better user engagement and satisfaction.
4. **Offline Functionality:** Native Android apps can offer offline functionality by storing data locally on the device to enable users to access content and features even without an internet connection.
5. **Integration with Platform Features:** Native apps can seamlessly integrate with other Android system features, such as notifications, sharing options, authentication methods, and in-app purchases.
6. **Optimized for Different Screen Sizes:** Native Android apps can adapt to various screen sizes and resolutions to ensure a consistent user experience across different devices, including smartphones and tablets.
7. **Security:** Native apps can implement security features provided by the Android platform, such as data encryption, secure network communication, and permissions management to protect user data.
8. **Distribution:** Native Android apps can be published on the Google Play Store, making them easily accessible to a vast audience of Android users worldwide.

Developing native Android applications requires knowledge of Java or Kotlin programming languages, the Android SDK, and development tools like Android Studio. While native apps offer numerous advantages in terms of performance and user experience, they may require more development time and effort compared to cross-platform or web-based alternatives. However, for apps that demand high performance, access to device-specific features, and a seamless user experience on Android devices, native development is often the preferred choice.

2.1.7 Advantages and Disadvantages of Android

Advantages of Android

1. **Open-Source Platform:** Android is an open-source platform, allowing developers to customize the operating system according to their needs. This flexibility encourages innovation and the development of a wide range of applications.
2. **Large User Base:** Android has a massive user base globally, making it an attractive platform for developers to reach a broad audience with their apps and services.
3. **Customization:** Android offers extensive customization options for users, allowing them to personalize their devices with widgets, themes, and third-party apps to suit their preferences.
4. **Google Ecosystem Integration:** Android seamlessly integrates with Google services such as Gmail, Google Drive, Google Maps, and Google Assistant, providing a cohesive user experience across devices.

5. **Hardware Diversity:** Android runs on a wide range of devices from various manufacturers, offering users a diverse selection of smartphones, tablets, and other gadgets at different price points and with varying features.
6. **App Ecosystem:** The Google Play Store hosts a vast number of apps, games, and utilities, providing users with a rich selection of software to enhance their device functionality.
7. **Multitasking and Productivity:** Android devices support multitasking features, allowing users to run multiple apps simultaneously and boost productivity with features like split-screen mode and floating apps.

Disadvantages of Android

1. **Fragmentation:** Android fragmentation refers to the diversity of devices running different versions of the operating system, which can lead to compatibility issues for developers and users.
2. **Security Concerns:** Due to its open nature, Android is more susceptible to malware and security threats compared to closed operating systems like iOS. Users need to be cautious when downloading apps from third-party sources.
3. **Software Updates:** Android updates are often delayed or inconsistent across devices and manufacturers, leading to a fragmented ecosystem with devices running outdated software versions.
4. **Performance Variability:** The performance of Android devices can vary based on hardware specifications, software optimization, and manufacturer customizations, leading to inconsistencies in user experience.
5. **Bloatware:** Some Android devices come pre-installed with bloatware or unnecessary apps from manufacturers and carriers, which can consume storage space and affect device performance.
6. **User Interface Fragmentation:** Android devices may have different user interfaces and customization layers (such as Samsung's One UI or Xiaomi's MIUI), leading to inconsistencies in design and user experience across devices.
7. **Privacy Concerns:** Android collects user data for personalized services and targeted advertising, raising privacy concerns among users who may prefer more stringent data protection measures.

2.2 The Android Application Components

An Android application is composed of several key components that work together to provide the desired functionality. Android application components are essential building blocks that define different aspects of an Android app's behavior and functionality. These components are crucial for creating interactive and dynamic mobile applications. Each component has a specific role, contributing to the overall functionality of an application These components are loosely coupled and are described in the application's manifest file (AndroidManifest.xml), which also includes metadata, hardware configuration, platform requirements, external libraries, and permissions.

Android components can be categorized into **Primary Components (Core Components)** and **Secondary Components (Non-Core)** based on their fundamental roles and supplementary functionalities within an application.

2.2.1 Primary (or) Core Components

Core components are crucial for the basic functionality and structure of an Android application. They handle the main processes, user interactions, and essential background tasks that make the app operational.

1. **Activities :** Activities represent the user interface (UI) of an application. They are responsible for interacting with users and handling user interactions. Each screen in an Android app is typically implemented as an activity. Each activity is typically designed to handle a specific task or function.

   Example : In the PhonePe application, the main screen activity is a key component that displays the user's balance, recent transactions, and various options like "Recharge," "Pay Bills," and "Transfer Money." Selecting an option opens another activity displaying relevant details and actions. Clicking on "Transfer Money" in the main screen opens another activity where users can enter the recipient's details and the amount to transfer. After entering the required information, the user can confirm and complete the transaction in this new activity.
2. **Services:** A service is a component that performs long-running operations in the background without a user interface. They handle tasks that need to continue even when the app is not in the foreground, such as data processing, network operations, or playing music.

   Example : When initiating a payment transfer in PhonePe, the app may start a background service to handle the transaction processing. This ensures continuity even if the app is minimized or the screen is locked. The service can manage network communication, handle retries in case of failure, and ensure that the transaction completes successfully. Additionally, the service can notify the user of the transaction status via notifications.
3. **Broadcast Receivers:** Broadcast Receivers respond to system-wide broadcast announcements. They are used to listen for and respond to broadcast messages from other apps or the system. Broadcast Receivers enable applications to be notified of events like battery status changes, network connectivity changes, or custom broadcasts sent by other applications.

   Example : A broadcast receiver may listen for network connectivity changes. If the internet connection is lost or restored, the app can respond accordingly, such as retrying a pending transaction or notifying the user.
4. **Content Providers:** Content Providers manage a shared set of app data. They allow apps to securely share data with other apps or access data from other apps. Content Providers encapsulate data storage and retrieval mechanisms, providing a standard interface for querying and updating data. They support CRUD (Create, Read, Update, Delete) operations and can handle complex data structures like files, databases, or even cloud storage.

   Example: To access and manage user contacts for payment purposes, PhonePe could use a content provider. This enables the app to read and write contact information stored on the device. For instance, when selecting a contact to send money to, the app would query the contacts content provider to retrieve and display the relevant contact details.

2.2.2 Secondary (or) Non-Core Components

Secondary (or) Non-core components enhance the primary functionalities and improve user experience, but they are not essential for the basic operation of the app. They provide additional features and optimizations.

1. **Fragments :** Fragments represent a behavior or a portion of the user interface in an Activity. Fragments are modular sections of an activity with their own lifecycle and UI. They can be dynamically added, removed, or replaced within an activity. They are reusable UI components

that can be combined to create a multi-pane UI and are commonly used for building flexible layouts.

Example : Imagine a Fragment as a separate section or a small window within the PhonePe app's Payment screen. It's like having a specific area on the screen that shows payment details and options. When we open the PaymentActivity in PhonePe, we might notice a specific section labeled as "Payment Details" or "Payment Options." This section is actually the PaymentFragment. In the PaymentFragment, users can view details such as the amount to be paid, payment methods available (like UPI, credit/debit card), recipient information, and any additional options related to the payment process.

2. **Intents :** Intents are messaging objects used to request actions from other app components. They can start an activity, start a service, or deliver a broadcast. Intents can carry data to be used by the target component. In simple terms, think of Intents as special messages that apps use to ask other parts of the app to do something specific. Apps use Intents to communicate and trigger actions within the app. They help in coordinating different parts of the app to work together seamlessly.

   Example : When a user clicks on the "Transfer Money" button in the main screen, an intent is used to start the money transfer activity, passing necessary information about the transaction. The Intent acts as a carrier, taking all the details about the money transfer from the main screen to the money transfer screen.

3. **Views and ViewGroups :** Views are the basic building blocks for user interface components, and ViewGroups are containers that hold multiple views and define their layout structure. Views include elements like buttons, text fields, and images, while ViewGroups include layouts like LinearLayout and RelativeLayout, which arrange the views.

   Example ; The main screen uses TextView to display the balance, ImageView for icons, and Button for actions like "Recharge" and "Pay Bills". These elements are organized within a LinearLayout to create a structured and user-friendly interface.

4. **Adapters :** Adapters act as a bridge between UI components and data sources to facilitate the display of data in UI components. Adapters are like messengers that fetch data from sources like a database and show it in UI elements such as lists or grids. Adapters bind data to the views within these components to allow dynamic data display.

   Example : In the transaction history section, an adapter helps display transaction details in a scrollable list for easy viewing.

5. **Notifications:** Notifications are used to alert users about events or updates even when the app is not in the foreground. They provide a way to keep users informed about important information.

   Example: Notifications in PhonePe serve as a valuable tool for keeping users informed about important events, such as successful payment transactions, even when the app is not actively being used. By sending notifications with transaction details, like the amount and recipient, PhonePe ensures users stay updated on their financial activities effortlessly.

6. **Loaders:** Loaders load data asynchronously in activities or fragments to ensure that data operations do not block the main UI thread. They manage the background task of loading data and provide the loaded data to the UI components.

Example: When fetching transaction history from a server, a loader performs this operation in the background to ensure a smooth and responsive user interface. Loaders play a crucial role in maintaining a responsive user interface by handling data loading tasks in the background, thus preventing any delays or freezes in the main UI thread.

7. **SharedPreferences**: SharedPreferences allow an app to store key-value pairs locally on the device. They are commonly used to store small amounts of data persistently. They are commonly used for saving user settings or preferences. The data is stored persistently across user sessions.

   Example: User preferences such as the default payment method, notification settings, and the last used bank account are stored using SharedPreferences.

8. **Widgets :** Widgets are self-contained UI components that can be added to the home screen to provide quick access to app functionalities without opening the app. Widgets can display dynamic data and offer interactive elements.

   Example : A widget on the home screen displays the current balance and recent transactions, allowing users to quickly check their financial status without opening the app.

9. **Intent Filters :** Intent Filters define the kinds of intents an activity, service, or broadcast receiver can respond to. They specify the types of actions the component can handle, allowing the system to match intents with the appropriate component.

   Example : Intent Filters in PhonePe act as instructions that guide the app on how to react when specific actions such as making a payment or accessing a particular feature within the app are initiated. These filters serve as guidelines that help the app understand and respond accurately to requests. It ensures that when users interact with the app, it knows precisely how to handle the intended actions.

	Example	Real World Flow - Understanding Android Application Components

1. **User opens PhonePe (Activity) :** The main activity displays the user's balance, recent transactions, and various options using Views and ViewGroups.
2. **User initiates a money transfer (Activity) :** A new activity opens to fill in transfer details.
3. **User selects recipient from contacts (Content Provider) :** The app uses a content provider to access contacts.
4. **User submits the transfer (Service) :** A background service handles the transaction processing.
5. **Transaction status (Broadcast Receiver) :** The broadcast receiver listens for the transaction status update from the server.
6. **Transaction list display (Adapter) :** An adapter binds transaction data to a RecyclerView in the transaction history screen.
7. **Fetching transaction history (Loader) :** A loader fetches transaction history data asynchronously.
8. **User receives transaction status (Notification) :** The app sends a notification about the transaction status.
9. **User settings (SharedPreferences) :** SharedPreferences is used to save user preferences like the default payment method.
10. **User scans QR code (Intent) :** An intent starts the camera activity for QR code scanning.
11. **Payment details section (Fragment) :** The payment screen uses a fragment to display payment details and options.

12. **Handling user actions (Intent Filters) :** Intent filters define the actions the app can respond to, such as initiating a payment.
13. **Quick access widget (Widget) :** A widget on the home screen displays the current balance and recent transactions, allowing users to quickly check their financial status without opening the app.

- **What are the Main Components of an Android Application?**
  The main components of an Android application are Activities, Services, Broadcast Receivers, and Content Providers.
- **What is the purpose of an Activity in an Android Application?**
  An Activity is responsible for providing a user interface (UI) for the application, handling user interactions, and navigating between different screens.
- **How does a Service differ from an Activity in Android?**
  A Service runs in the background without a user interface and performs long-running operations or tasks, such as playing music or downloading files, while an Activity provides a user interface for interacting with the app.
- **What is a Content Provider in Android?**
  A Content Provider manages access to a structured set of data, typically stored in a database, and allows data sharing between different applications.
- **What is a Broadcast Receiver in Android?**
  A Broadcast Receiver listens for broadcast messages from other applications or from the system and responds to them, allowing applications to receive and react to system-wide events.

What is an APK ?

An **APK (Android Package Kit)** is the file format used to distribute and install applications on Android devices. For example, when downloading an app from the Google Play Store, users are actually downloading an APK file that contains all the necessary components of the app, such as code, resources, and assets. This APK file can then be installed on a device to use the app. APK files allow for easy distribution and installation of Android applications outside of official app stores.

2.3 Exploring the Development Environment

To create robust and efficient Android applications, it is essential to have the right tools and frameworks in place. Exploring the Android Development Environment involves understanding the tools and resources available for creating Android applications. The Android development environment requires several components (or) tools (or) resources to develop and test android applications efficiently.

1. **Android SDK :** The Android SDK provides libraries and tools for app building.
2. **Android Studio:** Android Studio is a powerful IDE with code editing and visual design features
3. **Android Emulators:** Android Emulators simulate devices for testing without physical devices
4. **ADB (Android Debug Bridge):** ADB offers various debugging and troubleshooting functionalities.

5. **Gradle:** Gradle is the build automation tool used in Android Studio to manage project dependencies, build configurations, and tasks efficiently.

2.3.1 Android SDK

The Android SDK is a set of tools, libraries, and resources provided by Google to develop Android applications. It serves as the foundation for Android app development and provides essential components and APIs for building feature-rich apps.

What is an Android SDK?

An Android SDK (Software Development Kit) is a set of tools, libraries, and documentation provided by Google to help developers create applications for the Android platform. It includes everything developers need to start building Android apps, such as APIs for interacting with the device hardware, emulator for testing apps, and various other tools for debugging and profiling.

Key Features:

- **Libraries and APIs:** Provides access to a wide range of libraries and APIs for implementing various functionalities in Android apps. These libraries cover various aspects of Android development, from UI components to data storage and networking.
- **Development Tools:** Includes compilers, debuggers, and build tools for writing, compiling, and debugging Android applications.
- **Documentation:** Offers detailed documentation on APIs, best practices, and development guidelines for Android app development.
- **Sample Code and Templates:** Provides sample code snippets and templates to assist developers in implementing common tasks and features.
- **Updates and Support:** Regularly updated by Google to incorporate new features, enhancements, and compatibility with the latest Android versions.

2.3.2 Android Studio

The Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to programmers for software development. **Android Studio** is the official Integrated Development Environment (IDE) for Android app development. It is designed to streamline the entire app development process. It serves as a comprehensive platform for designing, coding, testing, and debugging Android applications. It provides a range of tools and features to facilitate app development.

Key Features:

- **Code Editor:** Android Studio provides a powerful code editor with features like syntax highlighting, code completion, refactoring, and code navigation to enhance productivity and code quality.
- **Layout Editor:** The Layout Editor enables developers to create visually appealing user interfaces with drag-and-drop functionality, real-time previews, and support for different screen sizes and orientations.

- **SDK Manager :** The Android SDK Manager is a tool within Android Studio that allows developers to download, install, and manage different versions of the Android SDK, as well as other essential tools and components.
- **Device Manager :** The Device Manager is a tool within Android Studio that allows developers to create and manage virtual devices that simulate physical Android devices. These virtual devices are used to test and debug applications on various configurations and Android versions without needing physical devices.
- **APK Analyzer:** Helps analyze APK size, contents, and dependencies to optimize app performance and reduce file size.
- **Built-in Emulator:** The built-in Android Emulator allows developers to test their apps on virtual devices with various configurations, screen sizes, and Android versions for comprehensive testing.
- **Device Testing:** Developers can test their apps on physical devices connected via USB for real-world testing scenarios, in addition to the Android Emulator, to ensure app compatibility and performance.
- **Version Control Integration:** Android Studio supports version control systems like Git, enabling developers to manage source code, track changes, and collaborate with team members efficiently within the IDE.
- **Extensive Plugin Ecosystem:** Android Studio offers a wide range of plugins and extensions from the Plugin Marketplace to extend functionality, integrate with external tools, and customize the development environment according to specific requirements.
- **Performance Profiling Tools:** Android Studio offers performance profiling tools to analyze app performance, identify bottlenecks, and optimize CPU, memory, and network usage for better user experience.
- **Build Tools:** Android Studio integrates the Gradle build system for automating build tasks, managing dependencies, and configuring build variants to streamline the app development proces
- **Integrated Debugger:** An integrated debugger allows developers to identify and fix issues in their code by setting breakpoints, inspecting variables, and stepping through code execution.

2.3.3 Android Emulators

Android Emulators are virtual devices that simulate the behavior of real Android devices for testing and debugging apps. They allow developers to test and debug applications on various device configurations without needing physical devices. They provide a virtual testing environment to ensure app compatibility, performance, and functionality across different device configurations.

Key Features:

- **Device Configurations:** Emulate different screen sizes, resolutions, and Android versions.
- **Device Testing:** Allows developers to test apps on virtual devices with various screen sizes, resolutions, and Android versions.
- **Debugging:** Facilitates app debugging, scenario simulation, and behavior analysis without physical devices.

- **Performance Testing:** Helps evaluate app performance, identify bottlenecks, and optimize resource usage during development.
- **Multi-Device Testing:** Supports testing on multiple virtual devices simultaneously to assess app behavior across different configurations.
- **Advanced Features:** Simulate phone calls, SMS, location, and other hardware features.

2.3.4 ADB (Android Debug Bridge)

The Android Debug Bridge (ADB) is a versatile command-line tool that serves as a bridge between a development machine and an Android device or emulator. It offers a range of debugging and troubleshooting functionalities to help in the development and testing of Android applications.

Key Functions:

- **Installing and Debugging Apps:** ADB allows developers to install and debug applications directly on connected Android devices for efficient testing and troubleshooting.
- **Accessing Device's Shell:** It provides access to the device's shell, allowing developers to perform advanced troubleshooting tasks and execute commands directly on the device.
- **Transferring Files:** ADB facilitates the seamless transfer of files between a computer and an Android device to simplify the tasks such as moving resources, logs, or APK files for testing purposes.
- **Managing Device State and Permissions:** Developers can use ADB to manage the state of the device, change permissions, and simulate various scenarios for testing different aspects of their applications.

2.3.5 Gradle

The Gradle Build System is a powerful build automation tool integrated into Android Studio to streamline the management of project dependencies, build configurations, and tasks efficiently.

Key Features:

- **Dependency Management:** Gradle simplifies the process of including libraries and resources in Android projects to ensure that dependencies are managed effectively.
- **Build Configurations:** Developers can define different build types and flavors in Gradle, allowing for customized configurations tailored to specific requirements such as debug, release, or flavor-specific builds.
- **Task Automation:** Gradle automates various tasks involved in the development lifecycle, such as compiling code, running tests, and packaging applications for distribution, enhancing productivity and consistency.
- **Integration with Android Studio:** Gradle seamlessly integrates with Android Studio, to provide a cohesive development environment where developers can leverage Gradle's capabilities within the IDE for a smooth and efficient development workflow.

2.4 Obtaining the Required Tools

Setting up an Android development environment is essential for creating, testing, and debugging Android applications. The primary components required are the **Java Development Kit (JDK)**

and **Android Studio**. These tools provide all the necessary resources for Android development, as Android Studio includes the Android SDK, Android Emulators, ADB, and Gradle.

1. **Java Development Kit (JDK) :** The JDK is crucial for developing Java applications, including Android apps. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), and other tools necessary for Java development.

   Steps to Obtain:

   - **Download the JDK:** Visit the Oracle JDK download page (https://www.oracle.com/java/technologies/downloads/) and download the JDK appropriate for the operating system being used (Windows, macOS, or Linux).
   - **Install the JDK:** Follow the installation instructions specific to the operating system (Windows, macOS, or Linux).
   - **Configure Environment Variables:** Set the JAVA_HOME environment variable to point to the JDK installation directory and add the JDK bin directory to the PATH.

2. **Android Studio :** Android Studio is the official Integrated Development Environment (IDE) for Android development. It offers a comprehensive suite of tools for building, testing, and debugging Android applications. Android Studio includes the **Android SDK**, **Android Emulators**, **ADB**, and **Gradle**, making it a one-stop solution for Android development.

   Steps to Obtain:

   - **Download Android Studio:** Available from the[Android developer website (https://developer.android.com/studio)
   - **Install Android Studio:** Run the installer and follow the setup wizard to install and configure the development environment.
   - **Configure the SDK:** During the installation process, Android Studio will prompt for the installation of the Android SDK and other necessary components. SDK components can be managed through the SDK Manager within Android Studio.
   - **Create a New Project:** Use the project wizard to start a new Android project, selecting the desired SDK version and project template.

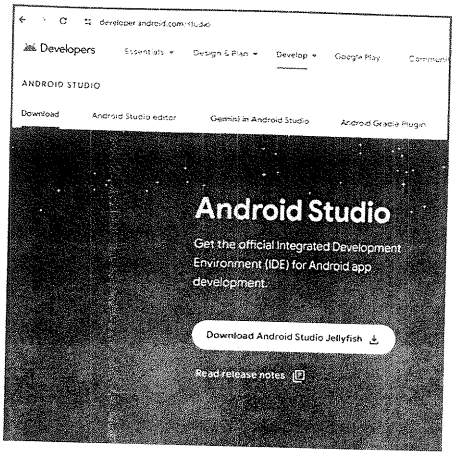
By installing these two components, all the necessary tools for Android development will be available, as Android Studio includes the Android SDK, Android Emulators, ADB, and Gradle. This streamlined setup ensures that everything needed to start building, testing, and debugging high-quality Android applications is in place.

2.5 Installing Android Studio

Step 1: Download Android Studio:

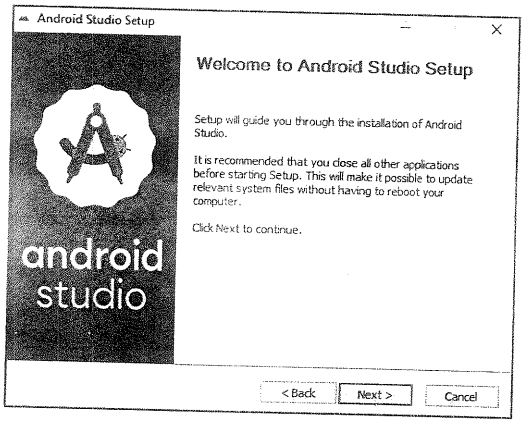
Visit the official Android Studio download page (https://developer.android.com/studio) and download the executable or zip file. Click the "Download Android Studio" button to begin.

Note that at the time of this book's publishing, the latest version of Android Studio is Jelly Fish 2023.3.1 Patch 1. However, new versions will continue to evolve. Download the latest version available at that time.



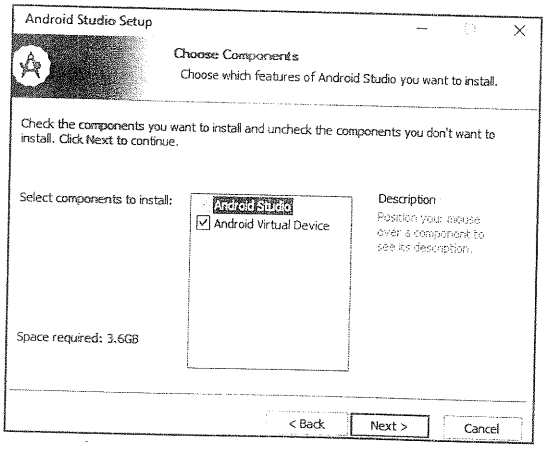
Step 2: Run the Installer:

Once the download is complete, open the downloaded file. This will start the Android Studio Setup Wizard.



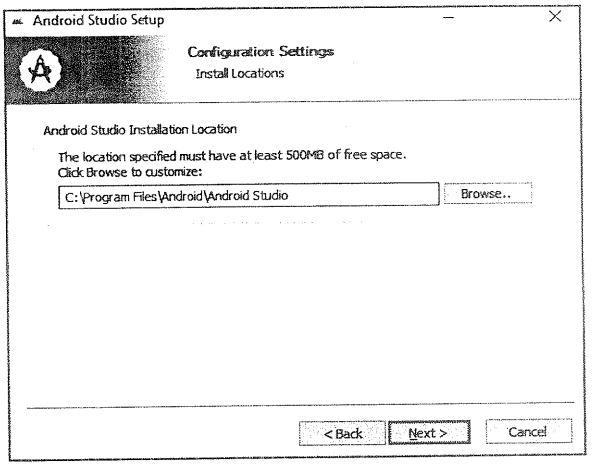
Step 3: Choose Components:

Select the components to install, such as Android Studio and Android Virtual Device (AVD).



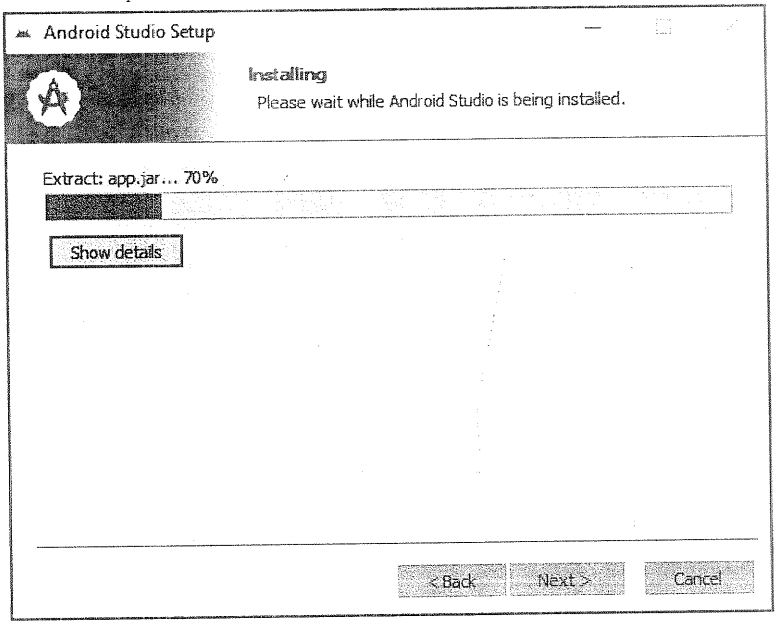
Step 4: Choose Install Location:

Choose the location where you want to install Android Studio. It is recommended to use the default location.



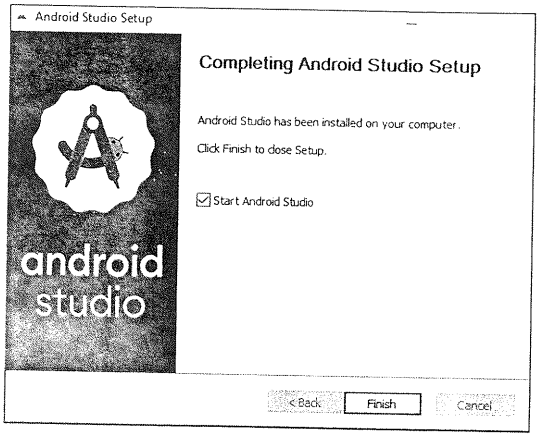
Step 5: Start the Installation:

Click on the "Install" button to start the installation process. Android Studio will begin installing the selected components.



Step 6: Complete the Installation:

Once the installation is complete, click on the "Finish" button. Android Studio is installed successfully.

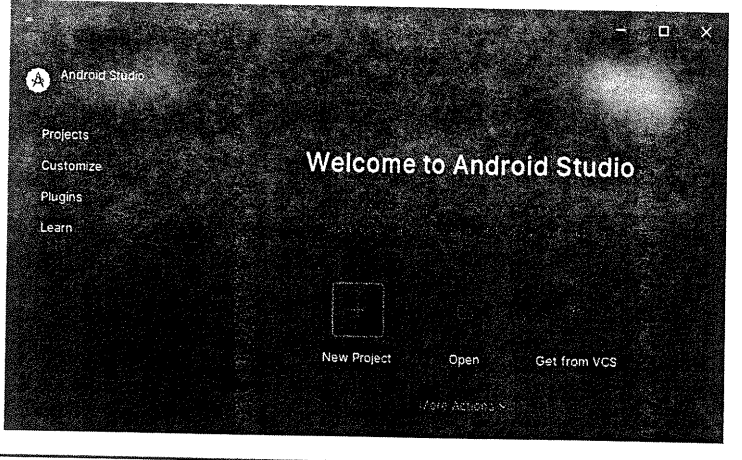


Step 7: Launch Android Studio:

After installation, launch Android Studio. It will prompt you to import settings if you had a previous installation. Choose the "Don't import settings" option and click "OK".

Step 8: Set Up Android Studio:

Android Studio will launch and prompt for the development environment setup. Follow the on-screen instructions to complete the setup process. Choose "Standard" as the Install Type and accept the License Agreement to download all necessary components such as the SDK, Emulators, Build Tools, and more. Once the components have finished downloading, the welcome screen will appear as shown below.

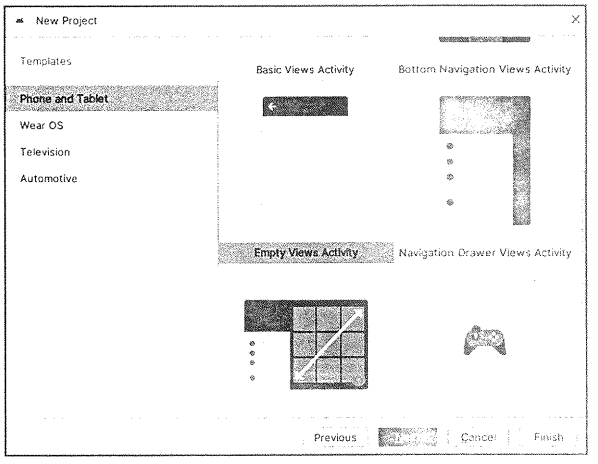


2.6 Your First Android Application - Hello World

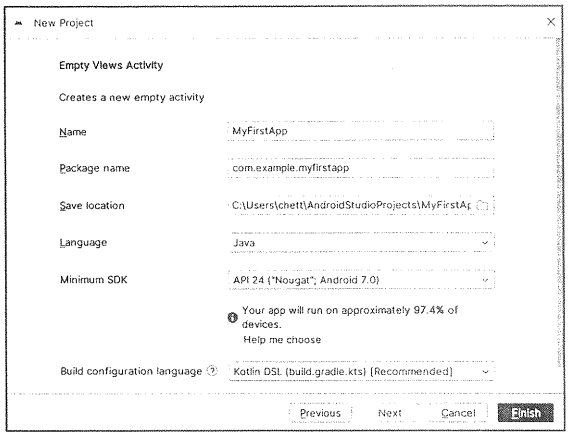
Let us creating a first Android application with a simple text message "Hello,World!" in the center of the mobile screen.

Step 1: Create a New Project

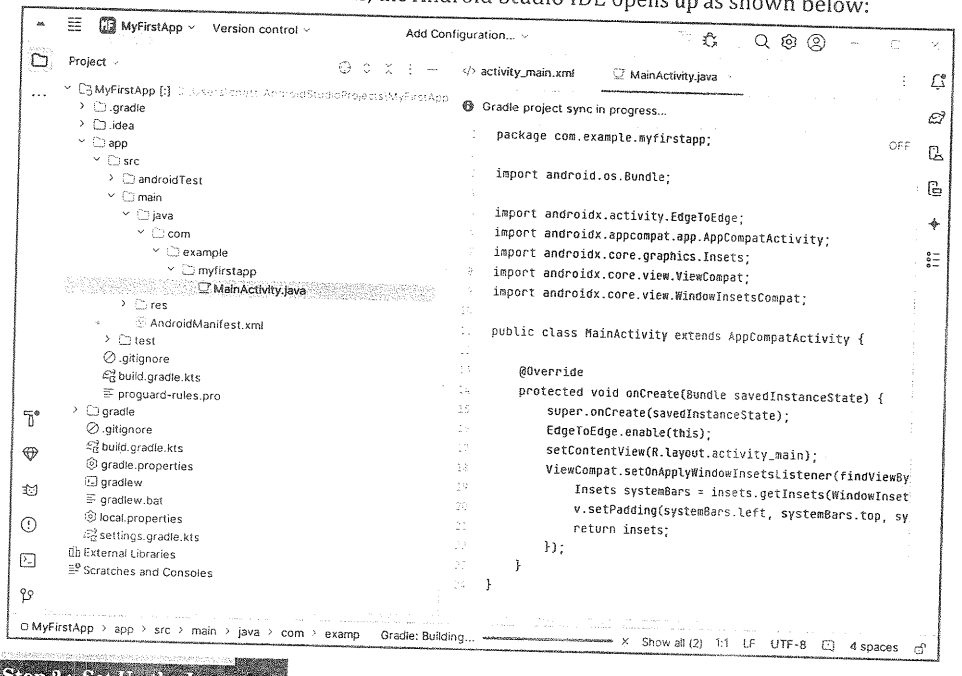
1. **Open Android Studio:** Launch Android Studio.
2. **Start a New Project:** Click on Start a new Android Studio project.
3. **Select Project Template:** In the "Select a Project Template" window, choose Empty Views Activity. Click Next.



4. **Configure the Project:**
   - **Name:** Enter "MyFirstApp".
   - **Package name:** Enter a unique package name, such as com.example.myfirstapp.
   - **Save location:** Choose a directory to save the project.
   - **Language:** Select Java.
   - **Minimum API level:** Select the minimum version of Android to support (default is usually fine).



- Click Finish and After this, the Android Studio IDE opens up as shown below:



Step 2 : Set Up the Layout

1. **Open the Layout File:** In the Project pane on the left side, navigate to app -> res -> layout. Double-click **activity_main.xml** to open it.
2. **Design the Layout:**
   - **Switch to Design View:** By default, the editor opens in Design mode, so no need to switch unless working directly with XML code is preferred.
   - **Add a View:**

     The default display in the screen layout shows the "Hello, World!" text view. To replace this with a different view, locate the desired view in the Palette on the left side of the Design view. Then, drag and drop the selected view onto the phone screen layout in the design view.
   - **Set Text Properties:**

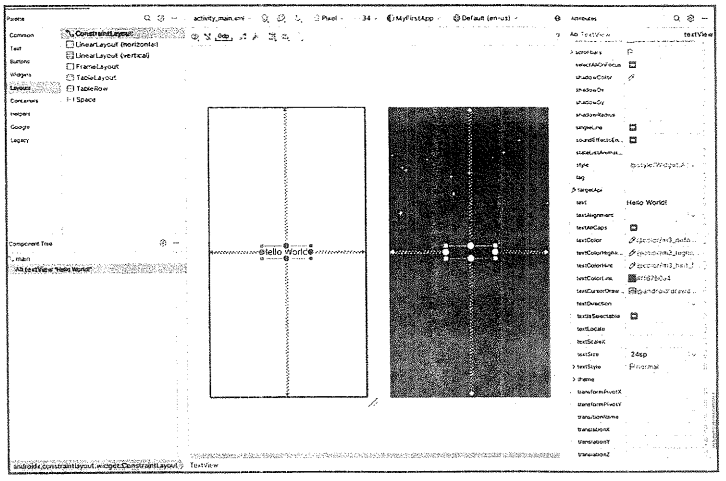
     Click on the TextView to select it.

     In the Attributes panel (usually on the right side), set the following properties:

     id: @+id/textView

     text: Hello, World!

     textSize: 24sp



- Alternatively, switch to the Code tab and replace the existing XML code with the following simple and minimum code.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello,World!"
        android:textSize="24sp"
        android:layout_centerInParent="true"/>
</RelativeLayout>
```

Step 3: Edit the Main Activity

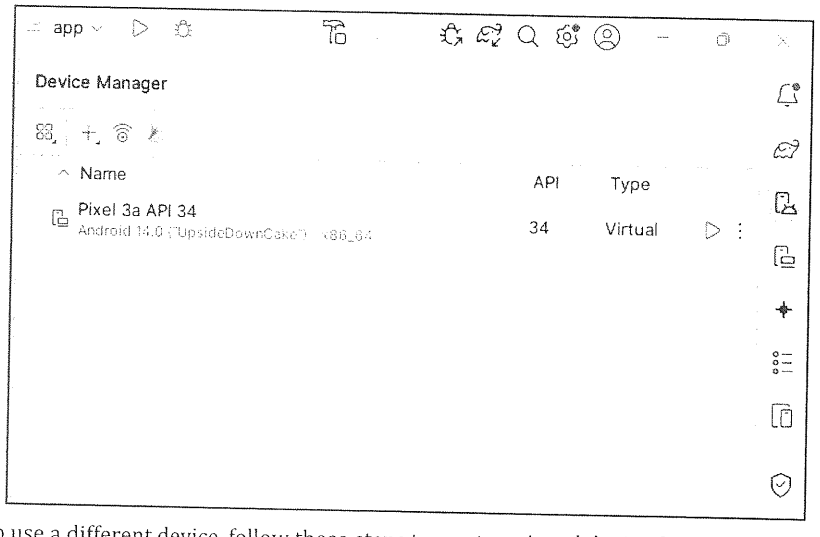
1. **Open MainActivity.java:**
   - In the Project pane, navigate to app -> java -> com.example.myfirstapp.
   - Double-click MainActivity.java to open it.
2. **Modify MainActivity.java:**
   - Ensure that the activity displays the layout defined. This is typically already set up, but confirm the following code is present:

Code MainActivity.java

```
package com.example.myfirstapp;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Step 4: Run the Application

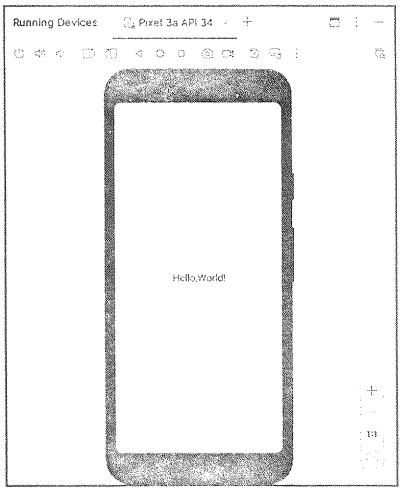
1. **Connect a Device:** We can connect an android device Via USB by enabling developer options and USB debugging on the device (or) we can set up an emulator to test the application. By default, Android Studio provides the Pixel 3a API 34 device for testing.



To use a different device, follow these steps to create a virtual device for testing.

- Go to Tools -> Device Manager.
- Click on Create Virtual Device.
- Choose a device definition and click Next.
- Select a system image and click Next.
- Configure AVD settings and click Finish.

2. **Run the Application:**
   - Click on the Run button (green triangle) in the toolbar, or press Shift + F10.
   - Select the connected device or emulator from the list.
   - Click OK and wait for the build process to complete and the app to install on the device/emulator.
   - The Output is shown below:



2.6.1 Understanding the Flow of Execution

When creating an Android project, three main components are typically encountered: the Java file representing an Activity (e.g., MainActivity.java), the XML layout file associated with the Activity (e.g., activity_main.xml), and the AndroidManifest.xml file. Understanding the flow of execution and the significance of each component is crucial for developing Android applications effectively:

Each activity in an Android application is associated with a corresponding XML layout file that defines the user interface for that particular screen. When multiple activities are created, each activity typically corresponds to a unique XML layout file, representing a distinct UI screen within the application.

Flow of Execution

The flow of execution in an Android application involves the system launching the main activity specified in the manifest file, instantiating the activity, calling its lifecycle methods in a predefined order, and setting the user interface by referencing an XML layout file within the activity's onCreate method.

1. **Launching the Application:** The Android system starts an application by looking for the AndroidManifest.xml file to determine the main activity to launch. The main activity is specified in the manifest file using intent filters, indicating that it is the activity to be launched when the application starts.
2. **Executing the Activity:** The main activity specified in the AndroidManifest.xml file is instantiated. The lifecycle methods of the activity (such as onCreate, onStart, etc.) are called in sequence, beginning with onCreate. Following onCreate, other lifecycle methods like onStart, onResume, onPause, onStop, onDestroy, and onRestart are called based on the state transitions of the activity. These methods allow developers to manage the behavior of the activity as it

moves through different states, such as becoming visible to the user, interacting with the user, being paused or stopped, and eventually being destroyed.

3. **Setting the User Interface:** Within the onCreate method of the activity, the activity xml layout file is referenced to define the user interface. The activity xml layout file specifies the arrangement of user interface elements like buttons, text views, images, and other widgets that compose the visual presentation of the activity. By setting it as the content view of the activity in the onCreate method, the user interface defined in the XML layout file becomes visible to the user when the activity is displayed on the screen.

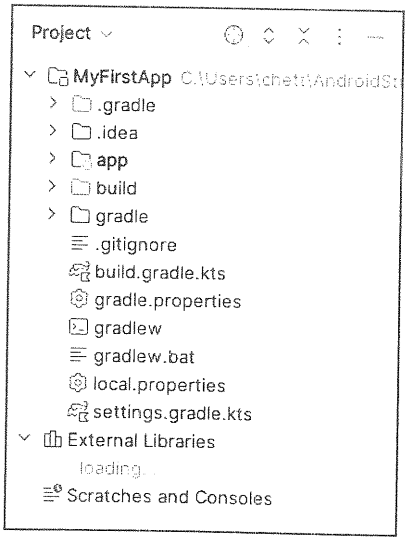
2.7 Exploring the IDE (Android Studio)

Android Studio is the official Integrated Development Environment (IDE) for Android app development, provided by Google. It offers a comprehensive set of tools and features designed to streamline the process of creating, testing, and deploying Android applications.

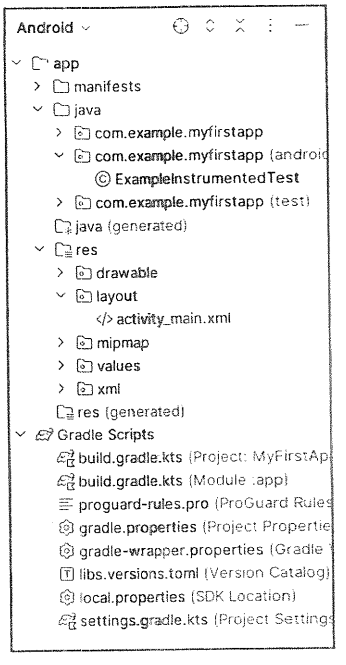
Android Studio is specifically tailored for building Android apps. It provides a unified environment where developers can write code, design user interfaces, debug applications, and manage project resources efficiently. Android Studio is built on top of IntelliJ IDEA, a popular Java IDE. This foundation brings powerful code editing capabilities, intelligent code completion, and a range of productivity features to Android developers.

2.7.1 Exploring the Integrated Development Environment (IDE) in Android Studio

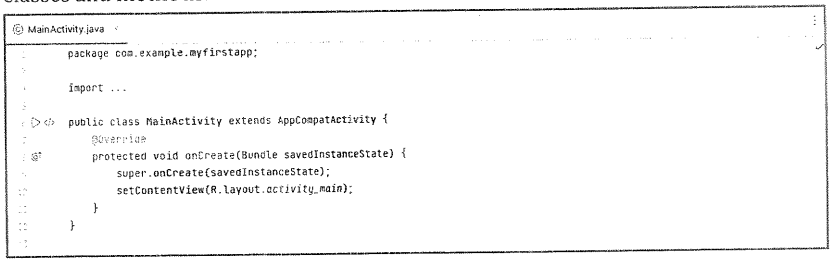
1. Project Structure:
   - **Project View:** Located on the left side of the IDE, the Project View provides a hierarchical representation of the project's files and directories, making it easy to navigate and manage project components.



   - **Android View:** This view organizes project files specifically for Android development, grouping Android-specific resources, manifests, Java classes, and other relevant files for efficient development.

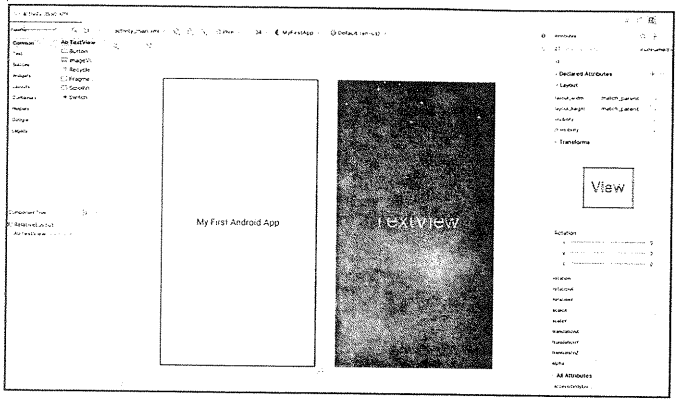


2. **Code Editor:**
   - **Code Editing:** The central area of the IDE is the code editor, where developers write and modify Java and XML code. Android Studio offers features like syntax highlighting, code completion, and error checking to enhance the coding experience.
   - **Code Navigation:** Utilize shortcuts and features such as "Go to Declaration" to swiftly navigate through the codebase, facilitating better understanding and management of classes and methods.

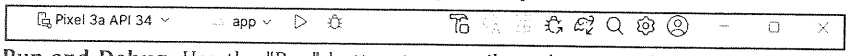


3. **Design Editor:**
   - **Layout Editor:** Android Studio's visual layout editor allows developers to design the app's user interface by dragging and dropping UI elements onto the design surface. It provides a real-time preview of the layout across different screen sizes and orientations.

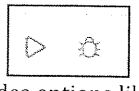
- **Component Tree:** The Component Tree displays the hierarchy of UI elements in the layout, enabling developers to organize and manage the structure of the user interface efficiently.



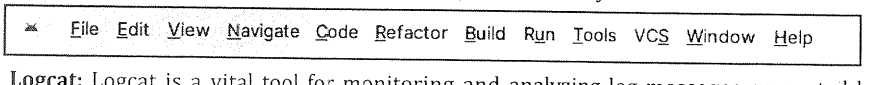
4. **Toolbar:** The Toolbar houses various buttons for common actions such as building and running the app, syncing project files, debugging, and accessing version control tools. It offers quick access to essential functions during app development.



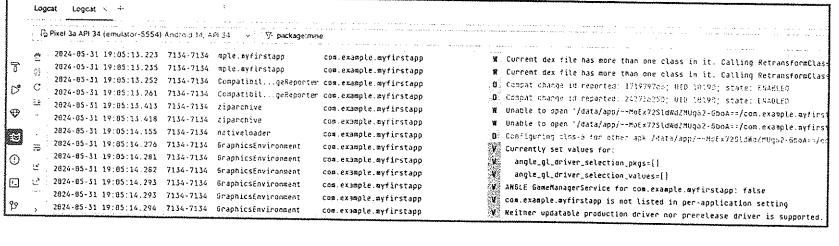
5. **Run and Debug:** Use the "Run" button to compile and execute the app on an emulator or physical device. The "Debug" button allows developers to set breakpoints, inspect variables, and step through the code for debugging purposes, aiding in identifying and resolving issues.



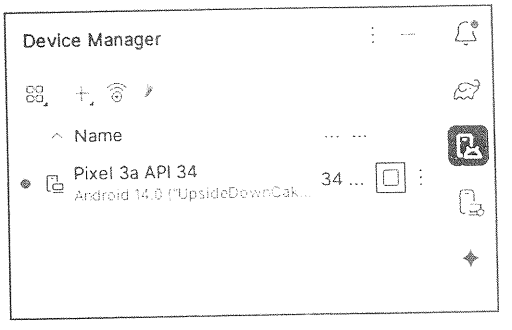
6. **Menu:** Android Studio's menu includes options like File, Edit, View, Navigate, Code, Refactor, Build, Run, Tools, VCS (Version Control System), Window, and Help. These menus provide access to various features and functionalities within the IDE, allowing developers to navigate, edit, build, run, and manage their Android projects effectively.



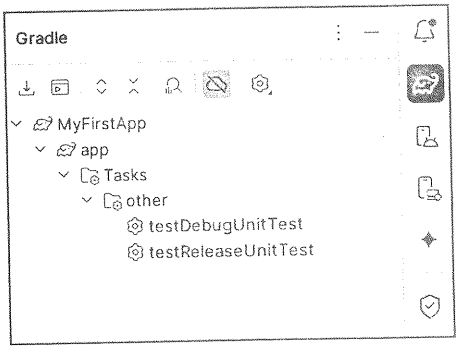
7. **Logcat:** Logcat is a vital tool for monitoring and analyzing log messages generated by the app during runtime. It provides insights into the app's behavior, error messages, and system warnings, assisting developers in diagnosing and troubleshooting issues effectively.



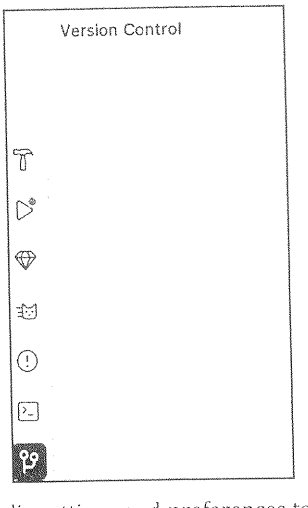
8. **Device Manager:** The Device Manager enables developers to create and manage Android Virtual Devices for testing the app on various device configurations and Android versions. It is instrumental in ensuring the app's compatibility and performance across different devices.



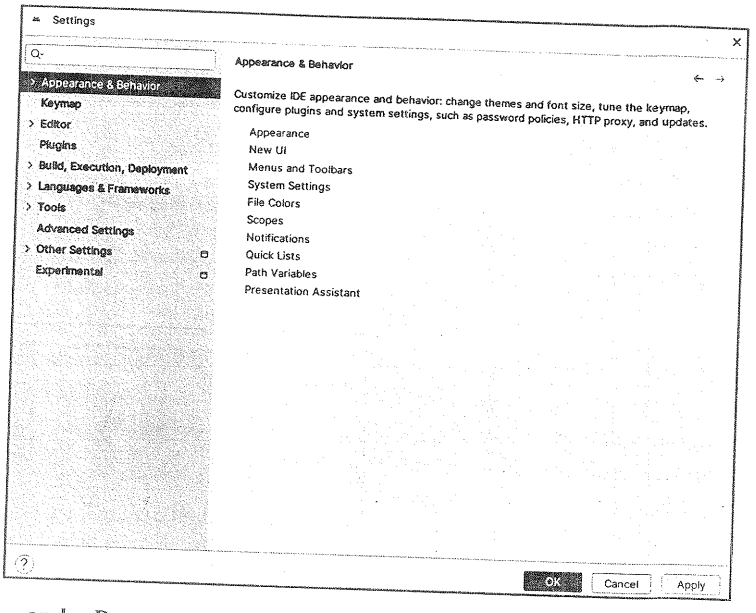
9. **Gradle Build System:** Android Studio utilizes the Gradle build system to automate the build process, manage dependencies, and configure build settings. Developers can customize build types, flavors, and dependencies in the build.gradle files to tailor the app's build process to specific requirements.



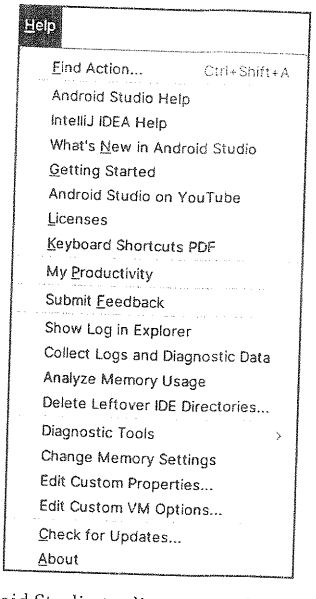
10. **Version Control:** Android Studio integrates version control systems like Git, allowing developers to track changes, manage project source code, and collaborate with team members efficiently. Version control features enable seamless code management and collaboration within the IDE.



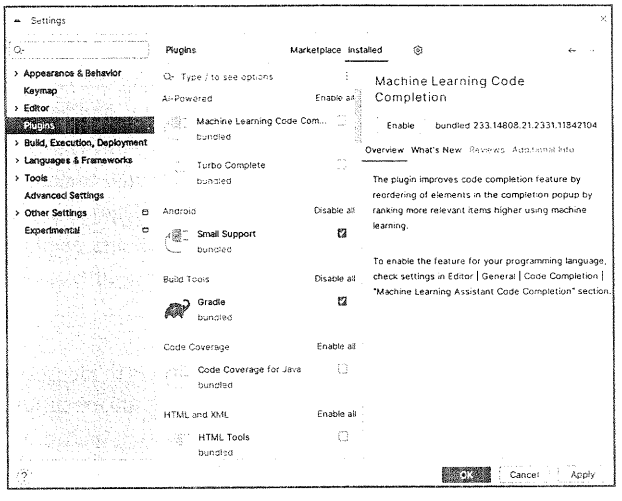
11. **Settings and Preferences:** Customize Android Studio settings and preferences to align with individual development workflows by accessing the Preferences or Settings menu. Developers can adjust editor settings, keymaps, appearance, and other configurations to personalize the IDE environment.



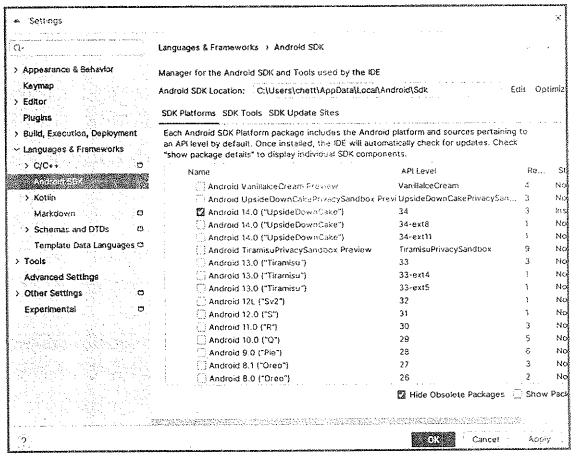
12. **Help and Documentation:** Access Android Studio documentation and support resources directly from the IDE. The Help menu provides access to documentation, tutorials, and community support, offering valuable assistance and guidance for developers during app development.



13. **Plugins:** Explore the Plugin Marketplace within Android Studio to discover and install plugins that extend the IDE's functionality. Plugins can enhance productivity, introduce new features, and integrate with external tools to streamline the development process and cater to specific requirements.



14. **SDK Manager :** The SDK Manager in Android Studio is a crucial tool that allows developers to manage the Android SDK components, platforms, tools, and system images required for app development.



2.7.2 Android Studio Developer Workflow Guide

1. **Set Up the Workspace:**
   - **Install Android Studio:** Download and install Android Studio, the official IDE for Android app development.
   - **Create a New Project:** Start a new project in Android Studio, select the desired project template, and configure project settings.
2. **Write the App:**
   - **Code Completion:** Utilize code completion features in Android Studio to speed up coding and reduce errors by suggesting code snippets and auto-completing syntax.

- **Refactoring:** Refactor code efficiently using built-in tools to improve code structure, readability, and maintainability.
- **Design Tools:** Design the user interface of the app using the Layout Editor, XML layout files, and visual design tools to create a visually appealing and user-friendly UI.

3. **Build and Run:**
   - **Compile the Project:** Build the project in Android Studio to compile the source code, resources, and dependencies into an APK (Android Package).
   - **Test on Emulators or Devices:** Run the app on emulators or physical devices to test functionality, UI responsiveness, and compatibility across different Android versions and screen sizes.
4. **Debug, Profile, and Test:**
   - **Debugging Tools:** Use Android Studio's debugging tools to identify and fix issues in the code, set breakpoints, inspect variables, and trace program execution.
   - **Performance Profilers:** Analyze app performance using built-in profilers to optimize CPU, memory, and network usage for better efficiency and user experience.
   - **Run Tests:** Write and run unit tests, integration tests, and UI tests to validate app behavior, functionality, and user interactions to ensure code quality and reliability.
5. **Publish:**
   - **Prepare for Release:** Prepare the app for release by configuring build variants, signing the APK, and optimizing resources for distribution.
   - **Sign the App:** Sign the APK with a digital certificate to ensure app authenticity and integrity.
   - **Publish to Google Play Store:** Create a developer account on Google Play Console, upload the signed APK, add app details, screenshots, and descriptions, set pricing and distribution options, and publish the app to the Google Play Store for users to download and install.

2.7.3 Understanding the Project Structure in Android Development

Let us understand the key components of an Android project structure and how they contribute to building a successful app.

1. **App Module:** The app module is the core of the Android project. It contains all the code, resources, and configurations specific to the app. Within the app module, we will find the following directories:
   - **Java/Kotlin:** This directory contains Java or Kotlin code. It's where we write the logic for app's functionality, including activities, fragments, services, and more.
   - **res:** The "res" directory is where we store all app's resources, such as layouts, drawable images, strings, and styles. It helps in separating code from app's visual elements.
   - **Manifest:** The "AndroidManifest.xml" file is a crucial part of the project. It contains essential information about the app, such as its name, package name, activities, permissions, and more.

- **Assets:** If app requires raw asset files, such as JSON, fonts, or custom configurations, we can store them in the "assets" directory.

2. **Gradle Scripts:** Gradle is the build system used in Android development. In a project, we will find two main Gradle files:
   - **build.gradle (Project):** This file configures the build settings for an entire project, such as the Gradle version and repositories.
   - **build.gradle (Module)**: This file configures the build settings specific to an app module, such as dependencies and packaging options.
3. **Dependencies**: Dependencies are external libraries or modules that an app relies on. We can manage dependencies in the "build.gradle" file for an app module. Gradle handles the downloading and integration of these dependencies into a project.
4. **Build Configurations:** Android projects can have different build configurations, such as debug and release. These configurations define how an app is built, packaged, and signed. We can customize these configurations in Gradle files.
5. **Resource Files:** Android uses a variety of resource files to define the look and feel of an app. Some common types of resource files include:
   - **Layouts:** XML files that define the structure and appearance of an app's UI.
   - **Drawable:** Image files or XML drawables used for app icons, backgrounds, and other visual elements.
   - **Values:** XML files that define constants, styles, and strings used in an app.
6. **Localization:** Android supports app localization to provide different language translations for an app's text resources. We can create separate resource directories for each language (example, "values-es" for Spanish).

2.7.4 Android Manifest File

The **AndroidManifest.xml** file is a crucial component of an Android application that provides essential information about the app to the Android system. It contains metadata about the application, its components (such as activities, services, broadcast receivers), permissions required, hardware and software features used, and more. It is located in the project's root directory as **AndroidManifest.xml**, this file defines how different parts of the app interact with each other and with the Android system.

1. Purpose and Importance of the Manifest File

- The manifest file defines the structure and metadata of the application.
- It includes nodes for each activity, service, content provider, and broadcast receiver that make up the application.
- It specifies the application's metadata, such as its icon, version number, and theme.
- The manifest file also declares permissions required by the application and sets hardware, screen, or platform requirements.

2. Structure (or) Components of AndroidManifest.xml

The AndroidManifest.xml file is structured in a hierarchical format with nested elements. Let us understand the components of AndroidManifest.xml.

1. **XML Declaration**

   The file starts with an XML declaration specifying the version of XML being used.

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

2. **<manifest> Element:**

   The '<manifest>' element is the root element of the AndroidManifest.xml file and contains information about the application package and versioning.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.myfirstapp"
          android:versionCode="1"
          android:versionName="1.0">
```

Attributes:

- **xmlns:android:** The XML namespace declaration. This must be included in the root element.
- **package:** The unique identifier for the application. This should match the package name used in the Java or Kotlin code.
- **android:versionCode:** : An integer value that represents the version code of the application.
- **android:versionName:** A string value that represents the release version of the application, shown to users.

3. **<application> Element:**

   The '<application>' element contains information about the application itself, such as its icon, label, theme, and components.

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
```

Attributes:

- **android:allowBackup:** Specifies whether the application's data can be backed up and restored.
- **android:icon:** Specifies the icon for the application.
- **android:label:** Specifies the default text label for the application.
- **android:roundIcon:** Specifies the round icon for the application.

- **android:supportsRtl:** Indicates whether the application supports right-to-left (RTL) layouts.
- **android:theme:** Specifies the default theme for the application.

4. **<activity> Element:**

Inside the '<application>' element, '<activity>' elements define the activities in the application. Each '<activity>' element represents a screen with a user interface.

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity android:name=".SecondActivity" />
```

Attributes:

- **android:name :** Specifies the name of the activity class.
- **<intent-filter> :** Defines the intent filters for the activity, specifying how it can be launched. The example shows an activity that responds to the MAIN action and LAUNCHER category, making it the entry point of the application.

5. **Services: <service>**

Declares a service, which is a component that performs long-running operations in the background.

```
<service android:name=".MyService" />
```

Attributes:

- **android:name:** Specifies the name of the service class.

6. **Broadcast Receivers: <receiver>**

Declares a broadcast receiver, which allows the application to listen for system-wide broadcast announcements.

```
<receiver android:name=".MyReceiver" />
```

Attributes:

- **android:name:** Specifies the name of the receiver class.

7. **Content Providers: <provider>**

Declares a content provider, which manages access to a structured set of data.

```
<provider android:name=".MyProvider"
          android:authorities="com.example.myapp.provider"
          android:exported="false" />
```

Attributes:

- **android:name:** Specifies the name of the provider class.
- **android:authorities:** Specifies the URI authority that identifies the provider.
- **android:exported:** Specifies whether the provider is accessible by other applications.

8. **Permissions:**

   The '<uses-permission>' element is used to declare permissions required by the application to access certain features or resources.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

   Attributes:

   - **android:name :** Specifies the name of the required permission (e.g., android.permission.INTERNET).

9. **Features: <uses-feature>**

   Declares the hardware and software features used or required by the application.

```
<uses-feature android:name="android.hardware.camera" android:required="true" />
```

   Attributes:

   - **android:name:** Specifies the name of the feature (e.g., android.hardware.camera).
   - **android:required:** Indicates whether the feature is required to run the application.

Example Manifest File

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp"
    android:versionCode="1"
    android:versionName="1.0">

    <!-- Application-level metadata and components -->
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

       <!-- Activities -->
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

       <!-- Services -->
        <service android:name=".MyService" />

       <!-- Broadcast Receivers -->
        <receiver android:name=".MyReceiver" />
```

```
        <!-- Content Providers -->
         <provider android:name=".MyProvider"
                   android:authorities="com.example.myapp.provider"
                   android:exported="false" />
    </application>

    <!-- Permissions -->
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <!-- Features -->
    <uses-feature android:name="android.hardware.camera" android:required="true" />

</manifest>
```

3. Importance (or) Significance of AndroidManifest.xml File

The AndroidManifest.xml file is of utmost importance in Android development due to the following reasons:

1. **Application Configuration :** It serves as a central configuration file for the Android application to provide essential details such as the package name, versioning, permissions, components, and more. Proper configuration in the manifest file ensures the app functions correctly within the Android ecosystem.
2. **Component Declaration :** The manifest file declares all components of the application, including activities, services, broadcast receivers, and content providers. This declaration is crucial for the Android system to recognize and interact with these components.
3. **Permissions Management :** Permissions declared in the manifest file define the security boundaries of the application. By specifying the permissions required for accessing sensitive resources or features, developers ensure that the app operates securely and adheres to Android's permission model.
4. **Intent Filters ;** Intent filters specified in the manifest file define how components can respond to incoming intents. This mechanism enables activities to be launched, services to be started, and broadcast receivers to respond to system events, enhancing the app's functionality and interoperability.
5. **Lifecycle Management:** The manifest file plays a role in managing the lifecycle of the application and its components. It defines how activities are launched, how services are started, and how the app interacts with the system, ensuring proper handling of events and transitions.
6. **Compatibility and Distribution :** Properly configuring the manifest file with details like versioning, hardware and software features, and supported screen sizes ensures compatibility with various devices and Android versions. Additionally, the manifest file provides information necessary for distributing the app on the Google Play Store or other platforms.
7. **System Integration :** The Android system relies on the manifest file to understand the structure and behavior of the application. By accurately defining components, permissions, and other details, developers enable seamless integration of the app with the system and other applications.

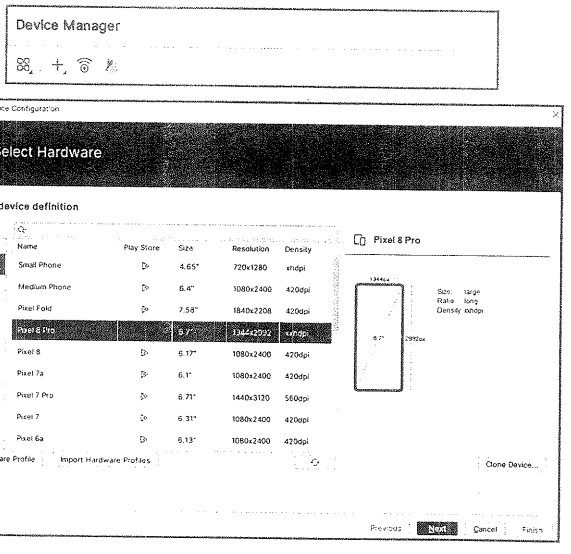
8. **Debugging and Troubleshooting :** During development and troubleshooting, the manifest file serves as a reference point for understanding the app's configuration and behavior. Any inconsistencies or errors in the manifest can impact the app's functionality and may lead to issues that need to be addressed.

2.7.5 Creating a Virtual Device

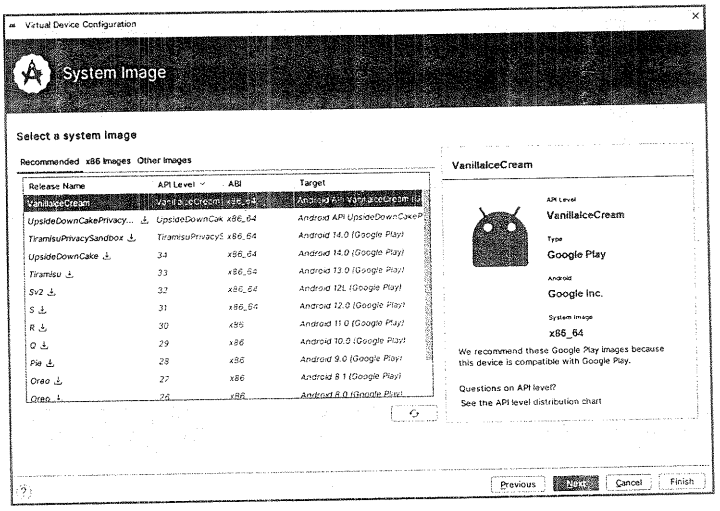
An Android Virtual Device (AVD) is a configuration that defines the characteristics of an Android phone, tablet, Wear OS, Android TV, or Automotive OS device to simulate in the Android Emulator. The Device Manager is a tool to launch from Android Studio that helps to create and manage AVDs.

Steps to Create an AVD (Android Virtual Device)

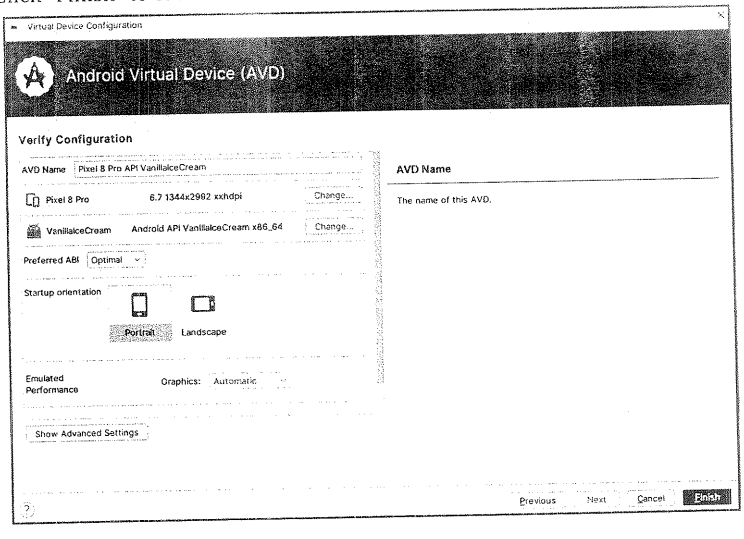
1. **Open Android Studio:**
   - Launch Android Studio on the computer.
2. **Access AVD Manager:**
   - Click on the "Tools" menu in the top toolbar.
   - Select "Device Manager" from the dropdown menu.
3. **Create a New Virtual Device:**
   - In the Device Manager window, click on the "Create Virtual Device" button.
   - Choose a hardware profile for the virtual device (e.g., Pixel, Nexus).
   - Click "Next" to proceed.



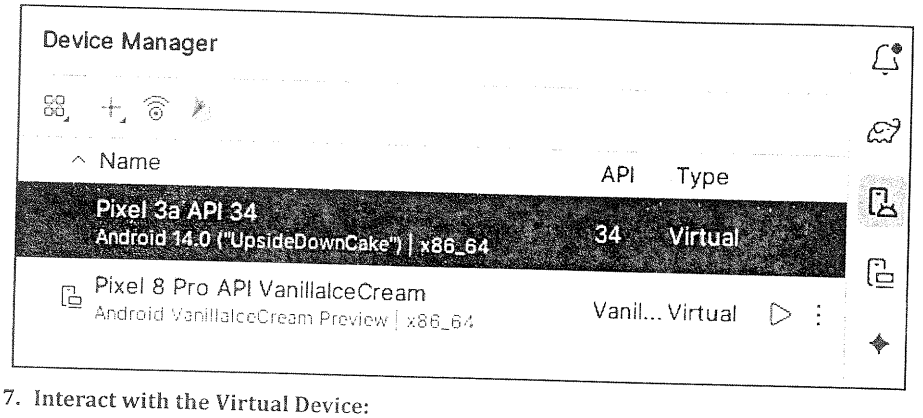
4. **Select a System Image:**
   - Choose a system image for the virtual device based on the desired Android version and API level.
   - Click "Next" to continue.



5. **Configure AVD Settings:**
   - Enter a name for the AVD to identify it in Android Studio.
   - Customize the AVD settings such as RAM size, internal storage, screen size, orientation, and other hardware properties.
   - Click "Finish" to create the AVD.



6. **Launch the Virtual Device:**
   - In the Device Manager Window, locate the newly created AVD in the list.
   - Click on the green play button to launch the virtual device.

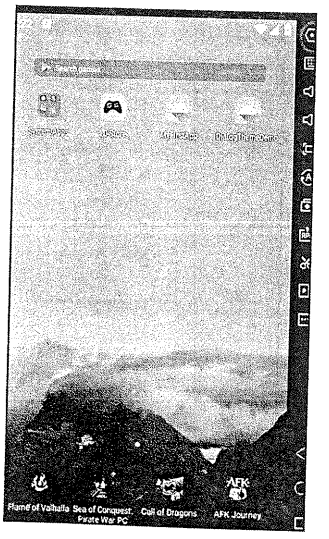


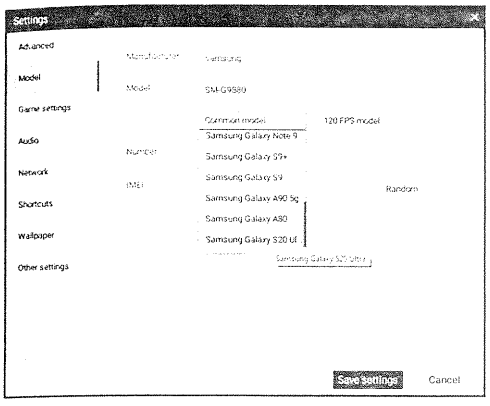
7. **Interact with the Virtual Device:**
   - Once the AVD is launched, interact with it as one would with a physical Android device.
   - Test apps, perform debugging, and evaluate the user experience on the virtual device.
   - By running our first app using newly created device (Pixel 8 Pro) display the "Hello,World!" output in Pixel 8 Pro Emulator.

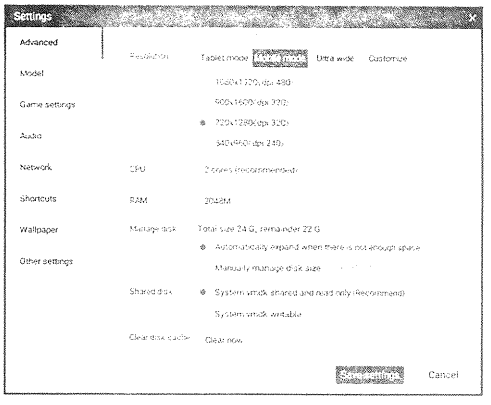
2.7.6 Using Lightweight Emulator (LDPlayer9) for Enhanced Android Development

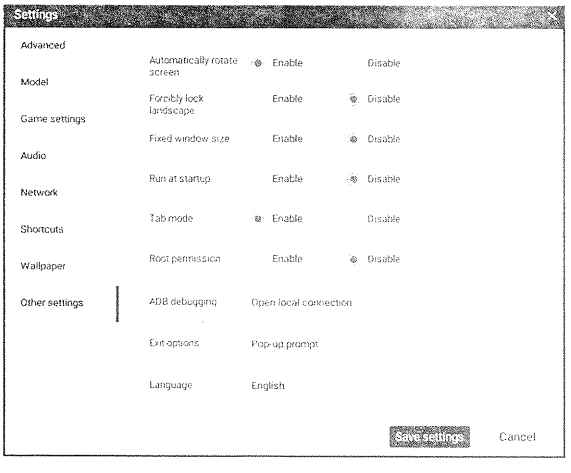
Running an app using a virtual device in Android Studio often encounters performance issues, memory limitations, and system crashes due to the high memory requirements of emulators. A practical solution is to use lightweight external Android emulators like LDPlayer9. Let us see how to install and set up LDPlayer9 to work with Android applications:

1. **Download LDPlayer9:** Visit the LDPlayer9 official website (https://www.ldplayer.net) and download the installer.
2. **Install LDPlayer9:** Run the downloaded installer and follow the on-screen instructions to complete the installation process.
3. **Set Up LDPlayer9:** Launch LDPlayer9 and click on Settings. Configure it according to the desired specifications (mobile mode, screen resolution, RAM allocation) as shown below.

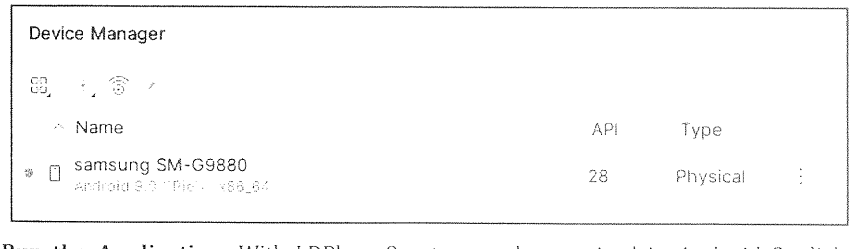




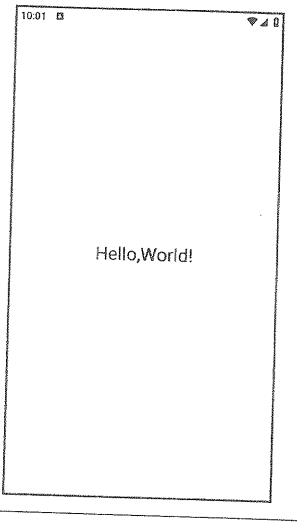




4. **Integrate with Android Studio:**
   - Open Android Studio and navigate to the Device Manager.
   - LDPlayer9 should be recognized as a connected device as shown below.



5. **Run the Application:** With LDPlayer9 set up and recognized in Android Studio's Device Manager, run the Android application from Android Studio and select LDPlayer9 as the target device.



Note:

Please note that all the mobile applications in this book underwent testing using LDPlayer9 as well as a physical device.

2.7 Testing Using a Physical Device

To connect a physical device and run an application on it using an integrated development environment (IDE) like Android Studio, follow these steps:

Step 1: Connect Your Device:

1. **Use a USB Cable:** Connect your physical device to your computer or laptop using a USB cable.

Step 2: Enable Developer Mode and USB Debugging

1. **Enable Developer Mode:**
   - Go to your device's settings menu.
   - Navigate to the 'About phone' section.
   - Tap the 'Build number' multiple times until developer mode is enabled.
2. **Enable USB Debugging:**
   - Go back to the main settings menu.
   - Find and open the 'Developer options' menu.
   - Enable 'USB debugging'.

Step 3: Set up the device in Android Studio:

1. Open Android Studio.
2. Click on the "Run" button in the toolbar.
3. Select the connected physical device as the deployment target.

Step 4: Run the application on the physical device:

1. Android Studio will build the application and deploy it to the connected physical device.
2. The application will run on the physical device, and you can interact with it directly on the device.

2.7.8 Creating Global gradle.properties File

A global gradle.properties file can be created to enhance the performance of Gradle in Android Studio. This file centralizes configuration settings, eliminating the need to set properties for each individual project. The file should be located under C:\Users\<user profile>\.gradle. (For example, C:\Users\Srikanth\.gradle).

Code gradle.properties

```
org.gradle.jvmargs=-Xmx2048m -XX:MaxPermSize=512m -Dfile.encoding=UTF-8
org.gradle.parallel=true
android.useAndroidX=true
android.nonTransitiveRClass=true
org.gradle.configureondemand=true
org.gradle.daemon=true
org.gradle.configuration-cache=true
org.gradle.caching=true
```

These properties improve Gradle's performance and optimize the build process in Android Studio.

Enabling Gradle Offline Mode

Enabling Gradle Offline Mode can be beneficial in environments with limited or no internet access as it improves build performance by utilizing dependencies stored locally instead of making network requests. To toggle Gradle Offline Mode in Android Studio, follow these steps:

1. **Open Android Studio**
2. **Navigate to the Gradle Tab:**
   - In the upper right corner of the Android Studio window, locate the Gradle tab and click on it to open the Gradle tool window.
3. **Toggle Offline Mode:**
   - Within the Gradle tool window, find the "Toggle Offline Mode" button. Click on this button to switch Gradle between online and offline modes.
   - The highlighted or activated button indicates that Gradle Offline Mode is enabled, allowing Gradle to utilize locally cached dependencies.
   - Clicking the button again will toggle Gradle back to online mode, where it fetches dependencies from remote repositories as needed.

It is important to note that during the initial project creation and build process, it is recommended to disable Gradle Offline Mode. This allows Gradle to fetch the necessary dependencies from the remote repositories. Once all the dependencies are downloaded and stored locally, subsequent builds

can benefit from enabling Offline Mode. By enabling Offline Mode after the initial setup, Gradle can efficiently utilize the locally cached dependencies without the need for repeated network requests, thereby expediting the build process in offline environments.

2.8 Debugging your Android Application in Android Studio

Debugging is an essential part of the app development process, allowing developers to identify and fix issues in their code. Android Studio provides a range of powerful debugging tools to help diagnose and resolve problems in Android applications. These tools include breakpoints, which can pause code execution at specific points, and Logcat, which displays system messages and log output from running applications. Additionally, the Android Profiler offers insights into CPU, memory, and network usage, helping to optimize app performance.

2.8.1 Overview of Debugging Tools in Android Studio

Android Studio offers a comprehensive set of debugging tools to help track down and fix bugs in code. Some key tools include:

1. **Logcat :** Logcat is a logging tool that displays messages from the Android system and the app. It allows developers to see what's happening in the app, including any error messages or warnings.
2. **Debugger :** The debugger allows to pause the app at specific points in the code and inspect the state of the app, including the values of variables.
3. **Profiler :** The profiler helps understand the performance of the app, including CPU, memory, and network usage. It can help identify performance bottlenecks and optimize the app.
4. **Layout Inspector :** The Layout Inspector allows inspection of the layout hierarchy of the app at runtime to identify layout issues.

2.8.2 Steps to Debug an Android Application in Android Studio:

We can debug Android applications in Android Studio to identify and resolve issues efficiently and to ensure the smooth functioning of their apps before releasing them to users. The steps in debugging the application are shown below.

1. **Set Breakpoints:**

   One of the most powerful debugging features in Android Studio is the ability to set breakpoints in code. Breakpoints allow the app to pause at specific points to inspect the state of the app.

   - **To Set Breakpoints:** Click in the left gutter next to a line of code to set a breakpoint. A red dot will appear.
   - **Manage Breakpoints:** View and manage breakpoints via Run > View Breakpoints.

   Once a breakpoint is set, the app can be run in debug mode by clicking on the "Debug" button in Android Studio. When the app reaches the breakpoint, it will pause to inspect the values of variables, step through the code, and see the state of the app.

2. **Connect a Device or Start an Emulator:**
   Before initiating the debugging process, developers need to connect a physical Android device to the computer via USB or start an Android Emulator from Android Studio's Device Manager. This connected device or emulator will be used to run and debug the application.
3. **Run the App in Debug Mode:**
   Developers can run the Android application in debug mode by clicking on the "Debug" icon (bug symbol) in the top toolbar of Android Studio or by using the shortcut Shift + F9. They can select the connected device or emulator to deploy the application for debugging.
4. **Navigate Through the App:**
   Once the application is running in debug mode on the device or emulator, developers can interact with the app to trigger the execution of the code. The application will pause at the set breakpoints to analyze the state of variables and understand the program flow.
5. **Inspect Variables and Evaluate Expressions:**
   Developers can utilize the Debug panel in Android Studio to inspect variable values, evaluate expressions, and monitor the application's state during runtime. This feature helps in understanding how data changes as the program executes.
   - Hover over variables to see their values or view them in the Variables pane in the Debug tool window.
   - Add variables to the Watches pane for continuous monitoring.
   - Use the Evaluate Expression tool (accessible from the Debug tool window) to inspect and evaluate expressions during debugging.
6. **Step Through the Code:**
   Android Studio provides various debugging controls such as Step Over, Step Into, and Step Out to navigate through the code line by line. By stepping through the code, developers can trace the execution path and identify any anomalies or errors.
   Use the stepping buttons in the Debug tool window to control code execution:
   - **Step Over (F8):** Moves to the next line of code.
   - **Step Into (F7):** Enters the method on the current line.
   - **Step Out (Shift+F8):** Exits the current method and returns to the caller.
7. **Evaluate Logs and Exceptions:**
   Developers should check the Logcat tab in Android Studio while debugging to view system logs, error messages, and exceptions thrown during the application's execution. Analyzing logs and exceptions helps in pinpointing issues and understanding the root cause of errors.
   - Open the Logcat tool window (View > Tool Windows > Logcat) to see log messages from the running app.
   - Use filters to narrow down logs by tag, level, or keywords.
8. **Use Profiling Tools:**
   For advanced debugging and optimization, developers can leverage Android Profiler in Android Studio. This tool provides insights into app performance, CPU usage, memory allocation, and network activity, helping developers optimize their applications for better efficiency and user experience.

- Use the Android Profiler tool (View > Tool Windows > Profiler) to analyze CPU, memory, and network performance.
- Record and inspect method traces, memory allocations, and network activity.

9. **Modify Code and Re-Test:**

   Based on the insights gained from debugging, developers can make necessary code modifications to address issues. After making changes, developers should remove breakpoints, re-run the application in debug mode, and verify that the fixes have resolved the identified problems.

2.9 Publishing your Android Application

Publishing an Android application on the Google Play Store is a crucial step in making an app available to users worldwide. It involves preparing the app for distribution, adhering to Google Play policies, and ensuring a smooth deployment process. It is the final step in the development process. Understanding the key elements and steps involved in publishing an Android app is essential for developers to reach a wider audience and maximize app visibility.

2.9.1 Important Terms and Key Elements in Publishing an Android Application

When publishing an Android application, understanding key concepts and terms is essential.

1. **APK (Android Package) :** APK stands for Android Package Kit. It is the file format used by the Android operating system for distribution and installation of mobile apps and middleware. An APK file contains all the elements that an app needs to install correctly on a device, including the app's code, resources, assets, certificates, and manifest file. When developers build an Android app, the Android SDK tools compile the code and package it into an APK, which can then be tested, shared, or uploaded to app distribution platforms like Google Play. For example, when building a simple calculator app, the resulting APK contains the app's code, images, and layout files.
2. **Google Play Developer Console :** The Google Play Developer Console is a platform provided by Google that allows developers to manage their Android applications on the Google Play Store. Key functionalities include:
   - **App Listings:** Create and manage app details, descriptions, screenshots, and promotional graphics.
   - **Releases:** Handle app releases across various channels like Alpha, Beta, and Production.
   - **App Signing:** Securely sign and manage app updates.
   - **Performance Monitoring:** Track app performance, crashes, and user feedback.
   - **Monetization:** Set app pricing, manage subscriptions, and view revenue reports.

   The console provides comprehensive tools to ensure efficient app management and optimization.
3. **Store Listing :** The Store Listing in the Google Play Console is the section where developers provide key information about their app for display on the Google Play Store. This includes:

- **App Name:** The name of the app as it appears on the store.
- **Description:** A detailed description of what the app does and its key features.
- **Screenshots and Promotional Graphics:** Visuals that showcase the app's interface and functionality.
- **App Icon:** The icon that represents the app on the store and on user devices.
- **Category:** The classification of the app (e.g., Education, Games).
- **Contact Information:** Support email and website for user assistance.
- **Privacy Policy:** A link to the app's privacy policy.

Creating a compelling store listing is essential for attracting users and providing them with a clear understanding of what the app offers.

4. **App Permissions :** App Permissions are declarations in the app's manifest file that request access to specific device features or user data. These permissions are essential for the app to function correctly and provide intended features. Examples include:
   - **Internet Access:** Required for apps that need to connect to the internet.
   - **Camera:** Needed for apps that use the device's camera.
   - **Location:** Necessary for apps that provide location-based services.
   - **Storage:** Allows apps to read and write to the device's storage.

   For example, a photo editing app might request permissions to access the camera and storage. Permissions must be clearly stated and justified to users to maintain trust and comply with privacy regulations. For detailed information, visit the official documentation.
5. **Release Channels :** Release Channels in the Google Play Console allow developers to distribute app updates to different user groups at various testing stages before a full production release. The primary channels include:
   1. **Internal Testing :** For quick, small-scale testing with internal team members.
   2. **Closed Testing (Alpha) :** For a limited number of trusted testers to get early feedback.
   3. **Open Testing (Beta) :** For a broader audience, allowing more user feedback while still being in a controlled environment.
   4. **Production :** The final release available to all users on the Google Play Store.

   Managing release channels effectively helps ensure app stability and quality before reaching a wide audience.
6. **App Signing :** App Signing is the process of digitally signing an APK to verify its authenticity and ensure its integrity. This is a critical security step for distributing Android applications.
   1. **Keystore Creation :** A keystore is a secure file that contains the cryptographic keys used to sign the app.
   2. **Signing the APK :** The APK is signed with the keystore's private key, generating a unique signature.
   3. **Verification :** When the app is installed, Android verifies the signature using the corresponding public key to ensure the app has not been tampered with.

2.9.2 Steps in Publishing Android Application

Publishing an Android application involves several crucial steps to ensure it is ready for distribution on the Google Play Store. The Key steps are :

Step 1: Preparing An App for Release

Before publishing an app, several steps must be completed to ensure it is ready for release. This involves:

1. **Testing :** Ensure the app is thoroughly tested on various devices and screen sizes to confirm it works correctly under different conditions.
2. **Optimization:** Optimize the app for performance, reduce app size, and improve startup times. For example, Use ProGuard to shrink and obfuscate code.
3. **Security :** Verify that the app is secure and free from vulnerabilities that could compromise user data.
4. **Compliance:** Ensure the app complies with all Google Play Store policies and guidelines.
5. **Update Version Information:** In the build.gradle.kts file, update the version code and version name. These values help track the app's versions and manage updates.

```
android {
        defaultConfig {
            versionCode 2       // Increment this for each release
            versionName "1.1"   // Update to reflect the new version
        }
}
```

   - **Version Code:** An integer value that must be incremented with each release. It is used to determine whether one version is more recent than another.
   - **Version Name:** A string that represents the release version of the app. This is visible to users.

6. **Check Permissions** Ensure all necessary permissions are included in the AndroidManifest. xml file. Permissions are essential for accessing device features and data.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.myapp">
      <uses-permission android:name="android.permission.INTERNET"/>
      <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
            ...
</manifest>
```

Common Permissions:

- **INTERNET:** Needed for network access.
- **ACCESS_FINE_LOCATION:** Required for precise location data.
- **READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE:** Needed for accessing and modifying external storage.

Ensure Permissions: Only request permissions that are necessary for the app's functionality to maintain user trust and comply with privacy regulations.

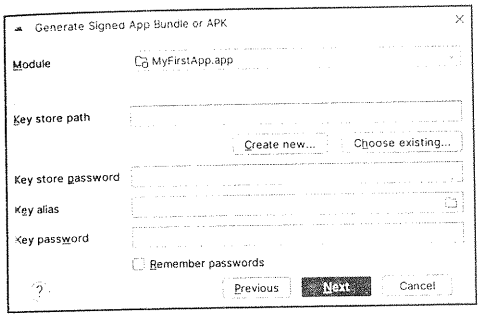
Step 2: Generating a Signed APK

To publish an app on the Google Play Store, we need to generate a signed APK (Android Package) file. This file is signed with a private key to ensure it comes from a trusted source.

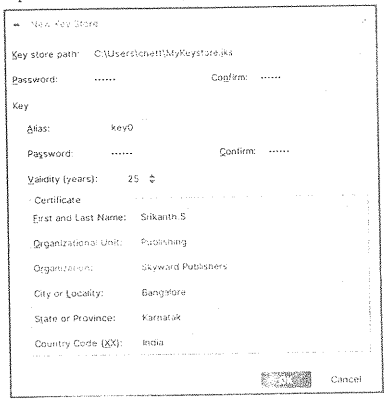
To generate a signed APK, follow these steps:

(a) In Android Studio, go to Build > Generate Signed Bundle / APK.

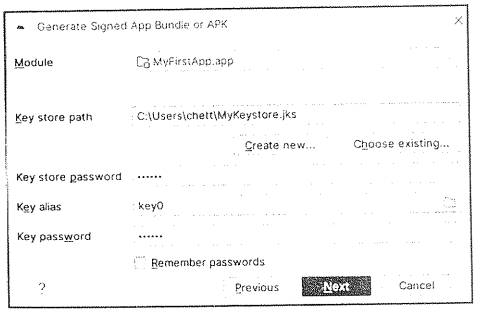
(b) Select "APK" and click "Next."



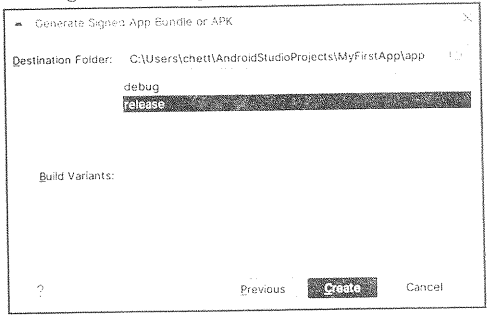
(c) If No Keystore Exists: Click Create new... and fill in the required details (location, passwords, alias).



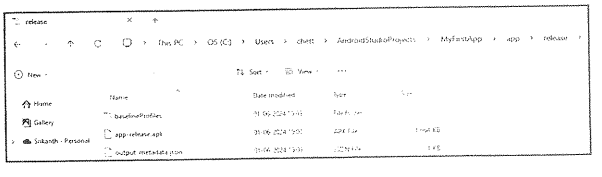
(d) Select keystore file, enter the keystore password, and the alias and click Next.



(e) Select the build type (e.g., release) and click "Finish." Android Studio will generate a signed APK file.



(f) The signed APK file will created as shown below.

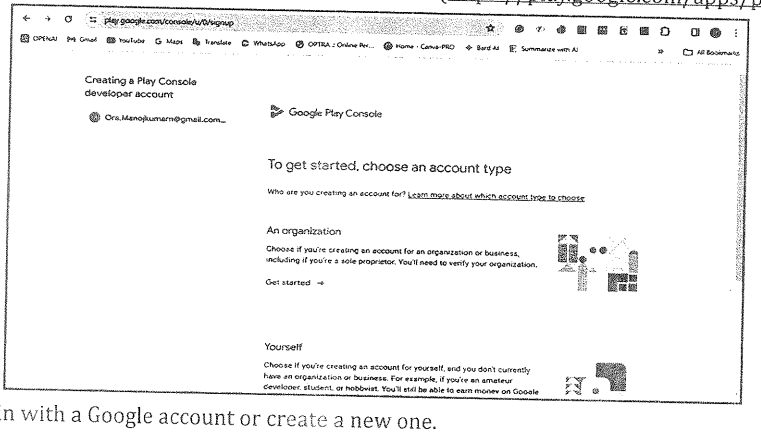


Step 3: Creating a Google Play Developer Account

Before publishing an app on the Google Play Store, we need to create a Google Play Developer account. This requires a one-time registration fee.

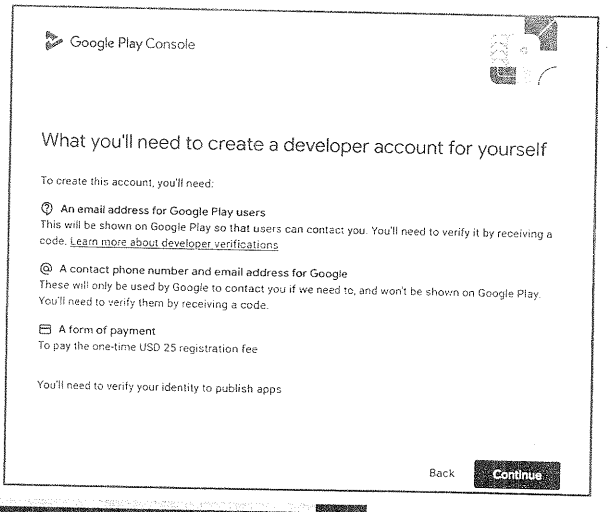
To create a Google Play Developer account, follow these steps:

(a) Go to the Google Play Developer Console website (https://play.google.com/apps/publish/).



(b) Sign in with a Google account or create a new one.

(c) Follow the on-screen instructions to complete the registration process and pay the registration fee.



Step 4: Uploading an App to the Google Play Console

Once we have a signed APK file and a Google Play Developer account, we can upload an app to the Google Play Console. This is the platform where we manage app's listing on the Google Play Store.

To upload an app, follow these steps:

- Log in to the Google Play Console.
- Click on "Create Application" and enter the details of an app, including the title, description, and screenshots.
- Upload signed APK file and provide any additional information requested.

(d) Set the pricing and distribution options for an app.

(e) Click "Save" and then "Publish" to submit an app to the Google Play Store.

Once an app is published on the Google Play Store, we can promote it to attract users. Consider using app store optimization (ASO) techniques to improve app's visibility in the store. We can also promote an app through social media, online advertising, and other channels to reach a wider audience.

2.10 Review Questions

Section -A

1. What is Android?
2. Name two key features of Android and provide examples.
3. What is Native Android Application?
4. What are the main components of an Android application?
5. What is the purpose of an Activity in an Android Application?
6. What is a Content Provider in Android?
7. What is a Broadcast Receiver in Android?
8. How does a Service differ from an Activity in Android?
9. What is the Android SDK?
10. What is Android Studio?
11. What is Android Emulator?
12. What is Android Virtual Device (AVD)?
13. What is the use of Gradle in Android Studio?
14. How do you set a breakpoint in Android Studio?
15. What is an APK?
16. What is the purpose of the Google Play Developer Console?
17. What are the steps to upload an app to the Google Play Console?
18. Define Android.
19. Give an example of how Android supports customization.
20. How does Android's integration with Google services benefit users?
21. What programming languages are used for Android app development?
22. What is Adaptive Battery in Android?
23. Explain the concept of "Multi-Tasking and Split-Screen Mode" in Android.
24. What is the role of an Activity in an Android application?
25. How do Content Providers function in Android?

26. What is the role of the Android Runtime?
27. What is the purpose of the AndroidManifest.xml file?
28. What is the purpose of the Layout Editor in Android Studio?
29. What is ADB (Android Debug Bridge)?
30. How can you view logs in Android Studio?
31. What is the purpose of app signing in Android?
32. How do you create a Google Play Developer account?
33. Describe the flow of execution in an Android application.
34. What is the role of Intent Filters in Android?
35. What are App Permissions? Mention any two permission.

Section - B

1. Explain the Architecture of Android.
2. What is Native Android Application? Explain the Characteristics of Native Android Applications.
3. What is an Android SDK? Write its features.
4. What is an Android Studio? Write its features.
5. Explain the Key Features of Android Emulators.
6. Explain ADB (Android Debug Bridge) and its features.
7. Explain the Required Tools for Android Development.
8. Explain the Flow of Execution of an Android application.
9. Explain the Project Structure in Android Studio.
10. Discuss the Importance (or) Significance of AndroidManifest.xml File.
11. Explain Debugging Tools in Android Studio.

Section - C

1. What is Android? Explain the Key Features of Android OS.
2. Write the Advantages and Disadvantages of Android.
3. Explain the Android Application Components.
4. Explain how to develop an android application to display "Hello World!".
5. What is an IDE? Explain the features of Android Studio IDE.
6. What is the use of Android Manifest File? Explain the components (or) structure of Android Manifest File.
7. Write the Steps to Debug an Android Application in Android Studio.
8. Explain the Steps in Publishing Android Application.

UNIT - I

CHAPTER 3

USING ACTIVITIES, FRAGMENTS AND INTENTS IN ANDROID

CONTENTS

- Introduction
- Working with Activities
- Using Intents
- Fragments
- Understanding the Back Stack in Android
- Managing Activity Navigation and Back Stack
- Review Questions

3.1 Introduction

This chapter focuses on the basic components of an Android app: **activities, fragments,** and **intents.** *An activity serves as a window that displays the user interface, with the primary purpose of facilitating user interaction.* Apps can have many activities, each going through different stages called the activity's lifecycle. Knowing this lifecycle is important for the app to work well.

In addition to activities, this chapter introduces Fragments. Fragments are like smaller parts of activities introduced in Android 3.0 for tablets and improved in Android 4.0. *Fragments can be thought of as smaller, modular sections of activities that can be combined to form a complete activity.* This modularity allows for more flexible and reusable UI components to enhance the application's adaptability to different screen sizes and orientations.

Intents are like connections that link activities within the app or even between different apps. They help in smoothly carrying out tasks, making everything seem like part of one app. Intents are used to interact with built-in apps like the Browser, Phone, and Maps to make the user experience richer and more integrated.

3.2 Working with Activities

Activities represent the user interface (UI) of an application. They are responsible for interacting with users and handling user interactions. Each screen in an Android app is typically implemented as an activity. Each activity is typically designed to handle a specific task or function.Each activity is a subclass of the Activity class and every app must have at least one activity. An activity is similar to a window or page in a desktop application or a webpage in a web application.

What is an Activity in Android?

An activity in Android is a single screen with a user interface where users can interact. It represents a window in an application that displays the UI elements. Each screen in an Android app is typically implemented as an activity.

Example	Activities in WhatsApp Mobile Application

In the context of WhatsApp, an activity refers to a specific screen or view within the WhatsApp application where users can perform various actions and interact with the app's features. Each screen or section in the WhatsApp app, such as the chat screen, contact list, settings screen, etc., is implemented as an activity.

- The chat screen where users can send and receive messages is implemented as a chat activity.
- The contact list screen where users can view their contacts and start new chats is implemented as a contacts activity.
- The settings screen where users can customize their app preferences is implemented as a settings activity.

Each activity serves a distinct purpose and provides a platform for users to engage with the app's functionalities.

3.2.1 Creating an Activity

Creating and managing activities is a fundamental aspect of Android app development. Activities are the building blocks of an Android app, representing individual screens that users interact with. To create a new activity in an Android app, follow these steps:

Step 1: Create a New Java or Kotlin Class for an Activity:

In Android Studio, right-click on app's package name in the Project pane, select "New," then "Activity," and choose the activity template to use (Example, "Empty Activity," "Empty Views Activity," etc.).

Example Creating an Activity (MainActivity.java)

```
package com.example.myfirstapp;

import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Explanation

- **package com.example.myfirstapp :** Defines the package name for the application.
- **import android.app.Activity :** Imports the Activity class.
- **public class MainActivity extends Activity:** Defines the MainActivity class that extends Activity.
- **onCreate(Bundle savedInstanceState):** The onCreate() method is where the activity is initialized.
  - It sets the user interface layout for this activity using setContentView(R.layout.activity_main).
  - This method is called when the activity is first created for any initial setup tasks, such as initializing variables, setting up listeners, or loading data.
  - super.onCreate(savedInstanceState); is a call to the superclass implementation of the onCreate() method. This call is essential as it ensures that the superclass (Activity or AppCompatActivity) can perform necessary initialization tasks before any custom initialization code in the subclass's onCreate() method is executed.
  - The savedInstanceState parameter is a Bundle that provides the activity's previously saved state. It is useful for the restoration of the activity to its previous state if necessary.

Step 2: Defining the Layout (XML):

Create an XML layout file for an activity's UI in the res/layout directory. We can design the layout using the visual editor in Android Studio or by editing the XML directly.

Example Creating a XML Layout File (activity_main.xml)

```
<!-- res/layout/activity_main.xml -->
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me" />
</LinearLayout>
```

Explanation

- **LinearLayout:** A layout that arranges its children in a single column or row.
- **TextView:** A UI element that displays text.
- **Button:** A UI element that can be clicked to perform an action.

Step 3: Declare the Activity in the Manifest:

Every activity must be declared in the AndroidManifest.xml file. Open the AndroidManifest.xml file and add an <activity> element for the new activity. Specify the activity's name and the main launcher intent filter if it's the entry point of an app.

Example Declare the Activity in the Manifest File - AndroidManifest.xml

```
<!-- AndroidManifest.xml -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myfirstapp">

   <application
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name">
        <activity android:name=".MainActivity">

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Explanation

- **<application>:** Declares the application.
- **<activity android:name=**".MainActivity"**>:** Declares the activity.
- **<intent-filter>:** Specifies that this activity is the main entry point of the app (launched from the home screen).

3.2.2 Understanding Activity Class

An Android activity is a crucial component of an Android app and it represent a single screen with a user interface. Each activity is a subclass of the **Activity class** and is a fundamental building block for creating a responsive and interactive user experience. Let us understand the basic structure and essential methods needed for a functional activity.

Code MainActivity.java

```
package com.example.myfirstapp;

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

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Set the content view to a specific layout
        setContentView(R.layout.activity_main);

        // Find the TextView in the layout and set its text
        TextView textView = findViewById(R.id.text_view);
        textView.setText("Hello, World!");
    }
}
```

Explanation

1. **Package Declaration:** Declares the package name, organizing classes and avoiding name conflicts.
2. **Import Statements:**
   - Imports necessary classes from the Android framework:
   - 'android.os.Bundle': For saving the state of an activity.
   - 'android.widget.TextView': For displaying text in the UI.
   - 'androidx.appcompat.app.AppCompatActivity': Base class for activities with support for modern Android features.
3. **Class Declaration:** Declares the 'MainActivity' class, extending 'AppCompatActivity' to inherit activity properties and behaviors.

4. **onCreate Method:** Lifecycle method called when the activity is first created.
   - '@Override' indicates this method overrides a superclass method.
   - 'super.onCreate(savedInstanceState)' calls the superclass's initialization method.
5. **Setting the Content View:** Sets the activity's user interface layout using 'R.layout.activity_main', referring to the XML layout file 'activity_main.xml'.
6. **Finding a View and Setting Text:**
   - 'findViewById' retrieves a view identified by the XML layout's ID attribute ('R.id.text_view').
   - 'setText("Hello, World!")' sets the 'TextView' text to "Hello, World!".

3.2.3 Activity Life Cycle

The lifecycle of an Android activity represents the series of stages an activity goes through from its creation to its destruction. Understanding the lifecycle is crucial for managing resources to ensure smooth user experience, and handling different states of an app effectively.

Activity Lifecycle Stages

1. **onCreate()**
   - **Description:** Called when the activity is first created.
   - **Purpose:** Initialize the activity. This is where you should perform all the initial setup: create views, bind data, etc.

Code

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

2. **onStart()**
   - **Description:** Called when the activity becomes visible to the user.
   - **Purpose:** Perform tasks that need to be done each time the activity becomes visible, like refreshing UI components.

Code

```
@Override
protected void onStart() {
    super.onStart();
    // Activity is now visible
}
```

3. **onResume()**
   - **Description:** Called when the activity starts interacting with the user.
   - **Purpose:** Resume any paused UI updates, animations, or tasks that need to run while the activity is in the foreground.

Code

```
@Override
protected void onResume() {
    super.onResume();
    // Resume updates, animations, or tasks
}
```

4. **onPause()**
   - **Description:** Called when the system is about to start resuming another activity.
   - **Purpose:** Pause ongoing tasks, animations, or other operations that should not continue while the activity is in the background. Save any persistent data.

Code

```
@Override
protected void onPause() {
    super.onPause();
    // Pause tasks, animations, etc.
}
```

5. **onStop()**
   - **Description:** Called when the activity is no longer visible to the user.
   - **Purpose:** Release resources that are not needed while the activity is not visible, such as stopping background processes.

Code

```
@Override
protected void onStop() {
    super.onStop();
    // Stop background processes
}
```

6. **onDestroy()**
   - **Description:** Called before the activity is destroyed.
   - **Purpose:** Clean up resources, save final states, and perform any necessary shutdown operations.

Code

```
@Override
protected void onDestroy() {
    super.onDestroy();
    // Clean up resources
}
```

7. **onRestart()**
   - **Description:** Called when the activity has been stopped and is restarting again.
   - **Purpose:** Perform any necessary actions to reinitialize the activity that was stopped.

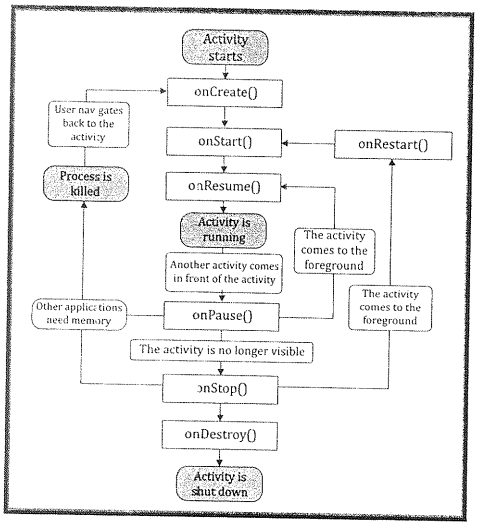
Code

```
@Override
 protected void onRestart() {
     super.onRestart();
     // Reinitialize the activity
 }
```

Visual Representation

The below figure shows the life cycle of an activity and the various stages it goes through from when the activity is started until it ends.

1. **Activity Starts - 'onCreate()':**
   - **Description:** Called when the activity is first created. It is where static setup tasks are performed, such as creating views and binding data.
   - **Purpose:** Initialize the activity, inflate the UI, and execute any one-time setup procedures.
2. **Activity is Being Created - 'onStart()':**
   - **Description:** Invoked when the activity becomes visible to the user.
   - **Purpose:** Make the activity visible to the user. It is visible but not in the foreground or interacting with the user.



3. **Activity Comes to the Foreground - 'onResume()':**
   - **Description:** Called when the activity starts interacting with the user.
   - **Purpose:** Bring the activity to the foreground and make it interactive.
4. **Activity is Running:**
   - **State:** The activity is now running, and the user can interact with it.

5. **Another Activity Comes in Front - 'onPause()':**
   - **Description:** Triggered when another activity is about to resume, partially obscuring the current activity.
   - **Purpose:** Save changes, stop animations, or perform actions that should not continue while the activity is in the background.
6. **Activity is No Longer Visible - 'onStop()':**
   - **Description:** Called when the activity is no longer visible to the user.
   - **Purpose:** Release unnecessary resources, save data to storage, etc.
7. **Activity Comes to the Foreground Again - 'onRestart()':**
   - **Description:** Invoked when the activity has been stopped and is restarting.
   - **Purpose:** Prepare the activity for restarting after being stopped.
8. **Process is Killed or User Navigates Back:**
   - 'onStart()' and 'onResume()' are called again to make the activity visible and interactive.
9. **Activity is Shut Down - 'onDestroy()':**
   - **Description:** Called before the activity is destroyed.
   - **Purpose:** Clean up resources, end threads, close databases, etc. This is the final method called before the activity is destroyed.

Example | **Understanding Life Cycle of an Activity**

Let's consider a scenario where a messaging app, similar to WhatsApp, has an activity that displays a user's chat screen. This example will explain how the activity lifecycle stages play out during various user interactions.

1. **User Opens the App**

   When the user first opens the app, the 'onCreate()' method is called. This method is responsible for initializing the activity. It sets up the user interface layout and loads the chat messages from the database or server.

   Lifecycle Method Involved: 'onCreate()'
   - **Purpose:** Initialize the activity, set up the UI, and load data.
   - **Example:** The chat screen layout is loaded, and messages are fetched to display.
2. **Activity Becomes Visible and Interactive**

   After 'onCreate()', the 'onStart()' method is called, making the activity visible to the user. Immediately after, the 'onResume()' method is called, making the activity interactive. At this point, the user can see the chat messages and interact with the chat screen (e.g., sending and receiving messages).

   Lifecycle Methods Involved: 'onStart()' and 'onResume()'
   - **onStart():** Makes the activity visible.
   - **onResume():** Makes the activity interactive, allowing user input and interaction.
3. **User Receives a Phone Call**

   When the user receives a phone call, the 'onPause()' method is called. This method pauses ongoing processes such as message fetching or animations. If the phone call application completely covers the chat screen, the 'onStop()' method is then called, releasing resources that aren't needed when the chat screen is not visible.

Lifecycle Methods Involved: 'onPause()' and 'onStop()'

- **onPause():** Pause any processes that should not continue while the activity is not in the foreground.
- **onStop():** Release resources to optimize performance since the activity is no longer visible.

4. **User Finishes the Call and Returns to the App**

   Once the user finishes the call and returns to the chat screen, the 'onRestart()' method is called. This method prepares the activity to be started again. Following 'onRestart()', the 'onStart()' and 'onResume()' methods are called to make the chat screen visible and interactive again.

   Lifecycle Methods Involved: 'onRestart()', 'onStart()', and 'onResume()'

   - **onRestart():** Prepare the activity to be started again.
   - **onStart():** Make the activity visible again.
   - **onResume():** Make the activity interactive again.

5. **User Exits the App**

   If the user decides to exit the app (e.g., by pressing the back button or swiping the app away from the recent apps list), the 'onPause()' method is called first to pause ongoing tasks. Next, the 'onStop()' method is called to release resources, and finally, the 'onDestroy()' method is called to clean up any remaining resources before the activity is completely removed from memory.

   Lifecycle Methods Involved: 'onPause()', 'onStop()', and 'onDestroy()'

   - **onPause():** Pause tasks that do not need to run when the user is not interacting with the activity.
   - **onStop():** Release resources to optimize performance.
   - **onDestroy():** Clean up any remaining resources and perform any final shutdown operations.

3.2.4 Features and Functionalities of an Activity

In Android development, an 'Activity' is a fundamental component of an application that represents a single screen with a user interface. Activities play a crucial role in the Android app lifecycle and provide various features that allow developers to create interactive and engaging user experiences. Here are some key features and functionalities of an 'Activity' in Android:

1. **User Interface (UI):**
   - Activities are used to present a visual interface to the user, typically defined in XML layout files.
   - Developers can customize the UI elements such as buttons, text fields, images, etc., to create interactive screens.
2. **Lifecycle Management:**
   - Activities have a well-defined lifecycle consisting of methods like 'onCreate()', 'onStart()', 'onResume()', 'onPause()', 'onStop()', 'onDestroy()', etc.
   - Developers can override these lifecycle methods to perform specific actions at different stages of the activity lifecycle.
3. **Navigation:**
   - Activities can be started, paused, resumed, stopped, and destroyed based on user interactions and system events.

- Developers can navigate between activities using explicit intents to switch from one screen to another.

4. **Intent Handling:**
   - Activities can receive intents to perform actions like opening a new activity, sharing data, or responding to system events.
   - Developers can extract data from intents to customize the behavior of the activity.
5. **Back Stack:**
   - Activities are managed in a back stack, allowing users to navigate back through the history of screens using the device's back button.
   - Developers can control the back stack behavior by defining the parent activity or setting flags in the intent.
6. **Communication:**
   - Activities can communicate with other components such as services, broadcast receivers, and content providers using intents.
   - Developers can pass data between activities using intent extras or by starting activities for results.
7. **Configuration Changes:**
   - Activities handle configuration changes like screen orientation, language, and keyboard availability by default.
   - Developers can manage configuration changes by overriding methods like 'onConfigurationChanged()' or handling them using resources.
8. **Result Handling:**
   - Activities can return results to the calling activity using the 'startActivityForResult()' method.
   - Developers can set a result code and data to be sent back to the calling activity when the child activity finishes.
9. **Permissions:**
   - Activities can request permissions from the user to access sensitive device features like camera, location, contacts, etc.
   - Developers can handle permission requests and responses to ensure the app functions correctly and securely.
10. **Task Affinity:**
    - Activities can have a task affinity that defines how they interact with other activities in the same task.
    - Developers can set task affinity in the manifest file to control the behavior of activities within the task.

These features make activities a versatile component in Android development, allowing developers to create dynamic and interactive applications with multiple screens and user interactions.

3.2.5 Applying Styles and Themes to an Activity

The styles and themes are powerful tools to define and apply consistent visual styles across an application or specific components like activities. Understanding styles and themes is essential for creating a visually appealing user interface.

1. **Styles:**
   - **Definition:** A style in Android is a collection of properties that specify the look and format for a View or a window. It can be used to share common attributes across multiple elements to ensure a consistent look and feel throughout the application.
   - **Usage:** Styles can be applied to individual views in layout XML files or set programmatically in Java/Kotlin code.
   - **Purpose:**
     - **Reusability:** Styles enable the reuse of a set of properties across different UI components to promote a consistent design and reducing redundancy.
     - **Customization:** They allow for easy customization and updates. Changing a style definition updates all components using that style.
   - **Components:**
     - **Style Resource:** Defined in an XML file under the res/values directory, typically in themes.xml.
     - **Attributes:** Consist of key-value pairs that define various properties like layout_width, layout_height, textColor, background, etc.
2. **Themes:**
   - **Definition:** A theme in Android is a style applied to an entire Activity or application, rather than an individual View. Themes are used to define the look and feel of all UI elements within the scope they are applied to. Themes can include styles for various UI components like buttons, text fields, dialogs, etc.
   - **Purpose:**
     - **Global Consistency:** Themes ensure a uniform appearance across the entire application or specific activities.
     - **Customization:** They allow for the definition of global attributes such as color palettes, font styles, and dimensions, which can be easily changed in one place.
   - **Components:**
     - **Theme Resource:** The themes are defined in themes.xml file within the res/values directory.
     - **Inheritance:** Themes can inherit properties from other themes, allowing for hierarchical and layered customization.
   - **Application Level vs. Activity Level:** Themes can be applied at the application level in the AndroidManifest.xml file or at the activity level in layout XML files.
   - **Customization:** Themes can be customized to create unique visual experiences for different parts of the application.

Example Applying a Style and Theme to an Activity

This example demonstrates how to define a simple style for a button and a theme for the application. The style is applied to the button in the layout, and the theme is set for the entire application in the manifest file. This ensures a consistent look and feel with minimal code.

Step 1: Create Colors in 'colors.xml'

Define the colors to be used in the style and theme in colors.xml file located under res/values folder.

Code colors.xml

```
<resources>
      <color name="colorPrimary">#6200EE</color>
      <color name="colorPrimaryDark">#3700B3</color>
      <color name="colorAccent">#03DAC5</color>
</resources>
```

Step 2: Define a Simple Style and Theme in themes.xml file

Define a single style and theme for demonstration purposes in themes.xml file located under res/value/themes/themes.xml

Code themes.xml

```
<resources>
<!-- Base application theme -->
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
      <item name="colorPrimary">@color/colorPrimary</item>
      <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
      <item name="colorAccent">@color/colorAccent</item>
</style>
<!-- Custom Button style -->
<style name="CustomButtonStyle">
      <item name="android:layout_width">wrap_content</item>
      <item name="android:layout_height">wrap_content</item>
      <item name="android:background">@color/colorAccent</item>
      <item name="android:textColor">#FFFFFF</item>
      <item name="android:padding">10dp</item>
</style>
</resources>
```

Step 3: Apply Styles to Layout in acivity_main.xml

Apply the defined style to a button in acivity_main.xml file located under res/layout

Code acivity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      tools:context=".MainActivity">
```

```
<Button
      android:id="@+id/button"
      style="@style/CustomButtonStyle"
      android:text="Click Me"
      android:layout_centerInParent="true"/>

</RelativeLayout>
```

Step 4: Apply the Theme in the Manifest

Set the theme for the entire application in the AndroidManifest.xml file.

Code AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.myfirstapp">
      <application
         android:allowBackup="true"
         android:icon="@mipmap/ic_launcher"
         android:label="@string/app_name"
         android:roundIcon="@mipmap/ic_launcher_round"
         android:supportsRtl="true"
         android:theme="@style/AppTheme">
         <activity android:name=".MainActivity">
            <intent-filter>
               <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
         </activity>
      </application>

</manifest>
```

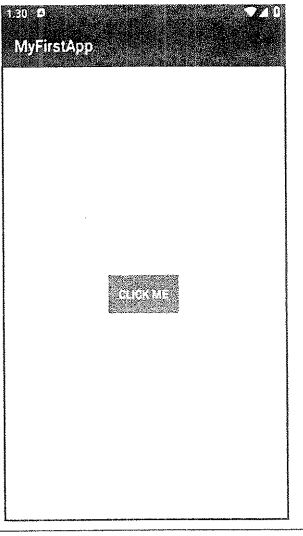
Step 5: Implement MainActivity

Create the MainActivity.java with basic setup.

Code MainActivity.java

```
package com.example.myfirstapp;
import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;

      public class MainActivity extends AppCompatActivity {

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

Output



3.2.6 Hiding the Activity Title

To hide the title of an activity in an Android application, we can set the 'android:theme' attribute in the 'AndroidManifest.xml' file for that specific activity. The steps to hide the activity title are shown below.

1. **Create a Custom Theme:**
   - Define a custom theme in the 'res/values/themes/themes.xml' file if it doesn't exist already. We can create a new style.

Code **themes.xml**

```
<!-- res/values/styles.xml -->
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <!-- Define any other style attributes here -->
</style>

<style name="AppTheme.NoActionBar">
    <item name="windowNoTitle">true</item>
    <item name="windowActionBar">false</item>
</style>
```

2. **Apply the Custom Theme to the Activity:**
   - In the 'AndroidManifest.xml' file, specify the custom theme for the activity where we want to hide the title.

Code AndroidManifest.xml

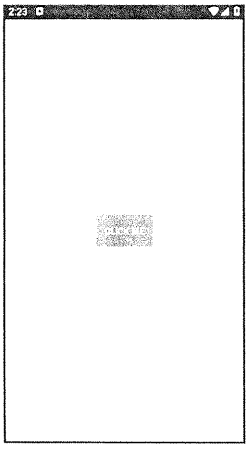
```
<!-- AndroidManifest.xml -->
<activity android:name=".YourActivity"
          android:theme="@style/AppTheme.NoActionBar">
    <!-- Other activity attributes -->
</activity>
```

3. **Alternative Approach - Programmatically:**
   - If we prefer to hide the title bar programmatically in the activity code, then we can use the 'requestWindowFeature(Window.FEATURE_NO_TITLE)' method before calling 'setContentView()'.

Code MainActivity.java

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    requestWindowFeature(Window.FEATURE_NO_TITLE); // Hide the title
    setContentView(R.layout.activity_main);
}
```

By applying either of these methods, we can hide the title of an activity in an Android application as shown in the below output . The XML-based approach using themes is a recommended way to manage the appearance of multiple activities consistently. It's important to note that the last option works only when no themes are applied, as themes take more priority in defining the appearance of activities.



3.2.7 Displaying a Dialog Window Using an Activity

A dialog in Android is a small window that prompts the user to make a decision or enter additional information. Dialogs do not fill the screen and are usually modal, meaning they require the user to interact with them before they can return to the parent activity. Dialogs are useful for:

- **Alerts:** Informing users of events that need their attention.
- **Confirmations:** Asking users to confirm an action.
- **Inputs:** Collecting user input without leaving the current context.

Types of Dialogs in Android

There are several types of dialogs in Android:

- **AlertDialog:** AlertDialog is a versatile dialog that can show a title, message, and buttons. It is used for prompting user actions, displaying information, or gathering input.
- **DatePickerDialog:** A dialog that allows users to select a date. It provides a user-friendly interface for date selection, showing a calendar view where users can pick a date.
- **TimePickerDialog:** A dialog that allows users to select a time. It can be configured to use either a 24-hour format or an AM/PM format, depending on the application's requirements.
- **Custom Dialogs:** Fully customized dialogs using custom layouts. Custom dialogs provide complete flexibility, allowing developers to design and implement any user interface within the dialog.
- **BottomSheetDialog :** A BottomSheetDialog is a dialog that slides up from the bottom of the screen. BottomSheetDialog provides a modern way to present actions or options to users, often used for actions related to the current content.

Example Displaying an AlertDialog

This example will demonstrate how to display an 'AlertDialog' when a button is clicked. The dialog will inform the user that the button has been clicked.

1. **Define the Layout:** Create a activity_main.xml with a button that will trigger the dialog.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button"
        style="@style/CustomButtonStyle"
        android:text="Click Me"
        android:layout_centerInParent="true"/>

</RelativeLayout>
```

2. **Implement MainActivity:** Set up the button to display a dialog when clicked in MainActivity.java

Code MainActivity.java

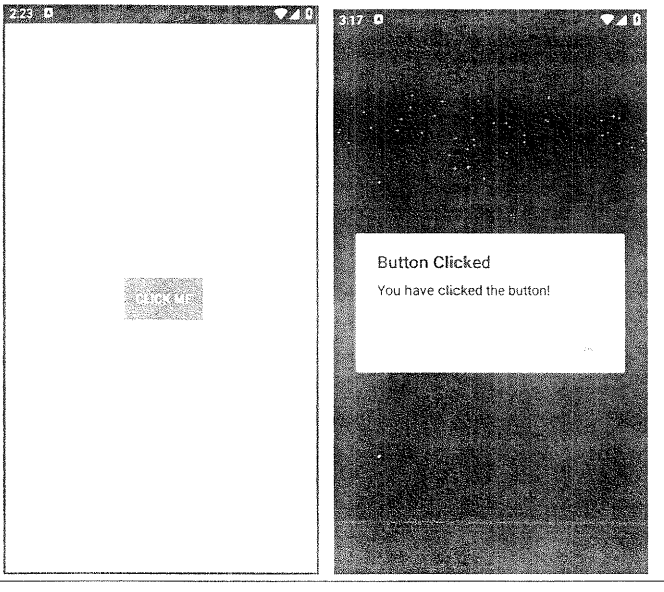
```
package com.example.myfirstapp;

import android.content.DialogInterface;
import android.os.Bundle;
import android.view.View;
```

```
import android.widget.Button;
import androidx.appcompat.app.AlertDialog;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
   @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
       Button showDialogButton = findViewById(R.id.button);
        showDialogButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                showAlertDialog();
            }
        });
    }

    private void showAlertDialog() {
        AlertDialog.Builder builder = new AlertDialog.Builder(this);
        builder.setTitle("Button Clicked")
          .setMessage("You have clicked the button!")
           .setPositiveButton("OK", new DialogInterface.OnClickListener()   {
                    public void onClick(DialogInterface dialog, int id) {
                        dialog.dismiss();
                    }
          });

        // Create and show the AlertDialog
        builder.create().show();
    }
}
```

Explanation

1. **Layout Definition:**
   - The layout file 'activity_main.xml' contains a 'Button' with the text "Click Me".
2. **Activity Implementation:**
   - In 'MainActivity.java', a 'Button' is referenced and a click listener is set up for it.
   - When the button is clicked, the 'showAlertDialog()' method is called.
   - The 'showAlertDialog()' method creates an 'AlertDialog' with a title, message, and an "OK" button. When the "OK" button is clicked, the dialog is dismissed.

Output



3.2.8 ProgressBar in Android

A 'ProgressBar' is a visual indicator of progress in an operation. It can be used to display the status of ongoing tasks, such as loading data from the internet, processing a large file, or any long-running operations that require user awareness.

There are two types of 'ProgressBar' in Android:

1. **Determinate ProgressBar:** Shows the exact progress of an operation, typically as a percentage.
2. **Indeterminate ProgressBar:** Indicates that an operation is in progress but does not show the exact progress.

Example | **Using ProgressBar in Android**

This example demonstrates how to use an indeterminate 'ProgressBar' to indicate loading, which is a common use case in Android applications. The example includes a button titled "Start Payment Transfer" that, when clicked, starts a 10-second delay to simulate a long-running task. After the delay, the message "Payment Transfer Done" is displayed.

1. **Define the Layout:** Create an 'activity_main.xml' with a 'ProgressBar' and a button to start the progress.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    tools:context=".MainActivity">

   <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start Payment Transfer"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp"/>

   <ProgressBar
        android:id="@+id/progressBar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:visibility="gone"
        android:layout_below="@id/button"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp"/>

   <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Payment Transfer Done"
        android:visibility="gone"
        android:layout_below="@id/progressBar"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp"/>
</RelativeLayout>
```

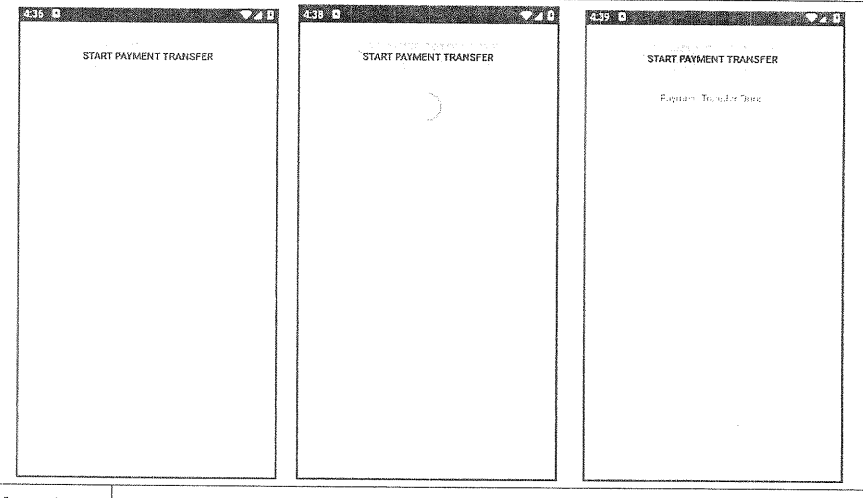
2. **Implement MainActivity:** Set up the button to start and stop the 'ProgressBar' when clicked in 'MainActivity.java'.

Code MainActivity.java

```
package com.example.myfirstapp;

import android.os.Bundle;
import android.os.Handler;
import android.view.View;
import android.widget.Button;
import android.widget.ProgressBar;
```

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

public class MainActivity extends AppCompatActivity {

    private ProgressBar progressBar;
    private Button button;
    private TextView textView;

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

        progressBar = findViewById(R.id.progressBar);
        button = findViewById(R.id.button);
        textView = findViewById(R.id.textView);

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startPaymentTransfer();
            }
        });
    }

    private void startPaymentTransfer() {
        progressBar.setVisibility(View.VISIBLE);
        textView.setVisibility(View.GONE);

        // Simulate a long-running task
        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                progressBar.setVisibility(View.GONE);
                textView.setVisibility(View.VISIBLE);
            }
        }, 10000); // Stop the progress bar after 10 seconds
    }
}
```

Output



Explanation:

1. **Layout Definition:**
   - The layout file 'activity_main.xml' contains a 'Button' with the text "Start Payment Transfer", an indeterminate 'ProgressBar', and a 'TextView' to display the completion message. The 'ProgressBar' and 'TextView' are initially set to 'android:visibility="gone"' to hide them until needed.
2. **Activity Implementation:**
   - In 'MainActivity.java', references to the 'Button', 'ProgressBar', and 'TextView' are obtained.
   - A click listener is set up for the button. When clicked, the 'startPaymentTransfer()' method is called.
   - The 'startPaymentTransfer()' method sets the 'ProgressBar' visibility to 'VISIBLE' and hides the 'TextView'.
   - A 'Handler' is used to simulate a long-running task. After 10 seconds, the 'ProgressBar' visibility is set to 'GONE', and the 'TextView' visibility is set to 'VISIBLE', displaying the message "Payment Transfer Done".

3.3 Using Intents

An Android application can contain zero or more activities. When an application has more than one activity, we often need to navigate from one activity to another activity. In Android, we navigate between activities through what is known as an **intent**. Intents are a fundamental part of Android development and it is used to communicate between various components within and across applications. They are used extensively to facilitate sharing content, starting specific activities, and responding to system events. Understanding intents and how to use them effectively is crucial for developing robust and interactive Android applications.

What is an Intent in Android?

An Intent in Android is a messaging object that can be used to request an action from another app component, such as activities, services, and broadcast receivers. It serves as a bridge to enable communication between different components within an application or between different applications on the device.

Intents can be used to:

- Start an activity.
- Start a service.
- Deliver a broadcast.

Example | **Intents in WhatsApp Mobile Application**

In the context of WhatsApp, an intent refers to a messaging object that allows different components within the app, as well as other apps, to communicate and perform specific actions. Intents are a fundamental concept in Android development, enabling seamless interaction between different parts of the app and across applications.

- **Sending Messages:**
  - When a user initiates sending a message in WhatsApp, an Intent is used to convey this action to the messaging activity.
  - The Intent specifies the action to send a message and includes the necessary data like the recipient's contact information and the message content.
  - This Intent triggers the messaging activity to handle the message composition and delivery process.
- **Opening Chats:**
  - To open a chat with a specific contact, an Intent is utilized to launch the chat activity with the relevant data.
  - The Intent contains the action to open a chat and the data identifying the target chat or contact.
  - WhatsApp processes this Intent to display the chat interface for the selected conversation to interact within the chat context.
- **Sharing Media:**
  - When users share media content like photos or videos in WhatsApp, an Intent is used to facilitate the sharing operation.
  - The Intent defines the action to send media content and includes the data representing the shared media file.
  - WhatsApp receives this Intent and handles the media sharing process by allowing users to select recipients and share the media seamlessly.
- **Invoking System Applications from WhatsApp:**
  - Intents can be used to invoke system applications like the camera or gallery to capture or select media, which can then be shared within WhatsApp.

In the context of WhatsApp, Intents serve as communication mechanisms that allow different components within the app to interact with each other and trigger specific actions. Each Intent encapsulates an operation to be performed, such as sending a message, opening a chat, or sharing media, and can target specific components like activities, services, or broadcast receivers.

3.3.1 Features and Functionalities of Intents

Intents in Android provide a versatile and powerful mechanism for inter-component communication, data exchange, and triggering actions across different parts of an application. The key features and functionalities of Intents in Android are listed below.

1. **Inter-Component Communication:**
   - Intents facilitate communication between different components of an Android application such as activities, services, and broadcast receivers.
2. **Explicit and Implicit Intents:**
   - Developers can use explicit Intents to target a specific component within the app or implicit Intents to trigger actions without specifying the target.
3. **Action Specification:**
   - Intents can specify the action to be performed such as sending a message, opening a web page, or sharing content.
4. **Data Passing:**
   - Intents allow the passing of data between components like text, URIs, or complex data structures.
5. **Intent Filters:**
   - Components can declare intent filters to specify the types of Intents they can respond to.
6. **Launching Activities:**
   - Intents are commonly used to start activities, transitioning users to different screens or views within the application.
7. **Service Invocation:**
   - Intents can trigger background services to perform tasks independently of the user interface such as downloading files or processing data.
8. **Broadcasting Events:**
   - Broadcast Intents allow components to send system-wide messages or notifications in case of inter-app communication and event handling.
9. **Explicit Component Invocation:**
   - Developers can explicitly specify the target component (activity, service) to handle the Intent, to ensure precise control over the action.
10. **Pending Intents:**
    - Pending Intents encapsulate an Intent and allow it to be executed at a later time or in a different context. It is useful for deferred actions like notifications.
11. **Intent Resolution:**
    - Android system resolves Intents by matching them with appropriate components based on their intent filters to ensure the correct handling of actions.
12. **Dynamic Behavior:**
    - Intents support dynamic behavior by enabling components to respond to various actions and data inputs based on runtime conditions.

3.3.2 Creating an Intent

In Android development, an Intent is a messaging object that facilitates communication between different components of an application, such as activities, services, and broadcast receivers. It is commonly used to start new activities and carry data between them. Let us understand how to create and use an Intent to start a new activity.

Syntax Creating an Intent Object

The basic syntax for creating an Intent to start a new activity is:

```
Intent intent = new Intent(CurrentActivity.this, TargetActivity.class);
```

This line creates a new Intent object. The first parameter is the context of the current activity (CurrentActivity.this), and the second parameter is the class of the activity to start (TargetActivity.class).

- **Intent:** This is a class in the Android framework used to define an action to be performed, such as starting another activity.
- **CurrentActivity.this:** This parameter specifies the current context, usually the activity from which the intent is being created. It provides the necessary context for the intent to start the new activity.
- **TargetActivity.class:** This parameter specifies the class of the activity that you want to navigate to. It tells the Android system which activity to launch.

3.3.3 Navigating Between Activities (or) Linking Activities Using Intents

So far, we have seen how to work with a single activity (single screen) in an Android application. However, most applications consist of multiple activities (multiple screens) to provide a richer user experience. Navigating from one screen to another is accomplished using an Intent. Intents allow us to start new activities and pass data between them seamlessly.

1. **Creating the Intent:**

   ```
   Intent intent = new Intent(CurrentActivity.this, TargetActivity.class);
   ```

   This line creates a new 'Intent' object. The first parameter is the context of the current activity ('CurrentActivity.this'), and the second parameter is the class of the activity to start ('TargetActivity.class').

2. **Adding Extra Data (Optional) :**

   ```
   intent.putExtra("KEY", "Value");
   ```

   This optional step demonstrates how to add extra data to the intent. The 'putExtra' method is used to add key-value pairs that can be passed to the target activity. For example, 'putExtra("USER_NAME", "Srikanth")' would pass the string "Srikanth" with the key "USER_NAME".

3. **Starting the Activity:**

   ```
   startActivity(intent);
   ```

   This line starts the target activity using the intent. When this method is called, the system launches the specified activity.

Example Navigating Between Activities

1. **Creating a New Project**
   (a) Open Android Studio.
   (b) Click on **"Start a new Android Studio project"**.
   (c) Choose **"Empty Views Activity"** and click **"Next"**.
   (d) Configure your project:
   - Name: 'MyApp'
   - Package name: 'com.example.myapp'
   - Save location: Choose your preferred location.
   - Language: Java
   - Minimum API level: API 21: Android 5.0 (Lollipop)

   (e) Click **"Finish"** to create the project.

2. **Adding XML Layout for MainActivity**
   1. Open 'activity_main.xml' in the 'res/layout' directory.
   2. Replace the existing code with the following:

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <Button
        android:id="@+id/button_start_activity"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start Second Activity" />
</LinearLayout>
```

3. **Adding Code to MainActivity**
   1. Open 'MainActivity.java' in the 'java/com.example.myapp' directory.
   2. Replace the existing code with the following:

Code MainActivity.java

```
package com.example.myapp;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;
```

```
import android.widget.Button;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        Button button = findViewById(R.id.button_start_activity);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Create the Intent to start SecondActivity
                Intent intent = new Intent(MainActivity.this, SecondActivity.class);

                // Add extra data to the Intent
                intent.putExtra("USER_NAME", "Srikanth");
                // Start SecondActivity
                startActivity(intent);
            }
        });
    }
}
```

4. **Creating a New Activity**
    1. Right-click on the 'java/com.example.myapp' directory.
    2. Select **New > Activity > Empty Views Activity**.
    3. Name the new activity 'SecondActivity'.
    4. Ensure the **Generate Layout File** checkbox is checked.
    5. Click **Finish** to create the new activity.
5. **Adding XML Layout for SecondActivity**
    1. Open 'activity_second.xml' in the 'res/layout' directory.
    2. Replace the existing code with the following:

Code activity_second.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <TextView
        android:id="@+id/textView_user_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="User Name: "/>
</LinearLayout>
```

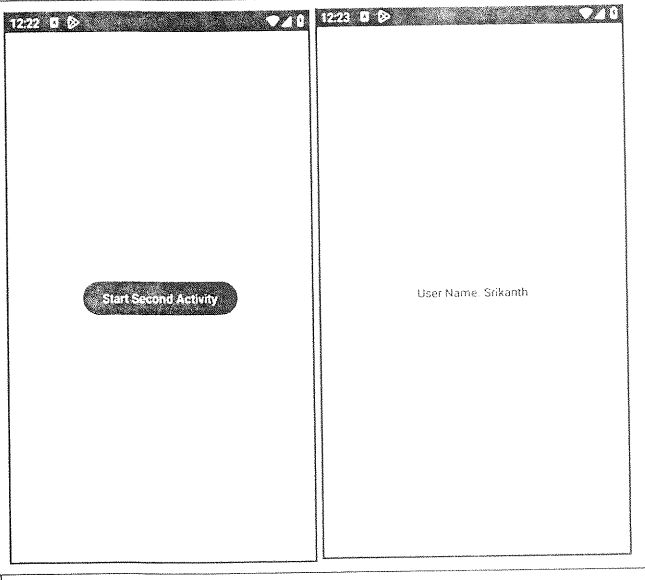
6. **Adding Code to SecondActivity**

   1. Open 'SecondActivity.java' in the 'java/com.example.myapp' directory.
   2. Replace the existing code with the following:

Code SecondActivity.java

```
package com.example.myapp;

import android.content.Intent;
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);

        // Get the Intent that started this activity
        Intent intent = getIntent();

        // Retrieve the data from the Intent
        String userName = intent.getStringExtra("USER_NAME");

        // Display the data in the TextView
        TextView textView = findViewById(R.id.textView_user_name);
        textView.setText("User Name: " + userName);
    }
}
```

Output:



Explanation

1. **MainActivity:**
   - The layout contains a button labeled "Start Second Activity".
   - In the 'MainActivity' class, an 'OnClickListener' is set on the button. When clicked, an 'Intent' is created to start 'SecondActivity'.
   - Data is added to the intent using the 'putExtra' method. In this example, the key "USER_NAME" is associated with the value "Srikanth".
   - 'startActivity(intent)' is called to navigate to 'SecondActivity'.
2. **SecondActivity:**
   - The layout contains a 'TextView' to display the user name.
   - In the 'SecondActivity' class, the 'Intent' that started the activity is retrieved using 'getIntent()'.
   - The data is extracted from the intent using 'getStringExtra' with the key "USER_NAME".
   - The extracted data is then displayed in the 'TextView'.

3.3.4 Types of Intents

In Android, Intents are a fundamental component used for communication between different parts of an application or between different applications on the device. There are two main types of Intents in Android: **Explicit Intents** and **Implicit Intents.**

1. **Explicit Intents:**

   An explicit intent is used to launch a specific app component (e.g., a particular Activity or Service). We use explicit intents when we know the target component's class name.

This type of intent is particularly useful for invoking internal functionalities or features within the app that are well-defined and known at the development stage.

Example	Explicit Intents

Scenario: Viewing a Contact's Profile in WhatsApp

- In WhatsApp, when a user taps on a contact's name to view their profile information, the app uses an explicit intent to launch the "ProfileActivity" within WhatsApp. This means that the app specifies the exact activity (ProfileActivity) that should be started to ensure that the user is taken directly to the contact's profile page within the app.

Scenario: Initiating a Payment in Google Pay

- In Google Pay, when a user selects a contact to send money to, the app uses an explicit intent to start the "PaymentActivity". This intent includes details about the selected contact, such as their name and payment information, and ensures that the "PaymentActivity" within Google Pay is the one handling the transaction.

Scenario: Viewing Account Details in Banking App

- In a banking app, when a user clicks on one of their accounts to view more details, the app uses an explicit intent to start the "AccountDetailsActivity". This intent includes the specific account ID, ensuring that the "AccountDetailsActivity" displays the correct information for the selected account.

Code	Explicit Intents

```
Intent intent = new Intent(CurrentActivity.this, ProfileActivity.class);
intent.putExtra("contact_id", contactId);
startActivity(intent);
```

Explanation

This code snippet demonstrates the use of an explicit intent to navigate from the current activity to another specific activity (ProfileActivity) within the same app.

- **Create Intent:** An intent is created to transition from the current activity to ProfileActivity.
- **Add Extra Data:** Additional data (contact ID) is added to the intent, which can be used by ProfileActivity to display specific information.
- **Start Activity:** The startActivity method is called with the intent, triggering the Android system to start ProfileActivity and passing the extra data along with it.

This approach ensures that when a user action (such as clicking on a contact's name) occurs, the app navigates to the correct activity and displays the relevant data for that specific contact.

2. **Implicit Intents:**

Implicit Intents are used to request functionality from other components on the device without specifying the exact component to be invoked. The Android system resolves the Intent to the appropriate component based on the Intent's action, data, and category.

An implicit intent does not specify the component to start. Instead, it declares a general action to perform, which allows a component from another app to handle it. The Android system matches the intent to a suitable app component by comparing the intent to the intent filters declared in the manifest file of other apps.

Example	Implicit Intents

Scenario: Sharing a Location in WhatsApp

- In WhatsApp, when a user wants to share their current location in a chat, the app uses an implicit intent to open the system's map application. This intent includes the location data but does not specify which map app should handle it. The Android system then displays a list of all installed map apps that can handle the location sharing, allowing the user to choose their preferred map app.

Scenario: Scanning a QR Code

- When a user chooses to scan a QR code for payment in Google Pay, the app uses an implicit intent to open any available QR code scanner app on the device. This intent specifies the action of scanning a QR code but does not specify a particular QR code scanner app. The Android system presents a list of all apps that can perform QR code scanning, letting the user pick their preferred app.

Scenario: Making a Call to Customer Support

- In a banking app, when a user clicks on a button to call customer support, the app uses an implicit intent to open the phone dialer. This intent includes the customer support phone number but does not specify which phone app should handle the call. The Android system presents the user with the phone dialer, allowing them to make the call using their preferred phone app.

Code	Impicit Intents

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("geo:37.7749,-122.4194"));
if (intent.resolveActivity(getPackageManager()) != null) {
    startActivity(intent);
}
```

Explanation

This code snippet demonstrates the use of an implicit intent to view a geographic location on a map. Here's the step-by-step process:

1. **Create Intent:** An implicit intent is created with the action to view content.
2. **Set Data:** The data for the intent is set to a geographic location (latitude 37.7749, longitude -122.4194).
3. **Check Availability:** The code checks if there is an application installed on the device that can handle the intent to view the geographic location.
4. **Start Activity:** If such an application exists, it starts the activity to view the location on a map.

This approach allows users to view a specific location on a map by simply clicking on a link or button that triggers this intent. The Android system handles finding an appropriate map application that can display the location, making the process seamless for the user.

3.3.5 Intent Class Methods

The 'Intent' class in Android is a fundamental component used to perform various operations, such as starting a new activity, sending broadcast messages, and communicating between components. Intents

facilitate the transfer of data between activities and other application components. Understanding the methods of the 'Intent' class is crucial for efficient application development. Below, we explore some key methods of the 'Intent' class, their descriptions, and examples.

Method	Description	Example
Intent()	Constructor to create a new empty Intent.	Intent intent = new Intent();
Intent (Context package Context, Class<?> cls)	Constructor to create an Intent specifying the context and the target activity class.	Intent intent = new Intent(MainActivity.this, SecondActivity.class);
putExtra(String name, String value)	Adds extended data to the intent.	intent.putExtra("USER_NAME", "Srikanth");
getStringExtra(String name)	Retrieves extended data from the intent.	String userName = intent.getStringExtra("USER_NAME");
setAction(String action)	Sets a specified action for the Intent.	intent.setAction(Intent.ACTION_VIEW);
getAction()	Retrieves the action associated with the Intent.	String action = intent.getAction();
setData(Uri data)	Sets the data URI for the Intent.	intent.setData(Uri.parse("https://www.google.com"));
getData()	Retrieves the data URI associated with the Intent.	Uri data = intent.getData();
setType(String type)	Sets an explicit MIME data type for the Intent.	intent.setType("image/*");
getType()	Retrieves the MIME type set for the Intent.	String type = intent.getType();
setFlags(int flags)	Sets special flags controlling how the Intent is handled.	intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
getFlags()	Retrieves the special flags associated with the Intent.	int flags = intent.getFlags();
resolveActivity (PackageManager pm)	Finds the first activity capable of handling the Intent.	if (intent.resolveActivity(getPackageManager()) != null) { startActivity(intent); }
addCategory(String category)	Adds a category to the Intent.	intent.addCategory(Intent.CATEGORY_DEFAULT);
getCategories()	Retrieves the categories associated with the Intent.	Set<String> categories = intent.getCategories();
removeExtra(String name)	Removes extended data from the Intent.	intent.removeExtra("USER_NAME");
toUri(int flags)	Converts the Intent to a URI string.	String uriString = intent.toUri(Intent.URI_INTENT_SCHEME);
toString()	Returns a string representation of the Intent.	String intentString = intent.toString();

3.3.6 Intent Class Attributes

The 'Intent' class in Android has several constants that are used to specify various attributes, such as categories, actions, flags, and extra keys. These attributes help in defining the behavior and purpose of the intent. Below is a table that outlines some of the key attributes of the 'Intent' class, their descriptions, and examples.

Attribute	Description	Example
Intent.ACTION_VIEW	Used to display data to the user.	intent.setAction(Intent.ACTION_VIEW);
Intent.ACTION_SEND	Used to send data from one activity to another.	intent.setAction(Intent.ACTION_SEND);
Intent.ACTION_MAIN	Indicates the main entry point of an application.	intent.setAction(Intent.ACTION_MAIN);
Intent.CATEGORY_DEFAULT	Indicates that an activity can be the default action to perform for a given intent.	intent.addCategory(Intent.CATEGORY_DEFAULT);
Intent.CATEGORY_BROWSABLE	Allows the activity to be started by a web browser to display data referenced by a link.	intent.addCategory(Intent.CATEGORY_BROWSABLE);
Intent.FLAG_ACTIVITY_NEW_TASK	If set, this activity will become the start of a new task on this history stack.	intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
Intent.FLAG_ACTIVITY_CLEAR_TOP	If set, and the activity being launched is already running in the current task, then instead of launching a new instance, all other activities on top of it will be closed.	intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);'
Intent.FLAG_GRANT_READ_URI_PERMISSION	Grants temporary read access to the content URI specified by the Intent's data.	intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
Intent.EXTRA_TEXT	Used to supply text data as an extra field in the intent.	intent.putExtra(Intent.EXTRA_TEXT, "Hello!");
Intent.EXTRA_EMAIL	A string array of email addresses to send an email to.	intent.putExtra(Intent.EXTRA_EMAIL, new String[]{"example@example.com"});

3.3.7 Intent Filters

Intent filters are components of an Android application manifest file that specify the types of intents an activity, service, or broadcast receiver can respond to. They define the capabilities of a component

and allow other components to interact with it based on the specified criteria. Intent filters consist of actions, categories, data, and MIME types that help Android system identify the appropriate component to handle a specific intent.

Components of an Intent Filter

Intent filters are composed of three main elements: action, category, and data. Each of these elements plays a crucial role in defining the scope and capability of the intent filter.

1. **Action** : The action element represents the general action to be performed by the intent. Actions are typically predefined constants in the 'Intent' class. Examples of actions include:
   - **'android.intent.action.VIEW':** This action indicates that an activity can display data to the user. It's commonly used for viewing web pages, images, and other types of content.
   - **'android.intent.action.SEND':** This action is used to send data from one activity to another. It's often used for sharing content such as text, images, or other data.
   - **'android.intent.action.DIAL':** This action is used to dial a number, opening the dialer with the number populated.
   - **'android.intent.action.MAIN':** This action is used to start the main activity of an application. It signifies the main entry point of the app and is often paired with the 'LAUNCHER' category to display the app icon in the launcher.
   - **'android.intent.action.EDIT':** This action indicates that the activity can edit specified data. It's typically used for editing contacts or calendar events.
   - **'android.intent.action.DELETE':** This action is used to delete specified data, such as files or database entries.
2. **Category** : The category element provides additional information about the action being performed. Categories help to refine the type of component that should handle the intent. Some common categories include:
   - **'android.intent.category.DEFAULT':** This is the most commonly used category and indicates that the activity can be launched by default if no other category is specified.
   - **'android.intent.category.BROWSABLE':** This category allows the activity to be started by a web browser to display data referenced by a link. It enables activities to handle URIs.
   - **'android.intent.category.LAUNCHER':** This category is used to indicate that the activity should be listed in the system's application launcher. It designates the main entry point for the application.
   - **'android.intent.category.HOME':** This category is used for the home screen of the device.
   - **'android.intent.category.APP_CONTACTS':** This category is used for activities that deal with contacts.
   - **'android.intent.category.APP_CALENDAR':** This category is used for activities related to calendar events.

   Categories are used in combination with actions to provide a more specific context for the intent to ensure that the most appropriate component is chosen to handle it.

3. **Data :** The data element specifies the type of data that the intent is interested in. This can include a URI scheme, a MIME type, or both. The data element allows components to declare their capability to handle specific types of data, such as:
   - **URI Scheme:** Specifies the protocol for accessing the data (e.g., 'http', 'https', 'mailto'). This indicates the type of data source the component can handle.
   - **Example:** 'android:scheme="http"' indicates that the component can handle HTTP URLs.
   - **MIME Type:** Specifies the format of the data the component can handle (e.g., 'image/*', 'text/plain'). This allows components to specify the exact type of data they can process.
   - **Example:** 'android:mimeType="image/*"' indicates that the component can handle any type of image file.

Example | **Scenario: Launching WhatsApp from a URL**

Imagine a scenario where a user clicks on a WhatsApp share link from a browser, and the link is designed to open the WhatsApp app directly. This involves the use of specific intent filters within the WhatsApp app to handle such links.

1. **User Action: Clicking a WhatsApp Share Link**

   When the user clicks on a link in a browser that is intended to share content via WhatsApp, an intent is created with the following details:
   - **Action:** 'android.intent.action.VIEW' - Indicates that the intent is viewing data.
   - **Data:** URI with a scheme that WhatsApp can handle, such as 'https://wa.me/'.
   - **Category:** 'android.intent.category.DEFAULT', 'android.intent.category.BROWSABLE' - Indicates that the activity can be launched from a browser and is the default action for this data.

2. **Intent Filter in WhatsApp**

   To handle such intents, WhatsApp would have an intent filter in its AndroidManifest.xml file that looks something like this:

Code | **Impicit Intents**

```
<activity android:name=".MainActivity">
    <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="wa.me" />
    </intent-filter>
</activity>
```

Example | **Scenario: Launching Main Activity**

For an application to launch its main activity when the app icon is tapped, it uses specific intent filters to designate the main entry point. This is done by defining the following:

1. **Action:** 'android.intent.action.MAIN' - Indicates that this activity is the main entry point for the application.

2. **Category:** 'android.intent.category.LAUNCHER' - Indicates that the activity should be listed in the system's application launcher.

AndroidManifest.xml file looks like this:

Code Impicit Intents

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

3.3.8 Returning Results from an Intent

In Android, returning results from an intent allows one activity to receive data from another activity it started. This mechanism is useful when one activity needs to get feedback or data from another activity. For example, an application may start an activity to allow the user to pick a contact, and the contact's information is then returned to the original activity.

Example Scenario

A simple example involves an application where an activity lets the user input their name in another activity. The entered name is then returned and displayed in the original activity.

Example MainActivity Layout ('activity_main.xml')

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center">

    <TextView
        android:id="@+id/tvResult"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Your Name Will Appear Here"
        android:textSize="18sp"
        android:padding="16dp"/>

    <Button
        android:id="@+id/btnEnterName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enter Name" />
</LinearLayout>
```

Explanation:

- A 'LinearLayout' with vertical orientation and center gravity is used to center the content on the screen.
- A 'TextView' displays the result returned from the second activity. Initially, it shows a placeholder text.
- A 'Button' labeled "Enter Name" will trigger the transition to the second activity when clicked.

Example MainActivity Class ('MainActivity.java')

```
package com.example.returnresult;

import android.content.Intent;
import android.os.Bundle;
import android.widget.Button;
import android.widget.TextView;

import androidx.annotation.Nullable;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private static final int REQUEST_CODE = 1;
    private TextView tvResult;

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

        tvResult = findViewById(R.id.tvResult);
        Button btnEnterName = findViewById(R.id.btnEnterName);

        btnEnterName.setOnClickListener(view -> {
            Intent intent = new Intent(MainActivity.this, NameInputActivity.class);
            startActivityForResult(intent, REQUEST_CODE);
        });
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if (requestCode == REQUEST_CODE && resultCode == RESULT_OK && data != null) {
            String name = data.getStringExtra("name");
            tvResult.setText(name);
        }
    }
}
```

Explanation:

- **'MainActivity' Class:** Inherits from 'AppCompatActivity' and acts as the main screen.
- **'onCreate' Method:** Initializes the UI components.
- Finds 'TextView' and 'Button' by their IDs.
- Sets an 'OnClickListener' on the button to start 'NameInputActivity' for a result using 'startActivityForResult'.
- **'onActivityResult' Method:** Handles the result returned from 'NameInputActivity'.
- Checks if the request code and result code match the expected values.
- Retrieves the name from the returned intent and updates the 'TextView' with this name.

Example NameInputActivity Layout ('activity_name_input.xml')

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center">

    <EditText
        android:id="@+id/etName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter your name"
        android:padding="16dp" />

    <Button
        android:id="@+id/btnSubmit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Submit" />
</LinearLayout>
```

Explanation

- A 'LinearLayout' with vertical orientation and center gravity centers the content on the screen.
- An 'EditText' allows the user to input their name.
- A 'Button' labeled "Submit" will send the entered name back to 'MainActivity' when clicked.

Example NameInputActivity Class ('NameInputActivity.java')

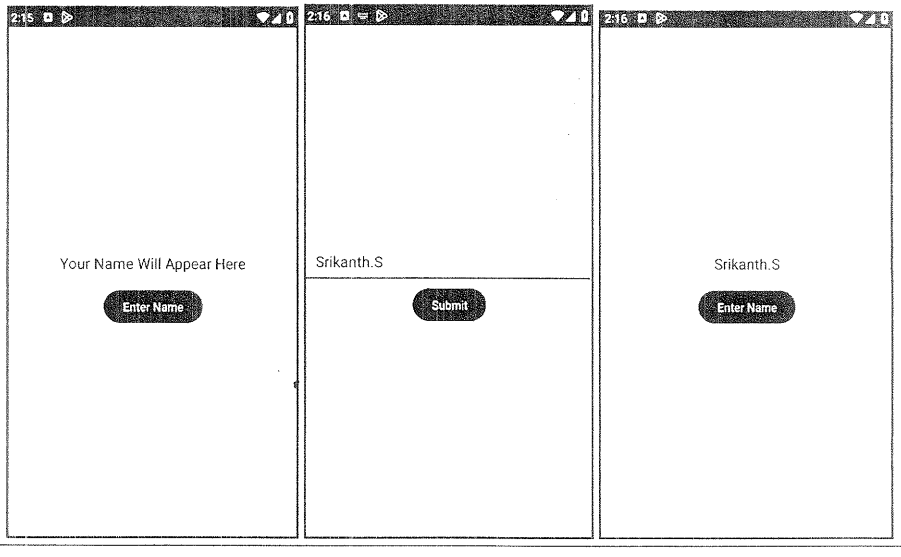
```
package com.example.returnresult;

import android.content.Intent;
import android.os.Bundle;
import android.widget.Button;
import android.widget.EditText;

import androidx.appcompat.app.AppCompatActivity;
```

```
public class NameInputActivity extends AppCompatActivity {

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

        EditText etName = findViewById(R.id.etName);
        Button btnSubmit = findViewById(R.id.btnSubmit);

        btnSubmit.setOnClickListener(view -> {
            String name = etName.getText().toString();
            Intent resultIntent = new Intent();
            resultIntent.putExtra("name", name);
            setResult(RESULT_OK, resultIntent);
            finish();
        });
    }
}
```

Explanation

- **'NameInputActivity' Class:** Inherits from 'AppCompatActivity' and acts as the screen where the user enters their name.
- **'onCreate' Method:** Initializes the UI components.
- Finds 'EditText' and 'Button' by their IDs.
- Sets an 'OnClickListener' on the button to capture the entered name, create a new intent, add the name to the intent, set the result to 'RESULT_OK', and finish the activity.

Output



3.3.9 Passing Data with Intents

Passing data with intents in Android refers to the process of transferring information between different components of an application such as activities or services, using Intent objects. This enables components to communicate and share data. Data can be passed through key-value pairs added to the intent using methods such as putExtra(). To pass data between activities, we should use the 'Extras' bundle in an intent. Extras are key-value pairs where the key is a string, and the value can be various data types like strings, integers, booleans, etc. This bundle allows to transfer complex data structures between activities efficiently.

Steps to Pass Data Using an Intent

1. **Creating the Intent:** An intent object is created in the source activity to specify the target activity and include any data to be passed.
2. **Adding Extras:** Data is added to the intent using methods like 'putExtra'.
3. **Starting the Activity:** The target activity is started using 'startActivity' or 'startActivityForResult' if a result is expected.
4. **Retrieving Data:** In the target activity, the data is retrieved from the intent using methods like 'getStringExtra' or 'getIntExtra'.

Example	Scenario

Imagine an application where a user enters their name on one screen, and it is displayed on another screen.

1. **Source Activity:** The activity where the user enters their name.
   - Create an intent to start the target activity.
   - Use 'putExtra' to add the entered name to the intent.
2. **Target Activity:** The activity where the entered name is displayed.
   - Retrieve the name using 'getStringExtra' from the intent.
   - Display the retrieved name in a 'TextView'.

Advantages of Using Intents for Data Passing

- **Decoupling Components:** Intents promote loose coupling between components, allowing for more modular and flexible application design.
- **Inter-App Communication:** Intents facilitate communication not only within an app but also between different apps.
- **Simplicity:** The key-value pair mechanism of extras makes it simple to pass various data types without complex serialization or deserialization processes.

3.3.10 Using the Intent Object to Invoke Built-in Application

Using the Intent object to invoke built-in applications refers to leveraging Android's Intents to request functionality from other applications that are pre-installed on the device. This allows to utilize existing apps for common tasks to enhance the user experience by providing seamless integration with system functionalities. This functionality can include opening the web browser, dialing a number, launching

the calendar or launching contacts apps. This integration enhances the user experience by utilizing the familiar system applications.

How to invoke Built-in Application Using the Intent Object?

To invoke built-in applications, an Intent object is created with a specified action and, if necessary, data. The action defines what kind of task is to be performed, and the data specifies the exact task or content to be handled. When the Intent is passed to the startActivity() method, the Android system identifies the appropriate application to handle the action.

Example | **Launching the Contacts App from a Button Click**

This example demonstrates a simple Android application with one screen containing a button. Clicking the button launches the device's contacts application.

Step 1: Create a New Android Project

(i) Open Android Studio and create a new project.

(ii) Choose "Empty Views Activity" and name the project (e.g., "ContactsLauncher").

Step 2: Update the Layout XML File

Edit the 'activity_main.xml' file to include a button in the center of the screen.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/button_open_contacts"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Open Contacts"
        android:layout_centerInParent="true" />
</RelativeLayout>
```

Step 3: Update the MainActivity Java File

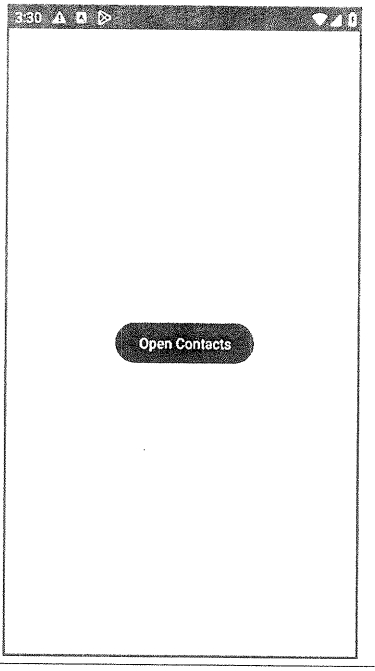
Edit the 'MainActivity.java' file to handle the button click and launch the contacts application.

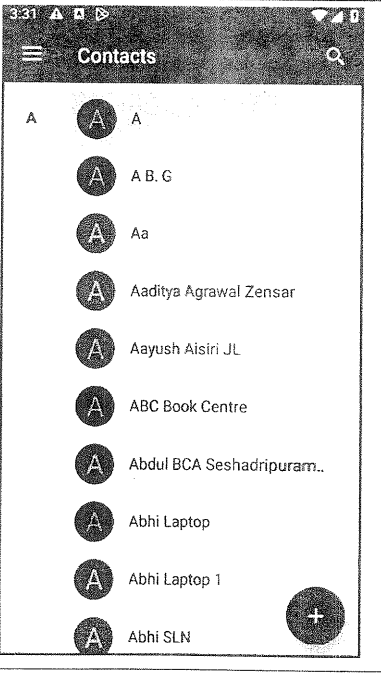
Code MainActivity.java

```
package com.example.contactslauncher;

import android.content.Intent;
import android.os.Bundle;
import android.provider.ContactsContract;
import android.view.View;
```

```
import android.widget.Button;
import androidx.appcompat.app.AppCompatActivity;
public class MainActivity extends AppCompatActivity {

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

        Button buttonOpenContacts = findViewById(R.id.button_open_contacts);
        buttonOpenContacts.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(Intent.ACTION_VIEW);
                intent.setData(ContactsContract.Contacts.CONTENT_URI);
                startActivity(intent);
            }
        });
    }
}
```

Output





Explanation:

1. **Layout XML File ('activity_main.xml'):**
   - Defines a 'RelativeLayout' with a single 'Button' centered in the parent layout.
2. **Main Activity Java File ('MainActivity.java'):**
   - **onCreate() Method:** Initializes the activity and sets the content view to 'activity_main.xml'.
   - **Button Click Listener:** Sets up an 'OnClickListener' for the button to handle click events.
   - **Intent Creation:** Creates an 'Intent' with the action 'Intent.ACTION_VIEW' and sets the data to the contacts content URI.
   - **Start Activity:** Starts the contacts application with the created intent using 'startActivity(intent)'.
3. **Run the Application**
   - Run the application on an Android device or emulator.
   - Click the "Open Contacts" button to launch the contacts app.

This example demonstrates how to use an Intent to invoke the built-in contacts application from an Android activity.

3.4 Fragments

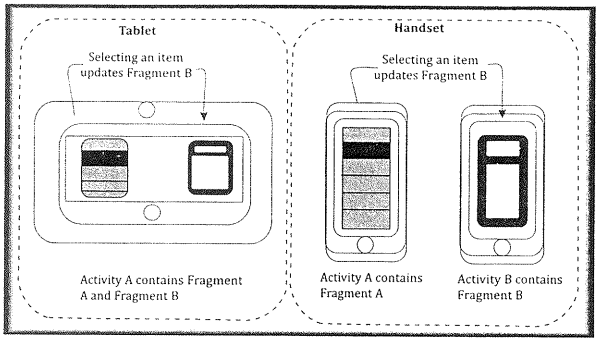
In Android development, an activity represents a single screen within an application. On smaller devices such as smartphones, an activity typically occupies the entire screen by displaying the various views that comprise the user interface of the application. However, on larger devices like tablets, displaying just one activity on the entire screen can seem inefficient and lead to an underutilized interface. To address this, Android employs fragments, which are essentially modular sections of an activity that can be combined to create a flexible and dynamic user interface.

Fragments are like mini-activities that can be embedded within an activity. They contain their own layout, lifecycle, and behavior. This modular approach allows developers to efficiently use the larger screen space by displaying multiple fragments simultaneously, which can be rearranged or replaced as needed. We can combine multiple fragments in a single activity to build a multi-pane UI. Fragments are useful for creating flexible layouts that can adapt to different screen sizes and orientations. They also make it easier to reuse and maintain code. Fragments can be dynamically added or removed from an activity for creating layouts that can adapt to different screen sizes and orientations.

What are Fragments in Android?

Fragments in Android are modular and reusable UI components that represent a portion of a user interface or behavior within an activity. In simple terms, Fragments are like mini-activities that can be embedded within an activity

They were introduced to address the challenges of creating flexible and responsive user interfaces for different screen sizes and orientations. Fragments are used to build dynamic and multi-pane layouts that can adapt to various device configurations. It is especially useful for applications that run on a variety of screen sizes such as smartphones and tablets.

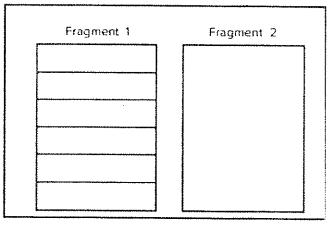


How Do Fragments Work?

Fragments are modular UI components that can be added to or removed from an activity while the application is running. This enables developers to design more flexible and responsive user interfaces. Each fragment has its own lifecycle and can manage its own user interface. It is easier to handle different parts of an activity independently using fragments.

Let us consider below two fragments.

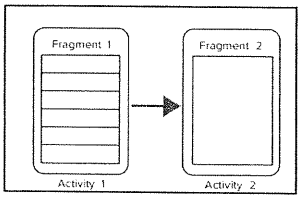
- **Fragment 1:** This fragment display a list of book titles.
- **Fragment 2:** This fragment show the details of a selected book.



The below Figure demonstrates two fragments that can be displayed side by side. Depending on the screen size and orientation, these fragments might either be part of the same activity or spread across different activities.

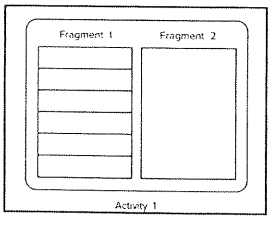
The below Figure illustrates a scenario where there are two activities:

- **Activity 1:** Contains Fragment 1, which shows a list of items.
- **Activity 2:** Contains Fragment 2, which displays details of the selected item.



When an item in Fragment 1 is selected, Activity 2 starts to show the details in Fragment 2.

The below Figure shows how both fragments can be displayed together within one activity when the application is running on a larger screen in landscape mode. This layout allows both the list and the details to be visible simultaneously to enhance the user experience on tablets.



Example	Fragments in PhonePe Mobile App

Scenario: Viewing Transaction History and Details

PhonePe can use fragments to display a list of transactions and detailed information about each transaction.

- **Transaction History Fragment:** Displays a list of recent transactions made by the user, including payments, money transfers, and bill payments. When a user selects a transaction, an Intent is used to pass the transaction details to the Transaction Detail Fragment.
- **Transaction Detail Fragment:** Displays detailed information about the selected transaction, such as transaction ID, amount, date, and status. Provides options to download the receipt, repeat the transaction, or raise a support query.
- **Tablet Layout:** Both Transaction History Fragment and Transaction Detail Fragment are displayed side by side, allowing users to browse transactions and view details simultaneously.
- **Smartphone Layout:** Initially, only the Transaction History Fragment is displayed. When a user selects a transaction, the Transaction Detail Fragment replaces the Transaction History Fragment to display the transaction details

3.4.1 Features and Benefits of Fragments

Fragments offer several key features that enhance the flexibility and functionality of user interfaces. The primary features of fragments are listed below.

1. **Modularity :** Fragments allow the UI to be broken down into smaller, manageable pieces. Each fragment represents a portion of the user interface, which simplifies the management and updating of specific parts of the app without affecting the entire UI.
2. **Reusability :** Fragments promote code reuse. Once a fragment is created, it can be used across multiple activities and different parts of the application. This reduces the need for duplicating code and ensures a consistent user experience.
3. **Lifecycle Management :** Fragments have their own lifecycle, similar to activities. This includes lifecycle methods such as 'onCreate()', 'onCreateView()', 'onActivityCreated()', 'onStart()', 'onResume()', 'onPause()', 'onStop()', 'onDestroyView()', and 'onDestroy()'. This allows to handle fragment-specific events and manage their state independently of the host activity.
4. **Dynamic User Interfaces :** Fragments enable the creation of dynamic and flexible UIs. They can be added, replaced, or removed at runtime based on user interactions or application logic.
5. **Communication :** Fragments can communicate with their host activity and other fragments. This communication is facilitated through interfaces, shared ViewModels, or other communication patterns to ensure that data and events are properly coordinated between fragments and activities.
6. **Support for Multiple Layouts :** Fragments are particularly useful for creating applications that need to run on various screen sizes, such as phones and tablets. It is possible to define different layouts for fragments in various resource folders (e.g., 'layout', 'layout-land', 'layout-sw600dp') to ensure the UI adapts seamlessly to phones, tablets, and different orientations.
7. **Back Stack Management :** Fragments can be added to the back stack. When a fragment transaction is added to the back stack, the user can navigate back to the previous state using

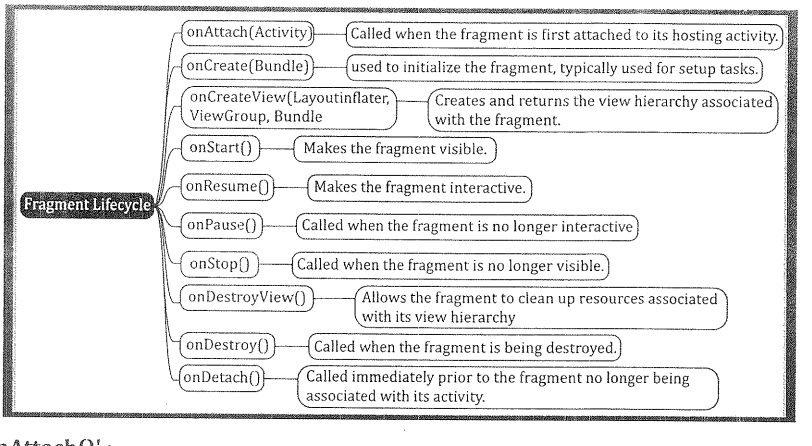
the back button, similar to how activities are managed. This enhances the user's navigation experience within the app.

8. **Nested Fragments :** Fragments can host other fragments within them. This allows for complex UIs where a fragment can contain child fragments to provide a deeper level of UI modularity and organization. Nested fragments are useful for creating sophisticated layouts and managing nested content.
9. **Improved Performance :** Using fragments can improve the performance of an application. Since fragments are lightweight compared to activities, they can reduce the overhead associated with managing multiple activities, leading to faster and more efficient applications.
10. **Seamless Transitions :** Fragments support seamless transitions and animations between different UI states. This enhances the visual appeal of the application and provides a smoother user experience when navigating between different sections of the app.

3.4.2 Fragment Lifecycle

Fragments in Android have a complex lifecycle that is closely related to the lifecycle of the hosting activity. Each lifecycle method provides a specific opportunity for developers to implement logic related to the fragment's state and interaction with the user.

Each method in the fragment lifecycle has a specific role in managing the fragment's state and interactions with the user and its host activity. Properly handling these lifecycle methods ensures that fragments are created, displayed, and destroyed in a way that maintains a smooth user experience and efficient resource management. A fragment lifecycle methods are explained below.



1. **'onAttach()' :**
   - **Purpose :** This method is called when the fragment is first attached to its host activity. It's a good place to perform initial setup that requires a reference to the activity.
   - **Typical Uses:** Initializing variables that need the context of the activity, or setting up callbacks to the activity.

Code

```
@Override
public void onAttach(Context context) {
    super.onAttach(context);
    // Example: Initialize variables or listeners that require the activity context
    Log.d("Fragment Lifecycle", "onAttach called");
}
```

2. **'onCreate()'**
   - **Purpose:** This method is called to do the initial creation of the fragment. It's used for non-graphical initializations.
   - **Typical Uses:** Setting up fragment-specific data, starting background tasks, or initializing components that don't need the fragment's view hierarchy.

Code

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Example: Initialize data or perform non-graphical setup
    Log.d("Fragment Lifecycle", "onCreate called");
}
```

3. **'onCreateView()'**
   - **Purpose:** Called to create the view hierarchy associated with the fragment. Here, the layout for the fragment is inflated.
   - **Typical Uses:** Inflating the fragment's XML layout and returning the root view of the layout.

Code

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
    // Example: Inflate the fragment's layout
    View view = inflater.inflate(R.layout.fragment_layout, container, false);
    Log.d("Fragment Lifecycle", "onCreateView called");
    return view;
}
```

4. **'onStart()'**
   - **Purpose:** Called when the fragment becomes visible to the user.
   - **Typical Uses:** Starting animations, initializing components that need to be visible, or updating the UI.

Code

```
@Override
public void onStart() {
    super.onStart();
    // Example: Start animations or refresh UI elements
    Log.d("Fragment Lifecycle", "onStart called");
}
```

5. **'onResume()'**
   - **Purpose:** Called when the fragment is visible to the user and actively running.
   - **Typical Uses:** Resuming any paused operations or interactions, starting audio/video playback, or refreshing dynamic content.

Code

```
@Override
public void onResume() {
    super.onResume();
    // Example: Resume operations or update UI with new data
    Log.d("Fragment Lifecycle", "onResume called");
}
```

6. **'onPause()'**
   - **Purpose:** Called when the fragment is no longer in the foreground.
   - **Typical Uses:** Pausing ongoing operations, such as stopping animations or video playback, or saving UI state.

Code

```
@Override
public void onPause() {
    super.onPause();
    // Example: Pause operations or save UI state
    Log.d("Fragment Lifecycle", "onPause called");
}
```

7. **'onStop()'**
   - **Purpose**: Called when the fragment is no longer visible to the user.
   - **Typical Uses:** Stopping operations that don't need to run when the fragment is not visible, like halting network requests or sensor listeners.

Code

```
@Override
public void onStop() {
    super.onStop();
    // Example: Stop operations that are not needed when fragment is not visible
    Log.d("Fragment Lifecycle", "onStop called");
}
```

8. 'onDestroyView()'
   - **Purpose:** Called when the view hierarchy associated with the fragment is being removed.
   - **Typical Uses:** Cleaning up resources related to the view, like unbinding views from their resources or removing listeners.

```
@Override
public void onDestroyView() {
    super.onDestroyView();
    // Example: Clean up resources related to the view
    Log.d("Fragment Lifecycle", "onDestroyView called");
}
```

9. 'onDestroy()'
   - **Purpose:** Called when the fragment is being destroyed.
   - **Typical Uses:** Cleaning up resources that aren't related to the view, such as stopping background tasks or releasing data bindings.

```
@Override
public void onDestroy() {
    super.onDestroy();
    // Example: Clean up other resources
    Log.d("Fragment Lifecycle", "onDestroy called");
}
```

10. 'onDetach()'
    - **Purpose:** Called when the fragment is detached from its host activity.
    - **Typical Uses:** Final cleanup actions, like nullifying references to the activity to prevent memory leaks.

```
@Override
public void onDetach() {
    super.onDetach();
    // Example: Final cleanup
    Log.d("Fragment Lifecycle", "onDetach called");
}
```

3.4.3 Implementing Fragments in Android Apps

1. **Defining a Fragment:**

   Defining a fragment in an Android app involves creating both an XML layout file and a corresponding Java class. The XML layout file defines the UI components of the fragment, while the Java class controls the fragment's behavior. This separation allows for modular design and reuse of UI components across different parts of the application.

Fragments can be added to activities in two ways:

- **Statically in XML Layouts:** The fragment is defined within the XML layout file of the activity. This approach is simple and convenient for fragments that will not change during the activity's lifecycle.
- **Dynamically in Java Code:** The fragment is added to the activity programmatically using the 'FragmentManager'. This method is more flexible and allows for adding, removing, or replacing fragments at runtime based on user interactions or other conditions.

2. **Embedding Fragments in Activities:**

   Fragments can be embedded in activities to create a composite UI. When embedding fragments, developers must ensure that the activity is compatible by extending from 'FragmentActivity' or 'AppCompatActivity'. These base classes provide the necessary support for managing fragments.

   - **Statically using XML Layout Files:** In this approach, the fragment is included directly in the activity's XML layout file using the '<fragment>' tag. This method is straightforward and useful for static UIs where the fragment will remain unchanged throughout the activity's lifecycle.
   - **Dynamically using Java Code:** This approach involves using the 'FragmentManager' and 'FragmentTransaction' classes to add, remove, or replace fragments programmatically. This method is preferred for dynamic UIs where fragments need to be swapped or updated based on user interactions or application logic.

3. **Fragment Lifecycle Management:**

   Managing the lifecycle of a fragment is crucial for maintaining UI state and ensuring a smooth user experience. Fragments have their own lifecycle, which is closely tied to the lifecycle of the host activity. Developers need to override lifecycle methods to perform initialization, setup the view, and handle cleanup tasks.

   Key lifecycle methods include:

   - **'onCreate':** Called when the fragment is first created. Used for initializing non-UI components.
   - **'onCreateView':** Called to create the view hierarchy associated with the fragment. Inflate the fragment's layout in this method.
   - **'onActivityCreated':** Called when the activity's 'onCreate' method has returned. Used to do final initialization once the activity's UI is fully initialized.
   - **'onStart':** Called when the fragment becomes visible to the user.
   - **'onResume':** Called when the fragment is ready to interact with the user.
   - **'onPause':** Called when the fragment is no longer interacting with the user but is still visible.
   - **'onStop':** Called when the fragment is no longer visible to the user.
   - **'onDestroyView':** Called when the view hierarchy associated with the fragment is being removed.

- **'onDestroy'**: Called when the fragment is being destroyed.
- **'onDetach'**: Called when the fragment is detached from the activity.

Properly managing these lifecycle methods ensures that resources are handled efficiently, and the user experience remains smooth even as the fragment's state changes.

3.4.4 Creating and Using Fragments

Fragments in Android provide a way to build dynamic and flexible user interfaces, which are essential for modern applications that need to adapt to various screen sizes and orientations. Let us create a simple example to demonstrate the creation and use of fragments within a main activity. This example includes two fragments: one containing a button and the other displaying text that changes when the button is clicked.

Example	Step by Step Implementation of Creating and Using Fragments

Step 1: Create a New Project:

1. Open Android Studio.
2. Start a New Android Studio Project.
3. Name Your Project: 'FragmentExampleApp'.
4. Select "Empty Views Activity" Template.
5. Set Language: Java.
6. Finish Project Setup

Step 2: Set Up the Main Activity Layout:

1. Edit 'activity_main.xml' to include two 'FrameLayout' containers for the fragments.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <FrameLayout
        android:id="@+id/fragment_container1"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />

    <FrameLayout
        android:id="@+id/fragment_container2"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />
</LinearLayout>
```

Explanation

- **LinearLayout:** The root layout of the activity. It arranges its child views in a vertical column.
- **FrameLayout (fragment_container1):** A placeholder for the first fragment. The 'layout_weight="1"' ensures it takes up half of the available vertical space.
- **FrameLayout (fragment_container2):** A placeholder for the second fragment. The 'layout_weight="1"' ensures it takes up the other half of the available vertical space.

Step 3: Create the First Fragment:

1. **Create a New Fragment Class:**
   - Right-click on 'com.example.fragmentexampleapp'.
   - Select 'New -> Java Class'.
   - Name the class 'FirstFragment'.

Code FirstFragment.java

```
package com.example.fragmentexampleapp;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import androidx.fragment.app.Fragment;

public class FirstFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
      View view = inflater.inflate(R.layout.fragment_first, container, false);

        // Set up button click listener
        Button button = view.findViewById(R.id.button_first);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
            ((MainActivity) getActivity()).updateSecondFragment("Button Clicked!");
            }
        });
        return view;
    }
}
```

Explanation

- **FirstFragment Class:** Extends the 'Fragment' class.
- **onCreateView Method:** Inflates the fragment's layout when the fragment is created.
- **Button Click Listener:** Sets up a listener on the button to call a method in the main activity.

2. **Create the Fragment Layout:**
   - Right-click on 'res/layout'.
   - Select 'New -> Layout Resource File'.
   - Name the file 'fragment_first.xml'.

Code fragment_first.xml

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

    <Button
        android:id="@+id/button_first"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me" />
</LinearLayout>
```

Explanation

- **LinearLayout:** The root layout of the fragment. It arranges its child views vertically and centers them within the layout.
- **Button:** Displays a button with the text "Click Me".

Step 4: Create the Second Fragment:

1. **Create a New Fragment Class:**
   - Right-click on 'com.example.fragmentexampleapp'.
   - Select 'New -> Java Class'.
   - Name the class 'SecondFragment'.

Code SecondFragment.java

```
package com.example.fragmentexampleapp;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;
import androidx.fragment.app.Fragment;
```

```
public class SecondFragment extends Fragment {
    private TextView textView;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        View view = inflater.inflate(R.layout.fragment_second, container, false);
        textView = view.findViewById(R.id.text_view_second);
        return view;
    }
    public void updateText(String text) {
        if (textView != null) {
            textView.setText(text);
        }
    }
}
```

Explanation

- **SecondFragment Class:** Extends the 'Fragment' class.
- **onCreateView Method:** Inflates the fragment's layout when the fragment is created.
- **updateText Method:** Updates the text in the TextView.

2. **Create the Fragment Layout:**
   - Right-click on 'res/layout'.
   - Select 'New -> Layout Resource File'.
   - Name the file 'fragment_second.xml'.

Code **fragment_second.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<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/text_view_second"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is the Second Fragment" />
</LinearLayout>
```

Explanation

- **LinearLayout:** The root layout of the fragment. It arranges its child views vertically and centers them within the layout.
- **TextView:** Displays the text "This is the Second Fragment".

Step 5: Load Fragments in Main Activity:

Update 'MainActivity.java' to load both fragments.

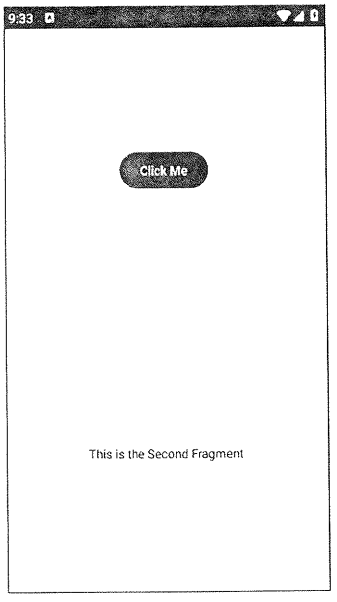
Code MainActivity.java

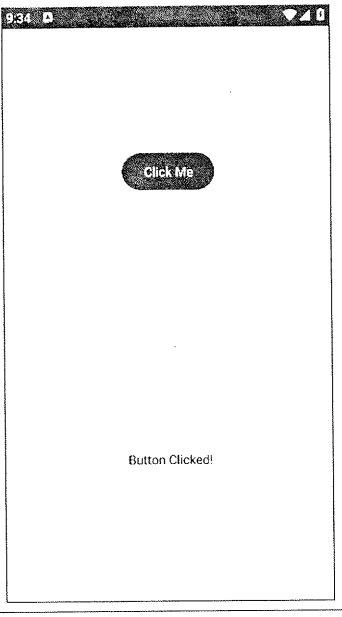
```
package com.example.fragmentexampleapp;
 import android.os.Bundle;
 import androidx.appcompat.app.AppCompatActivity;
 import androidx.fragment.app.FragmentManager;
 import androidx.fragment.app.FragmentTransaction;

 public class MainActivity extends AppCompatActivity {
     private SecondFragment secondFragment;

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

         // Load FirstFragment into the first container
         FirstFragment firstFragment = new FirstFragment();
         FragmentManager fragmentManager = getSupportFragmentManager();
         FragmentTransaction fragmentTransaction1
                 = fragmentManager.beginTransaction();
         fragmentTransaction1.add(R.id.fragment_container1, firstFragment);
         fragmentTransaction1.commit();

         // Load SecondFragment into the second container
         secondFragment = new SecondFragment();
         FragmentTransaction fragmentTransaction2
                         = fragmentManager.beginTransaction();
        fragmentTransaction2.add(R.id.fragment_container2, secondFragment);
         fragmentTransaction2.commit();
     }

     public void updateSecondFragment(String text) {
         secondFragment.updateText(text);
    }
 }
```

Explanation

- **MainActivity Class:** Extends 'AppCompatActivity'.
- **onCreate Method:** Sets the content view to 'activity_main.xml'.
- **FragmentManager:** Manages the fragments within the activity.
- **FragmentTransaction:** Used to add, replace, or remove fragments in the activity.
- **FirstFragment:** An instance of 'FirstFragment' is created and added to 'fragment_container1'.
- **SecondFragment:** An instance of 'SecondFragment' is created and added to 'fragment_container2'.
- **updateSecondFragment Method:** Updates the text in the second fragment by calling the 'updateText' method.

Output





The output behavior of the app is as follows: When the app starts, it displays two fragments side by side within the main activity, each taking up half of the available vertical space. The first fragment contains a button labeled "Click Me," and the second fragment displays a text view with the text "This is the Second Fragment." When the user clicks the button in the first fragment, the text in the second fragment is updated to "Button Clicked!" through the interaction between the fragments facilitated by the main activity's updateSecondFragment method. This demonstrates the communication between fragments using the main activity as a mediator.

3.4.5 Adding Fragments Dynamically

Fragments in Android are modular components that represent portions of the user interface in an activity. They allow for more flexible UI designs and can be reused across different activities. While

fragments can be added statically through XML layout files, adding fragments dynamically provides more flexibility, allowing fragments to be added, replaced, or removed at runtime based on user interactions or application logic.

Dynamic fragment management involves creating and manipulating fragment transactions using the FragmentManager and FragmentTransaction classes. These transactions allow adding, replacing, and removing fragments in response to user actions or other events. This approach is particularly useful in scenarios where the UI needs to adapt dynamically, such as multi-pane layouts in tablets or varying content based on user selections.

Steps to Add Fragments Dynamically:

1. **Set Up the Project :** Create a new Android project in Android Studio.
2. **Create the Fragment Class :** Define a fragment class that will be added dynamically.
3. **Create Fragment Layout:** Define the layout XML file for the fragment.
4. **Modify the Activity Layout :** Add a container (FrameLayout) in the activity's layout file where the fragment will be dynamically added.
5. **Add Fragment Dynamically in Activity :** Use the FragmentManager and FragmentTransaction to add the fragment dynamically within the activity code.

Examples | Steps to Add Fragments Dynamically

1. **Set Up the Project**

   Create a new Android project named 'DynamicFragmentExample'.

2. **Create the Fragment Class**

   Right-click on 'com.example.dynamicfragmentexample', then select 'New -> Java Class', and name it 'ExampleFragment'.

Code | ExampleFragment.java

```
package com.example.dynamicfragmentexample;

import android.os.Bundle;
import androidx.fragment.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class ExampleFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_example, container, false);
    }
}
```

Explanation

This class extends Fragment and overrides the onCreateView method to inflate the fragment's layout. The onCreateView method is called when the fragment is ready to display its UI. The inflater.inflate method is used to create the view hierarchy from the specified XML resource.

3. **Create Fragment Layout:**

Right-click on the 'res/layout' folder, then select 'New -> XML -> Layout XML file', and name it 'fragment_example.xml'.

Code fragment_example.xml

```
<?xml version="1.0" encoding="utf-8"?>
<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="Hello from Fragment"
        android:textSize="20sp" />
</LinearLayout>
```

Explanation

This layout contains a TextView to display a message. The LinearLayout is centered within its parent, and the TextView is placed inside it with a simple text message.

4. **Modify the Activity Layout:**

Open 'activity_main.xml' and add a 'FrameLayout' to serve as the fragment container.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/buttonAddFragment"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Add Fragment"
        android:layout_centerInParent="true"/>
```

```
    <FrameLayout
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_below="@id/buttonAddFragment"/>
</RelativeLayout>
```

Explanation

The layout includes a Button to trigger the addition of the fragment and a FrameLayout that acts as the container for the fragment. The FrameLayout is placed below the button to hold the fragment's UI.

5. **Add Fragment Dynamically in Activity:**

Open 'MainActivity.java' and modify it to add the fragment dynamically.

Code MainActivity.java

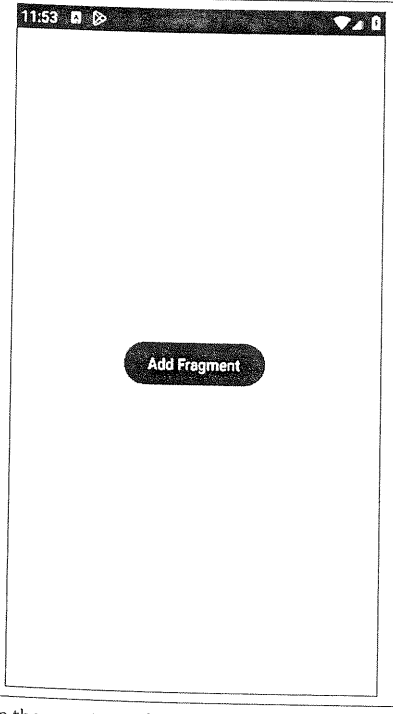
```
package com.example.dynamicfragmentexample;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import androidx.appcompat.app.AppCompatActivity;
import androidx.fragment.app.FragmentManager;
import androidx.fragment.app.FragmentTransaction;

public class MainActivity extends AppCompatActivity {

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

        Button buttonAddFragment = findViewById(R.id.buttonAddFragment);
        buttonAddFragment.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                addFragment();
            }
        });
    }

    private void addFragment() {
        ExampleFragment exampleFragment = new ExampleFragment();
        FragmentManager fragmentManager = getSupportFragmentManager();
```

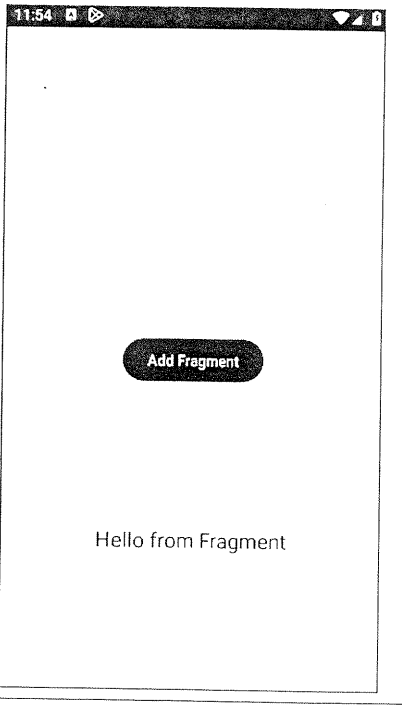
```
        FragmentTransaction fragmentTransaction
                            = fragmentManager.beginTransaction();
        fragmentTransaction.add(R.id.fragment_container, exampleFragment);
        fragmentTransaction.commit();
    }
}
```

Explanation

The MainActivity contains a Button click listener that triggers the addFragment method. When the button is clicked, the addFragment method creates an instance of ExampleFragment. It then uses the FragmentManager to begin a transaction and adds the fragment to the fragment_container specified in the activity layout. Finally, the transaction is committed to display the fragment.

Output





hen the app starts, the main activity displays a button labeled "Add Fragment" centered on the screen. Below e button, there is an empty FrameLayout which serves as a container for dynamically added fragments. itially, this container is empty, so only the button is visible. When the user clicks the "Add Fragment" button, e app dynamically creates an instance of ExampleFragment and adds it to the FrameLayout container. As esult, the fragment's layout, which includes a centered TextView with the text "Hello from Fragment," is played below the button. Each click on the "Add Fragment" button will add the fragment, demonstrating the namic addition of fragments at runtime.

3.5 Understanding the Back Stack in Android

The back stack is a fundamental concept in Android application development, managing the order in which activities are displayed and navigated.

What is the Back Stack?

The back stack is a stack-like data structure used by the Android system to keep track of the sequence of activities the user interacts with. It operates on the principle of Last In, First Out (LIFO), meaning the last activity added to the stack is the first one to be removed when the back button is pressed.

Key Concepts of the Back Stack:

1. **Stack Structure:**
   - Each activity is like an item in a stack. When an activity is started, it is pushed onto the top of the stack.
   - Pressing the back button pops the current activity off the stack, revealing the previous activity.
2. **Activity Lifecycle:**
   - When an activity is pushed onto the stack, it goes through a series of lifecycle callbacks (e.g., onCreate(), onStart(), onResume()).
   - When an activity is popped off the stack, it undergoes callbacks such as onPause(), onStop(), and potentially onDestroy() if it is no longer needed.
3. **Navigation:**
   - **Starting a New Activity:** When starting a new activity, the current activity is paused, and the new activity is pushed onto the stack and becomes the foreground activity.
   - **Going Back:** Pressing the back button will pop the current activity off the stack, resuming the previous activity in the stack.

3.6 Managing Activity Navigation and Back Stack

In Android app development, managing the navigation flow and the back stack of activities is crucial for providing a seamless user experience. The back stack is a stack of activities that the user has navigated through, and it determines the behavior of the back button.

The below simple example demonstrates the basic use of the back stack with two activities, providing a foundation for understanding how activity navigation works in Android.

	Example	Step By Step Implemenation of Back Stack
Step 1: Create the Project: Create an Android project by selecting "Empty Views Activity", naming the project "BackStackExample", and choosing Java as the language.		

Step 2: Update activity_main.xml

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Main Activity"
        android:textSize="24sp"
        android:padding="16dp"/>

    <Button
        android:id="@+id/button_start_second_activity"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Go to Second Activity"
        android:padding="16dp"/>
</LinearLayout>
```

Explanation

- This XML file defines the user interface for 'MainActivity'.
- It uses a 'LinearLayout' with vertical orientation to arrange the 'TextView' and 'Button' elements.
- The 'TextView' displays "Main Activity".
- The 'Button' is labeled "Go to Second Activity" and has an ID 'button_start_second_activity'.

Step 3: Update MainActivity.java:

Code MainActivity.java

```
package com.example.backstackexample;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
```

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

     Button buttonStartSecondActivity = findViewById(R.id.button_start_second_activity);
    buttonStartSecondActivity.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
               Intent intent = new Intent(MainActivity.this, SecondActivity.class);
                 startActivity(intent);
            }
        });
    }
}
```

Explanation

- This Java file defines the behavior of 'MainActivity'.
- It sets the content view to 'activity_main.xml'.
- It finds the button with ID 'button_start_second_activity' and sets an 'OnClickListener'.
- When the button is clicked, it starts 'SecondActivity' using an 'Intent'.

Step 4: Create SecondActivity and Its Layout (activity_second.xml)

Code activity_second.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Second Activity"
        android:textSize="24sp"
        android:padding="16dp"/>
```

```
    <Button
        android:id="@+id/button_back_to_main"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Back to Main Activity"
        android:padding="16dp"/>
</LinearLayout>
```

Explanation

- This XML file defines the user interface for 'SecondActivity'.
- It uses a 'LinearLayout' with vertical orientation to arrange the 'TextView' and 'Button' elements.
- The 'TextView' displays "Second Activity".
- The 'Button' is labeled "Back to Main Activity" and has an ID 'button_back_to_main'.

Code SecondActivity.java

```
package com.example.backstackexample;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import androidx.appcompat.app.AppCompatActivity;

public class SecondActivity extends AppCompatActivity {

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

        Button buttonBackToMain = findViewById(R.id.button_back_to_main);
        buttonBackToMain.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                finish(); // This will close SecondActivity and return to MainActivity
            }
        });
    }
}
```

Explanation

- This Java file defines the behavior of 'SecondActivity'.
- It sets the content view to 'activity_second.xml'.
- It finds the button with ID 'button_back_to_main' and sets an 'OnClickListener'.
- When the button is clicked, it calls 'finish()' to close 'SecondActivity' and return to 'MainActivity'.

Step 5: Declare SecondActivity in AndroidManifest.xml:

Code AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.backstackexample">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.BackStackExample">
        <activity android:name=".SecondActivity" />
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

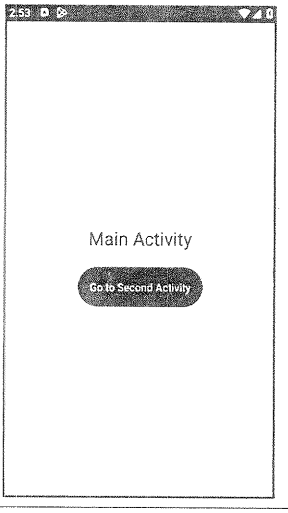
Explanation

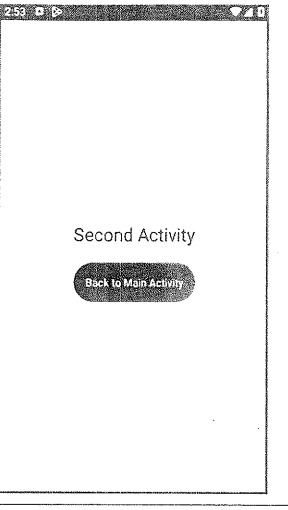
- The 'manifest' tag encapsulates the entire AndroidManifest.xml file.
- The 'application' tag includes the application components.
- 'MainActivity' is declared as the launcher activity with an intent filter.
- 'SecondActivity' is declared without an intent filter.

Step 6: Running the App:

1. Launch the app, and 'MainActivity' is displayed.
2. Click the "Go to Second Activity" button to start 'SecondActivity'.
3. Click the "Back to Main Activity" button to finish 'SecondActivity' and return to 'MainActivity'.

Output





3.7 Implementing a New Activity Using Explicit Intent, Implicit Intent and Content Provider

Let us develop an Android application named "**NewActivityDemo**" that demonstrates how to use explicit intents, implicit intents, and content providers. The main activity of the application will contain three buttons, each of which will open a new activity or perform a specific action. The first button will use an explicit intent to open a second activity within the same app. The second button will use an implicit intent to perform an action such as opening a web browser. The third button will use a content provider to open the device's contacts application.

Step 1: Create a New Project:

Create an Android project by selecting "Empty Views Activity", naming the project "NewActivityDemo", and choosing Java as the language.

Step 2: Set Up the Main Activity Layout:

In this step, we will create the layout for the main activity with three buttons that will trigger different intents.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

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

```
        android:layout_height="wrap_content"
        android:text="Open Explicit Activity"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="100dp"/>

    <Button
        android:id="@+id/button_open_implicit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Open Implicit Intent"
        android:layout_below="@id/button_open_explicit"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp"/>

    <Button
        android:id="@+id/button_open_contacts"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Open Contacts"
        android:layout_below="@id/button_open_implicit"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp"/>
</RelativeLayout>
```

Explanation

- **'RelativeLayout'**: The root layout of the activity, allowing child views to be positioned relative to each other.
- **'Button'**: Three buttons are added, each centered horizontally. The first button is positioned with a margin from the top, and subsequent buttons are placed below the previous ones with margins for spacing.
- The first button will open an explicit activity, the second will trigger an implicit intent, and the third will open the contacts application.

Step 3: Create the Second Activity for Explicit Intent:

We need to create a new activity that will be opened using an explicit intent.

1. Right-click on 'com.example.newactivitydemo'.
2. Select 'New -> Activity -> Empty Activity'.
3. Name the activity 'SecondActivity'.

Code SecondActivity.java

```
package com.example.newactivitydemo;

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

```
public class SecondActivity extends AppCompatActivity {

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

We need to create new Layout File for Second Activity.

1. Right-click on 'res/layout'.
2. Select 'New -> Layout Resource File'.
3. Name the file 'activity_second.xml'.

Code activity_second.xml

```
'''xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to Second Activity"
        android:layout_centerInParent="true"
        android:textSize="20sp"/>
</RelativeLayout>
```

Explanation

- 'SecondActivity.java': This activity extends 'AppCompatActivity' and sets its content view to 'activity_second.xml'.
- 'activity_second.xml': This layout contains a 'RelativeLayout' with a centered 'TextView' displaying "Welcome to Second Activity". This will be shown when the explicit intent is triggered.

Step 4: Update the Manifest File:

We need to declare the new activity in the manifest file to ensure that the Android system recognizes it.

Code AndroidManifest.xml

```
        <activity android:name=".SecondActivity"></activity>
```

Explanation

- 'SecondActivity' is declared in the manifest file within the '<application>' tag to ensure that it is recognized by the Android system.

Step 5: Modify the MainActivity Java File:

We will now handle the button clicks in the main activity to perform the respective actions.

Code MainActivity.java

```
package com.example.newactivitydemo;

import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.provider.ContactsContract;
import android.view.View;
import android.widget.Button;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        Button buttonOpenExplicit = findViewById(R.id.button_open_explicit);
        buttonOpenExplicit.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
               Intent explicitIntent = new Intent(MainActivity.
                                                  this, SecondActivity.class);
                startActivity(explicitIntent);
            }
        });

        Button buttonOpenImplicit = findViewById(R.id.button_open_implicit);
        buttonOpenImplicit.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent implicitIntent = new Intent(Intent.ACTION_VIEW,
                                    Uri.parse("https://www.google.com"));
              if (implicitIntent.resolveActivity(getPackageManager()) != null) {
                    startActivity(implicitIntent);
                }
            }
        });

        Button buttonOpenContacts = findViewById(R.id.button_open_contacts);
        buttonOpenContacts.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
```

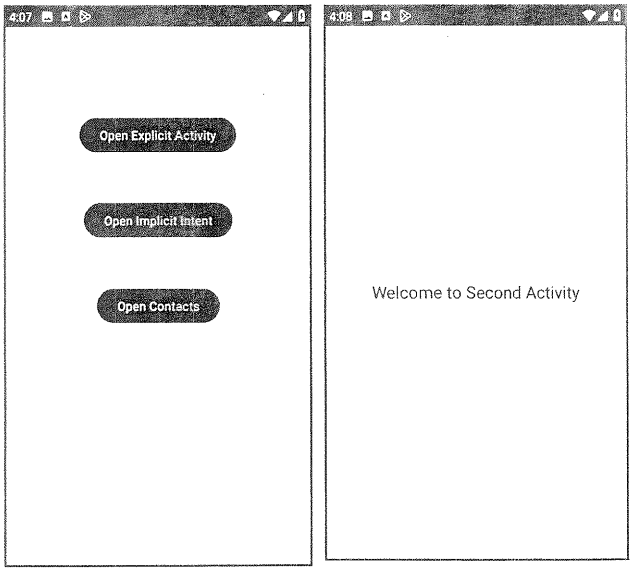
```
                Intent contactsIntent = new Intent(Intent.ACTION_VIEW);
                contactsIntent.setData(ContactsContract.Contacts.CONTENT_URI);
            if (contactsIntent.resolveActivity(getPackageManager()) != null) {
                startActivity(contactsIntent);
            }
        }
    });
  }
}
```

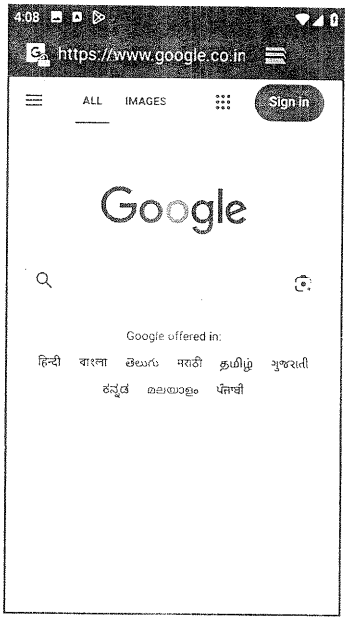
Explanation

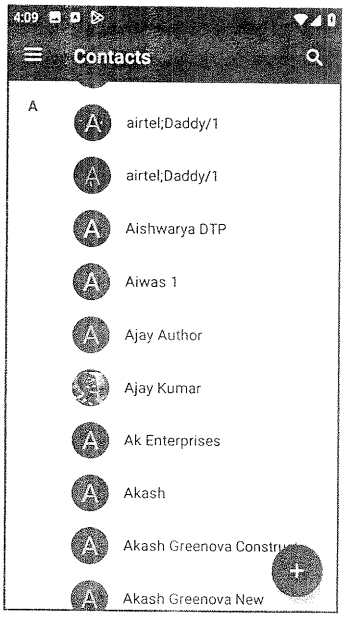
- 'MainActivity' class extends 'AppCompatActivity' and sets the content view to 'activity_main.xml'.
- Three buttons are referenced using 'findViewById'.
- 'buttonOpenExplicit': Sets an 'OnClickListener' that creates an explicit intent to start 'SecondActivity' and calls 'startActivity' to navigate to it.
- 'buttonOpenImplicit': Sets an 'OnClickListener' that creates an implicit intent with the action to view a URL. It checks if there is an app that can handle this intent using 'resolveActivity' before calling 'startActivity'.
- 'buttonOpenContacts': Sets an 'OnClickListener' that creates an intent to view contacts using the 'ContactsContract.Contacts.CONTENT_URI' and starts the contacts app.

Output

When the application runs, the main activity displays three buttons labeled "Open Explicit Activity," "Open Implicit Intent," and "Open Contacts." Clicking the "Open Explicit Activity" button uses an explicit intent to navigate to 'SecondActivity', which displays a message "Welcome to Second Activity." Clicking the "Open Implicit Intent" button uses an implicit intent to open a web browser and navigate to "https://www.google.co.in." Clicking the "Open Contacts" button uses a content provider to open the device's contacts application.







3.8 Review Questions

Section - A

1. What is an activity in Android?
2. Define a fragment in Android ?
3. What is fragment's primary purpose?
4. What is an Intent in Android ?
5. How is an Intent used to start an activity?
6. What is the role of the onCreate() method in the activity lifecycle?
7. What is the setContentView() and findViewById() method used for in an activity?
8. Define Styles and Themes in Android.
9. What is Dialog in Android?
10. What is ProgressBar?
11. List the types of Intents.
12. What are the main differences between explicit and implicit Intents?
13. List two common categories used in Android Intents and their purposes.
14. What is the use of onActivityResult()?
15. Describe the onPause() method in the activity lifecycle.
16. How to invoke Built-in Application Using the Intent Object?
17. What is the function of the putExtra() method in an Intent?

18. What are the benefits of using fragments in Android applications?
19. How do you dynamically add a fragment to an activity in Android?
20. Describe the onStart() method in the fragment lifecycle.
21. What is back stack in Android?
22. What are the two ways of adding fragments to an activity?
23. Explain the purpose of the onAttach() method in a fragment's lifecycle.
24. What is the purpose of the onCreateView() method in a fragment?
25. Describe the purpose of the onDestroyView() method in a fragment.
26. Describe the purpose of the onResume() method in a fragment's lifecycle.
27. List two common actions used in Android Intents
28. How can you declare an activity in the AndroidManifest.xml file?

Section - B

1. Explain the steps to create a new activity in Android.
2. Explain the Features and Functionalities of an Activity.
3. How to create a style and theme in android?
4. How to Hide the Activity Title? Explain with an example.
5. What is Dialog in Android? Explain the Types of Dialogs.
6. How to Display an AlertDialog?
7. How to Use ProgressBar in Android?
8. Explain the Features and Functionalities of Intents.
9. How to Create an Intent? Explain with example.
10. How Explicit Intent and Implicit Intent works?
11. Explain Any Five Intent Class Methods and Attributes.
12. Explain the difference between the startActivity() and startActivityForResult() methods.
13. Explain the steps to Pass Data with Intents.
14. Write the Features and Benefits of Fragments.
15. Discuss the process of adding fragments dynamically in an Android application.

Section - C

1. Describe the complete lifecycle of an Android activity and the purpose of each lifecycle method.
2. Explain the steps involved in applying a Style and Theme to an Activity.
3. Explain the Steps involved in Navigating Between Activities (or) Linking Activities Using Intents.
4. Explain the types of Intents with sample code.
5. What are Intent Filters? Explain the components of Intent Filters.
6. Explain the lifecycle of a fragment in Android, detailing the purpose and use of each lifecycle method.
7. Explain the steps to implement fragments in an Android app, including creating, adding, and managing fragments dynamically.
8. Explain how to manage activity navigation and the back stack in Android.

UNIT - II

CHAPTER 4

UNDERSTANDING THE ANDROID USER INTERFACE

CONTENTS

- Introduction
- Compilation Process of XML Layout Files in Android
- Understanding the Components of a Screen
- Adapting to Display Orientation
- Managing Changes To Screen Orientation
  - What Happens to an Activity's State When Orientation Changes?
  - Persisting State Information during Changes in Configuration
  - Detecting Orientation Changes
  - Controlling the Orientation of the Activity
- Utilizing The Action Bar
  - Creating an ActionBar
  - Adding Action Items to the Action Bar
- Creating The User Interface Programmatically
- Listening For UI Notifications
  - Listening for UI Notifications at the Activity Level
  - Registering Views for Notifications
- Review Questions

4.1 Introduction

In Chapter 3, the Activity class and its life cycle were discussed. It was established that an activity serves as a means for users to interact with the application. However, an activity on its own does not have a presence on the screen. Instead, it utilizes Views and ViewGroups to render the screen. This chapter delves into the details of creating user interfaces in Android and the ways in which users interact with them. Additionally, it covers how to manage changes in screen orientation on Android devices.

4.2 Compilation Process of XML Layout Files in Android

We have studied the basic unit of an Android application is an activity. An activity displays the user interface (UI) of the application, which may contain widgets such as buttons, labels, textboxes, and more. Typically, the UI is defined using an XML file (e.g., the main_activity.xml file located in the res/ layout folder of the project), which may look similar to the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
</LinearLayout>
```

When an Android application is compiled, the XML layout file defining the user interface (UI) components is processed to generate the corresponding Android GUI classes. This process involves converting the XML elements and attributes into Java objects and methods that define the UI components and their properties. Let us understand how this compilation process works:

1. **XML Layout File Processing:**
   - The XML layout file, such as 'main_activity.xml', contains the structure and attributes of the UI components like 'TextView', 'Button', 'LinearLayout', etc.
   - Each XML element represents a specific UI component, and the attributes define properties like dimensions, text content, colors, and positioning.
2. **Compilation into Android GUI Classes:**
   - When the Android application is compiled, the build process includes parsing the XML layout files and generating corresponding Java classes that represent the UI components.
   - Each XML element is translated into its equivalent Android GUI class, such as 'TextView' for '<TextView>', 'Button' for '<Button>', and so on.
   - The attributes defined in the XML file are mapped to setter methods in the generated Java classes. For example, 'android:layout_width="fill_parent"' in XML corresponds to 'setLayoutWidth(ViewGroup.LayoutParams.FILL_PARENT)' in Java.

3. **Loading the UI in 'onCreate()' Method:**
   - In the 'onCreate()' method of the Activity class, the 'setContentView()' method is called to load the UI defined in the XML layout file.
   - The 'R.layout.main' parameter specifies the resource ID of the layout file to be inflated and displayed on the screen.
   - By calling 'setContentView(R.layout.main)', the Android system inflates the XML layout, creates the corresponding view hierarchy, and sets it as the content view of the activity.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}
```

4. **UI Creation at Runtime:**
   - When the activity is launched, the Android system executes the 'onCreate()' method, where the UI layout is loaded dynamically at runtime.
   - The UI components defined in the XML layout file are instantiated as Java objects, configured with the specified attributes, and added to the activity's view hierarchy.
   - The UI elements are rendered on the screen based on the layout parameters and properties set in the XML file, providing the visual representation of the application's interface.

4.3 Understanding the Components of a Screen

In Android app development, understanding the components of a screen is important for creating engaging and user-friendly interfaces. A screen in an Android app is composed of various elements that work together to present information and interact with users.

These components include:

1. **Views:** Views are the building blocks of an Android app's user interface. They represent UI elements such as buttons, text fields, images, and more. Each view is a subclass of the View class and is responsible for drawing itself on the screen and handling user interactions.
2. **ViewGroups:** ViewGroups are special types of views that can contain other views or viewgroups. They are used to organize and arrange multiple views on the screen. Common ViewGroup classes include LinearLayout, RelativeLayout, and ConstraintLayout.
3. **Layouts:** Layouts are XML files that define the structure of a screen. They specify the arrangement and hierarchy of views and ViewGroups. Android provides a variety of layout types, such as LinearLayout, RelativeLayout, FrameLayout, ConstraintLayout, and more, each with its own way of arranging child views.
4. **Widgets:** Widgets are interactive elements that users can add to their home screens or app widgets. Examples of widgets include buttons, text boxes, and progress bars. Widgets are implemented using the AppWidgetProvider class.

5. **Resources:** Resources are external elements such as images, strings, colors, and dimensions that are used to enhance the user interface of an app. They are stored in the res directory of an Android project and can be accessed programmatically.

4.3.1 View

In Android development, a View is a fundamental building block used to create the user interface (UI) of an application. Views are responsible for displaying interactive elements on the screen such as buttons, text fields, images, and more. Understanding Views is crucial for designing visually appealing and functional Android apps. All Views in Android are subclasses of the **android.view.View** class, which provides basic functionality for rendering and interacting with UI elements.

What is a View in Android?

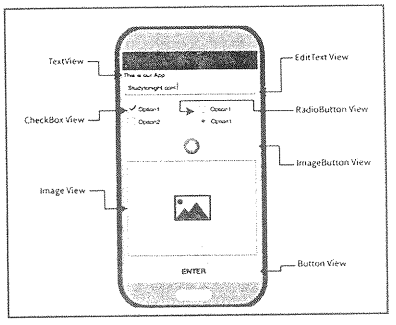
A View in Android is a fundamental UI (User Interface) component that represents a rectangular area on the screen. It is responsible for drawing content and handling user interactions. Views are the building blocks for creating user interfaces in Android applications. Examples of views include buttons, text fields, images, and other interactive elements.

Key Concepts and Features of Views:

1. **Appearance and Behavior:**
   - Views define the visual appearance and behavior of UI elements, such as buttons, text fields, checkboxes, and more.
   - Views have various attributes that define their size, position, appearance, and behavior. Common attributes include layout_width, layout_height, padding, margin, background, visibility, etc.
2. **Hierarchy:**
   - Views can be organized in a hierarchical structure to create complex UI layouts. A View can be a parent or a child of other Views.
   - Parent Views contain and manage child Views, forming a tree-like structure known as the View hierarchy.
   - The parent View is usually a ViewGroup, which can contain multiple child Views.
3. **Event Handling:**
   - Views can respond to user interactions like clicks, touches, swipes, and more through event handling mechanisms.
   - Event listeners can be attached to Views to execute specific actions in response to user input.
4. **Rendering:**
   - Views are responsible for rendering themselves on the screen. This involves drawing the content within the View's rectangular area.

Common Views in Android:

Some common views used in Android applications, along with their XML definitions are given below. These views are essential components in creating interactive and user-friendly interfaces in Android applications. Each view type has its own unique attributes and behaviors, allowing for flexible and dynamic UI design.



1. **TextView:**
   - **Description:**
     - Displays text to the user.
     - Commonly used for labels, instructions, or any text content.

Code

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, World!"
    android:textSize="20sp"
    android:textColor="#000000"/>
```

Explanation

- 'android:layout_width="wrap_content"': The width of the TextView will be just enough to fit the content.
- 'android:layout_height="wrap_content"': The height of the TextView will be just enough to fit the content.
- 'android:text="Hello, World!"': Sets the text displayed by the TextView.
- 'android:textSize="20sp"': Sets the size of the text to 20 scale-independent pixels (sp).
- 'android:textColor="#000000"': Sets the color of the text to black.

2. **Button:**
   - **Description:**
     - A push-button that can be pressed or clicked by the user.
     - Used to perform actions when clicked.

Code

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click Me"/>
```

Explanation

- 'android:layout_width="wrap_content"': The width of the Button will be just enough to fit the content.
- 'android:layout_height="wrap_content"': The height of the Button will be just enough to fit the content.
- 'android:text="Click Me"': Sets the text displayed on the Button.

3. **EditText:**
   - **Description:**
     - A text field that allows users to input text.
     - Commonly used for forms and input fields.

Code

```
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Enter text"/>
```

Explanation

- 'android:layout_width="match_parent"': The width of the EditText will expand to match the width of its parent container.
- 'android:layout_height="wrap_content"': The height of the EditText will be just enough to fit the content.
- 'android:hint="Enter text"': Sets a hint that is displayed when the EditText is empty, prompting the user for input.

4. **ImageView:**
   - **Description:**
     - Displays an image resource.
     - Used for displaying images, icons, or any drawable resources.

Code

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/sample_image"/>
```

Explanation

- 'android:layout_width="wrap_content"': The width of the ImageView will be just enough to fit the content.
- 'android:layout_height="wrap_content"': The height of the ImageView will be just enough to fit the content.
- 'android:src="@drawable/sample_image"': Sets the image resource to be displayed in the ImageView. The '@drawable/sample_image' reference should point to an image resource in the 'res/drawable' directory.

5. **CheckBox**
   - **Description:**
     - A checkbox that can be checked or unchecked.
     - Used for binary choices or options in forms.

Code

```
<CheckBox
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Check me"/>
```

Explanation

- 'android:layout_width="wrap_content"': The width of the CheckBox will be just enough to fit the content.
- 'android:layout_height="wrap_content"': The height of the CheckBox will be just enough to fit the content.
- 'android:text="Check me"': Sets the text displayed next to the CheckBox.

6. **RadioButton:**
   - **Description:**
     - A radio button, usually used in a group where only one option can be selected at a time.
     - Commonly used in forms where a single choice must be made from multiple options.

Code

```
<RadioButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Option 1"/>
```

Explanation

- 'android:layout_width="wrap_content"': The width of the RadioButton will be just enough to fit the content.
- 'android:layout_height="wrap_content"': The height of the RadioButton will be just enough to fit the content.
- 'android:text="Option 1"': Sets the text displayed next to the RadioButton.

Advantages of Views

1. **Modularity:** Views are modular components that represent UI elements. It is easy to reuse and customize them across different parts of the application.
2. **Interactivity:** Views can handle user interactions like clicks, touches, and gestures to create interactive and engaging user interfaces.
3. **Customization:** Views can be customized with attributes like colors, sizes, fonts, and styles to match the design requirements of the application.

4. **Accessibility:** Views can be made accessible by providing content descriptions, labels, and focusability for users with disabilities.
5. **Event Handling:** Views can respond to events like clicks, touches, and key presses to implement specific actions based on user interactions.

Disadvantages of Views

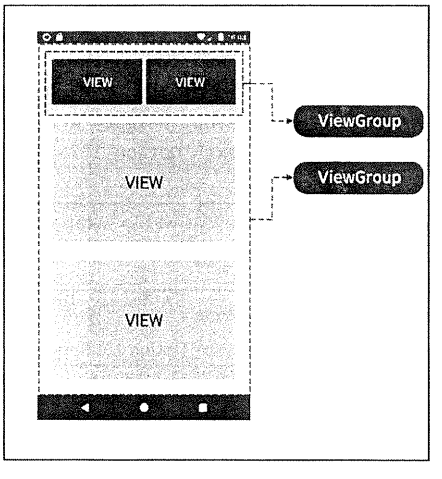
1. **Limited Layout Control:** Views do not have built-in layout capabilities and hence it is challenging to control the positioning and arrangement of Views within the UI.
2. **Complex UIs:** Creating complex user interfaces with multiple Views may lead to nested View structures. It increases the complexity of managing layout hierarchies.
3. **Performance Impact:** Having a large number of Views on the screen can impact performance, especially on older devices with limited resources.
4. Maintenance: Managing individual Views scattered across the application can make maintenance and updates more difficult.

4.3.2 Viewgroup

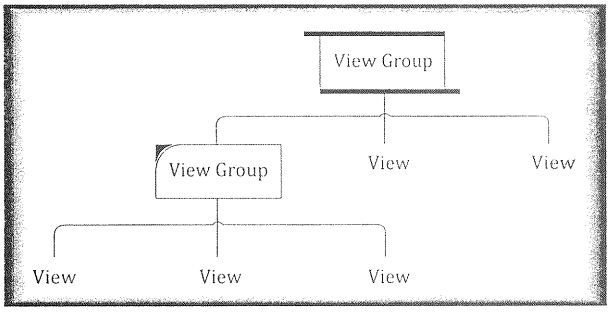
A ViewGroup is a special type of View that acts as a container for other Views to organize and manage the layout of UI components. ViewGroup extends the functionality of Views by providing a way to arrange and control the positioning of multiple child Views within a single container. In simple terms, ViewGroup is a special type of view that can contain other views (including other view groups) and define their layout properties.

Think of a ViewGroup as a container or a box that can hold other smaller boxes inside it. These smaller boxes can be buttons, text fields, images, or even other containers. These smaller boxes can be arranged and organized within a container using a Layout. A ViewGroup helps in organizing these smaller boxes in a structured way. It decides how these boxes (views) are placed on the screen. ViewGroup is the base class for all layouts and view containers in Android.

Example: LinearLayout, RelativeLayout, ConstraintLayout are considered as ViewGroups because they can hold and organize multiple Views.



ViewGroup forms a hierarchical structure where parent ViewGroups can contain various types of child Views and even nested ViewGroups to create complex and flexible UI designs. ViewGroup plays a crucial role in structuring the visual layout of Android apps and controlling the positioning, sizing, and alignment of UI elements within the user interface.



View hierarchy, which defines a UI layout

What is a View Group in Android?

ViewGroup is a special type of view that can contain other views (including other view groups) and define their layout properties. ViewGroup extends the functionality of Views by providing a way to arrange and control the positioning of multiple child Views within a single container.

Features of ViewGroup

1. **Layout Management:**
   - ViewGroup is responsible for defining the layout structure of the UI by arranging child Views in a specific order and position.
   - Different types of ViewGroups offer various layout behaviors like linear arrangement, relative positioning, grid layout, etc.
2. **View Hierarchy:**
   - ViewGroup forms a hierarchical structure where parent ViewGroups can contain multiple child Views and even nested ViewGroups.
   - The hierarchy allows for complex and flexible UI designs with nested layouts and organized components.
3. **Positioning and Sizing:**
   - ViewGroup controls the positioning, sizing, and alignment of child Views within its boundaries.
   - Developers can specify layout parameters like width, height, margins, and gravity to control how child Views are displayed.
4. **Event Propagation:**
   - ViewGroup manages the propagation of touch events and input gestures to child Views based on their positions and touch intercepting behavior.
   - Event handling in ViewGroup allows for interactive user experiences and touch responsiveness in the UI.

Examples of ViewGroups

Android supports the following ViewGroups:

1. **Linear Layout:** Arranges UI elements either vertically or horizontally in a single row or column.
2. **Relative Layout:** Positions UI elements relative to each other or the parent layout.
3. **Constraint Layout:** Allows creating complex layouts by setting constraints between UI elements.
4. **Frame Layout:** Places UI elements on top of each other, useful for layering views.
5. **Table Layout:** Organizes UI elements in rows and columns, similar to an HTML table.
6. **Scroll View:** Enables scrolling through a list of views that exceed the screen size.

These components work together to create flexible and responsive user interfaces in Android applications.

Creating and Using ViewGroups:

1. ViewGroups can be defined in XML layout files to structure the UI hierarchy and define the layout relationships between Views.
2. Developers can nest multiple ViewGroups within each other to create complex layouts with different arrangements and behaviors.
3. ViewGroup properties like padding, margins, gravity, and layout parameters can be customized to achieve the desired UI design.

Example Simple ViewGroup with a TextView, EditText, Button, and CheckBox

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enter your name:"
        android:textSize="18sp"
        android:padding="8dp"/>

    <EditText
        android:id="@+id/editTextName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Name"
        android:padding="8dp"/>
```

```
    <CheckBox
        android:id="@+id/checkBoxAgree"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I agree to the terms and conditions"
        android:padding="8dp"/>

    <Button
        android:id="@+id/buttonSubmit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Submit"
        android:padding="8dp"/>
</LinearLayout>
```

Advantages of ViewGroup

1. **Layout Management:** ViewGroup provides layout management capabilities to organize and position multiple child Views within a single container.
2. **Hierarchical Structure:** ViewGroup forms a hierarchical structure where parent ViewGroups can contain nested ViewGroups and Views to facilitate the creation of complex UI designs.
3. **Flexibility:** ViewGroup offers flexibility in arranging and controlling the layout of UI components to create responsive and adaptive user interfaces.
4. **Event Propagation:** ViewGroup manages event propagation to child Views to simplify the handling of touch events and interactions within the UI.
5. **Reusability:** ViewGroup can be reused across different parts of the application to maintain consistent layout structures and design patterns.

Disadvantages of ViewGroup

1. **Complexity:** Managing nested ViewGroups and their relationships can introduce complexity, especially in large and complex UI designs.
2. **Performance Overhead:** Using multiple nested ViewGroups can impact performance, especially if not optimized properly.
3. **Learning Curve:** Understanding the behavior and interactions of different ViewGroups may require additional learning compared to individual Views.
4. **Maintenance Challenges:** Updating and maintaining complex ViewGroup structures may require careful attention to avoid layout inconsistencies and conflicts.

4.3.3 Layouts

Layouts in Android are a type of ViewGroup used to define the structure and arrangement of views on the screen. They determine how child views are positioned, sized, and aligned within a container. Layouts provide a flexible way to create complex user interfaces by organizing views in different ways. Think of a Layout as a blueprint or a template that guides where each element should be placed on the screen.

Layouts are defined in XML files located in the res/layout directory of the Android project. XML tags and attributes are used to specify the type of layout and its properties.

Example: LinearLayout, RelativeLayout, ConstraintLayout are different types of Layouts used to design the user interface of an Android app.

What is a Layout in Android?

Layouts in Android are containers that hold and arrange UI elements, such as buttons, text views, images, etc., within an Activity or Fragment. They determine the structure, positioning, and sizing of these UI elements on the screen.

How Layout is Different from ViewGroup ?

- Layout is primarily focused on defining the structure and arrangement of UI elements on the screen, while a ViewGroup is a container that can hold and manage multiple Views.
- Layouts are used to create the overall design and structure of the user interface, while ViewGroups provide a way to group and organize Views within that design.
- Layouts are a type of ViewGroup, but not all ViewGroups are Layouts. ViewGroups can contain any type of View, including other ViewGroups.elements on the screen.

Example | **Understanding View, ViewGroup, and Layout in Android**

Views are individual UI elements, ViewGroups are containers that hold multiple Views, and Layouts define how Views and ViewGroups are organized on the screen. Understanding the distinction between Views, ViewGroups, and Layouts is essential for designing effective and visually appealing user interfaces in Android applications.

Consider a scenario where a button and text field are individual Views. When these two Views are combined within a LinearLayout, the LinearLayout acts as a ViewGroup (ViewGroup 1) that holds and organizes them together.

Additionally, if a checkbox and radio button are structured using a RelativeLayout, this RelativeLayout serves as another ViewGroup (ViewGroup 2) for these Views. The RelativeLayout specifies the positioning of the checkbox and radio button relative to each other.

Each ViewGroup follows a different layout strategy: LinearLayout focuses on arranging Views in a linear manner, while RelativeLayout emphasizes the relative positioning of Views.

By combining ViewGroup 1 (LinearLayout with button and text) and ViewGroup 2 (RelativeLayout with checkbox and radio button) into a new ViewGroup, we create a higher-level container that encapsulates both sets of Views. This combined ViewGroup represents a more complex layout structure that integrates different layout approaches.

Key Characteristics of Layouts in Android

Layouts in Android have several key characteristics that define how UI elements are organized and displayed on the screen. The essential characteristics of layouts are:

1. **Structure:**
   - Layouts define the overall structure and arrangement of UI elements within an Android app.

- They act as containers that hold and organize various Views and ViewGroups.

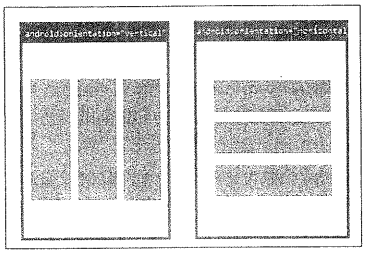
2. **Positioning:**
   - Layouts determine the positioning of UI elements relative to each other or the parent layout.
   - They specify where each UI element should be placed on the screen.
3. **Sizing:**
   - Layouts control the size of UI elements within the layout.
   - They define how much space each UI element occupies on the screen.
4. **Orientation:**
   - Some layouts, like LinearLayout, allow arranging UI elements either vertically or horizontally.
   - The orientation of the layout affects the flow and alignment of UI elements.
5. **Nesting:**
   - Layouts can be nested within each other to create complex UI designs.
   - This nesting allows for creating intricate and layered user interfaces.
6. **Flexibility:**
   - Layouts provide flexibility in designing responsive UIs that adapt to different screen sizes and orientations.
   - ConstraintLayout, for example, offers powerful tools for creating adaptive layouts.
7. **Alignment:**
   - Layouts enable the alignment of UI elements within the layout.
   - They allow for precise control over how elements are aligned horizontally and vertically.
8. **Margins and Padding:**
   - Layouts support setting margins around UI elements to create spacing between them.
   - Padding within layouts defines the space between the edge of the layout and its child elements.
9. **Scalability:**
   - Layouts should be scalable to accommodate varying screen sizes and resolutions.
   - They should adapt well to different devices without compromising the user experience.
10. **Performance:**
    - Efficient use of layouts is crucial for optimizing app performance.
    - Avoiding nested layouts and using appropriate layout types can improve rendering speed.
11. **XML Representation:**
    - Layouts are typically defined in XML files, making it easy to visualize and modify the UI structure.
    - XML tags and attributes are used to specify the type of layout and its properties.
12. **Dynamic Layout Creation:**
    - Layouts can also be created programmatically in Java/Kotlin code rather than using XML files.
    - This allows for dynamic UI creation based on runtime conditions.

Types of Layouts:

1. **Linear Layout:** Arranges UI elements either vertically or horizontally in a single row or column.
2. **Relative Layout:** Positions UI elements relative to each other or the parent layout.
3. **Constraint Layout:** Allows creating complex layouts by setting constraints between UI elements.
4. **Frame Layout:** Places UI elements on top of each other, useful for layering views.
5. **Table Layout:** Organizes UI elements in rows and columns, similar to an HTML table.
6. **Scroll View:** Enables scrolling through a list of views that exceed the screen size.

4.3.3.1 Linear Layout

LinearLayout is one of the simplest and most commonly used layout managers in Android. It arranges its child views in a single row (horizontally) or a single column (vertically). This straightforward layout structure makes it easy to align and position views in a linear fashion.



What is a Linear Layout ?

A LinearLayout organizes its children into a single row or column. Linear Layout organizes Views in a linear direction either horizontally or vertically based on the orientation attribute. Views are placed one after another in the order they are defined within the layout.

	Example	Understanding Linear Layout

In a room with furniture like chairs and tables, Linear Layout in Android arranges items in a straight line, either horizontally or vertically. It's similar to placing chairs in a row or stacking books on a shelf. Each item is positioned next to the other in the order they are added to create a simple and linear arrangement. If more items are added, they will follow the same line, similar to adding more chairs in a row. Linear Layout helps in organizing elements in a straightforward manner, much like arranging items in a room in a linear fashion.

Key Features

1. **Orientation:**
   - Determines whether child views are aligned horizontally or vertically.
   - Controlled by the 'android:orientation' attribute.
2. **Weight Distribution:**
   - Distributes extra space within a LinearLayout.
   - Controlled by the 'android:layout_weight' attribute.

3. **Gravity:**
   - Specifies how child views should be positioned within the LinearLayout.
   - Controlled by the 'android:gravity' attribute.
4. **Margins and Padding:**
   - Provides spacing around and inside child views.
   - Controlled by the 'android:layout_margin' and 'android:padding' attributes.
5. **Baseline Alignment:**
   - Aligns text baselines of child views.
   - Controlled by the 'android:baselineAligned' attribute.

Attributes

Attribute	Description	Example
android:orientation	Sets the orientation of the layout (horizontal or vertical)	android:orientation="vertical"
android:layout_weight	Distributes extra space among children	android:layout_weight="1"
android:gravity	Aligns all children within the parent	android:gravity="center"
android:layout_gravity	Aligns individual children within the parent	android:layout_gravity="right"
android:padding	Sets the padding inside the parent layout	android:padding="16dp"
android:layout_margin	Sets the margin around each child view	android:layout_margin="8dp"
android:baselineAligned	Aligns baselines of children (text alignment)	android:baselineAligned="false"

When to Choose LinearLayout

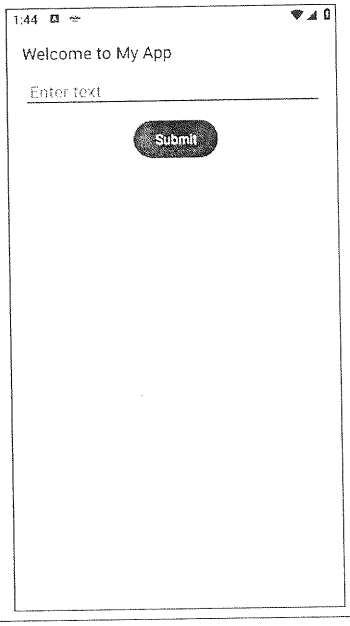
- When a simple, linear arrangement of child views is needed.
- When vertical or horizontal alignment of views is required.
- When the number of child views is small, as deep hierarchies can affect performance.

Example Vertical LinearLayout

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to My App"
        android:textSize="18sp"
        android:layout_marginTop="20dp"
        android:layout_marginBottom="8dp"/>
```

```
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter text"
        android:padding="8dp"
        android:layout_marginBottom="8dp"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Submit"
        android:layout_gravity='certer"/>
</LinearLayout>
```

Output



Explanation

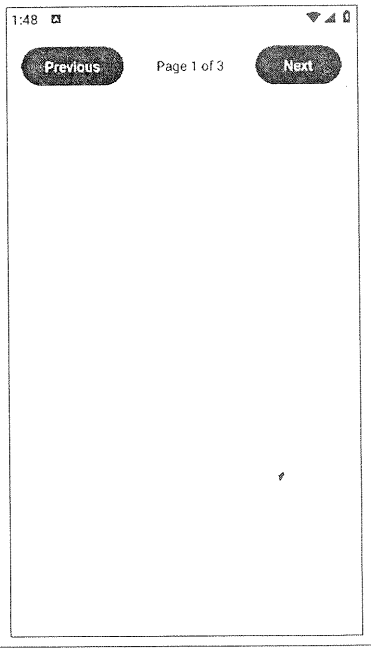
This XML layout defines a simple user interface using a 'LinearLayout' with a vertical orientation. The 'LinearLayout' is set to match the parent width and height, with 16dp padding around its edges. Inside the 'LinearLayout', there are three child views: a 'TextView', an 'EditText', and a 'Button'. The 'TextView' displays the text "Welcome to My App" with a text size of 18sp and margins of 20dp at the top and 8dp at the bottom, providing some spacing around it. Below the 'TextView', the 'EditText' allows for user input with a hint "Enter text", a padding of 8dp inside it, and a bottom margin of 8dp to create space between it and the next view. Lastly, the 'Button' with the text "Submit" is centered horizontally within the parent layout using the 'layout_gravity' attribute.

Examples Horizontal LinearLayout

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    android:padding="16dp"
    android:layout_marginTop="20dp">
```

```
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Previous"
        android:layout_marginRight="8dp"/>

    <TextView
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Page 1 of 3"
        android:layout_weight="1"
        android:gravity="center"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Next"
        android:layout_marginLeft="8dp"/>
</LinearLayout>
```

Output



Explanation

This XML layout defines a horizontal LinearLayout that spans the full width and height of its parent, with a 16dp padding around its edges and a 20dp top margin. Inside the LinearLayout, there are three child views arranged horizontally: a Button, a TextView, and another Button. The first Button, labeled "Previous," is set to wrap its content and has an 8dp right margin to create space between it and the TextView. The TextView displays "Page 1 of 3" and is centered horizontally within its allocated space, occupying the remaining width due to its layout_weight attribute set to 1. This attribute ensures the TextView takes up the remaining space between the two buttons. The second Button, labeled "Next," also wraps its content and has an 8dp left margin to separate it from the TextView. This setup creates a user interface with navigation buttons on either side of a centered page indicator.

Advantages of Linear Layout

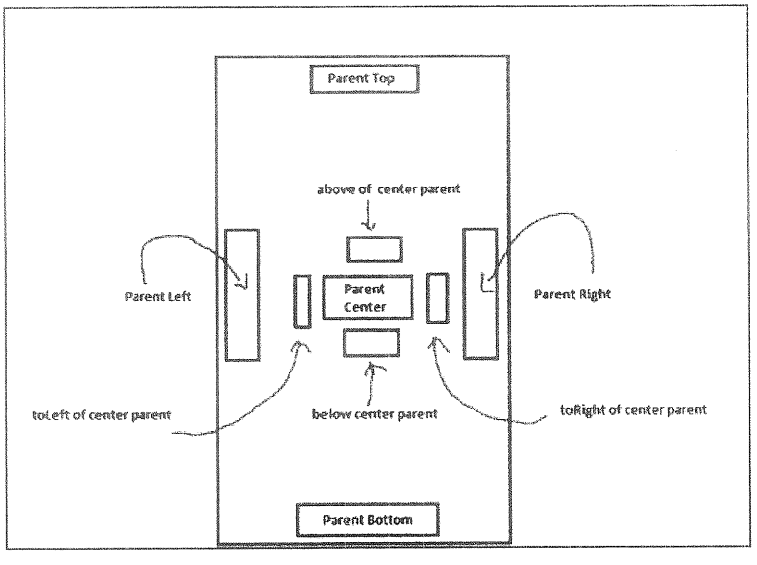
- **Simplicity:** Easy to use and understand, making it ideal for simple layouts.
- **Predictability:** Provides predictable and straightforward arrangement of child views.
- **Control:** Offers fine control over the layout with weight distribution, gravity, and margins.

Disadvantages of Linear Layout

- **Performance:** Can lead to performance issues with deep hierarchies, as each child view adds to the rendering complexity.
- **Flexibility:** Less flexible compared to other layouts like ConstraintLayout, especially for more complex UIs.
- **Spacing:** Requires careful management of spacing and alignment, especially when dealing with different screen sizes and resolutions.

4.3.3.2 RelativeLayout

RelativeLayout is a versatile layout manager in Android that allows for flexible positioning of its child views relative to each other or to the parent container. This makes it possible to create complex and responsive user interfaces with a relatively flat view hierarchy.



What is a Relative Layout ?

A RelativeLayout arranges its child views based on their relationships to one another or to the parent layout. For example, one view can be placed to the right of another view, or aligned to the top of the parent.

Example	Understanding Relative Layout
In a room with furniture like chairs and tables, Relative Layout in Android helps position items in relation to each other. It's like arranging furniture based on how they relate within the room. For example, we can place a chair next to a table or align a lamp with a sofa. Each item's position is determined by its connection to other items, allowing for a layout that adjusts based on these relationships. If we move one item, others positioned relative to it will shift accordingly, similar to how a chair in a room moves based on their connections with table and other furniture. Relative Layout makes it easy to create flexible arrangements much like organizing furniture in a room based on their relationships.	

Key Features

1. **Relative Positioning:**

   Views can be positioned relative to the parent layout or to other views within the layout.

2. **Alignment:**

   Child views can be aligned to the left, right, top, bottom, or center of the parent layout.

3. **Layering:**

   Views can overlap each other, which is useful for creating layered UI effects.

4. **Margins and Padding:**

   Provides control over the spacing between views and within views.

5. **Baseline Alignment:**

   Aligns the baselines of text in different views for a clean and aligned appearance.

Attributes

Attribute	Description	Example
android:layout_alignParentTop	Aligns the view to the top of the parent	android:layout_alignParentTop="true"
android:layout_below	Positions the view below another view	android:layout_below="@id/otherView"
android:layout_toRightOf	Positions the view to the right of another view	android:layout_toRightOf="@id/otherView"
android:layout_centerInParent	Centers the view within the parent layout	android:layout_centerInParent="true"
android:layout_margin	Specifies the space around the view	android:layout_margin="8dp"
android:layout_alignBaseline	Aligns the baseline of the view with another view	android:layout_alignBaseline="@id/otherView"

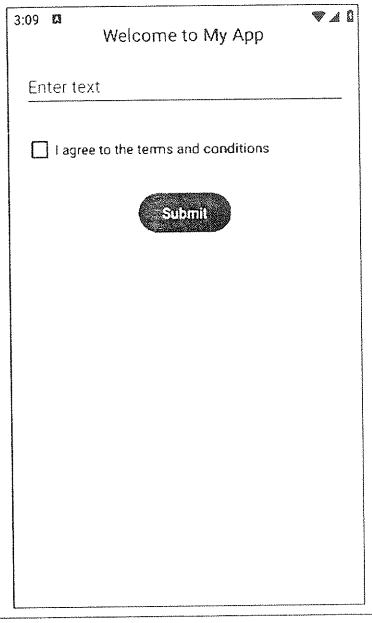
When to Choose RelativeLayout

- When the UI requires complex positioning of views relative to each other or the parent.
- When a flat view hierarchy is desired to improve performance.

- When creating dynamic and responsive layouts that adapt to different screen sizes and orientations.

Example RelativeLayout Example

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to My App"
        android:textSize="18sp"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"/>

    <EditText
        android:id="@+id/editTextName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter text"
        android:layout_below="@id/textView"
        android:layout_marginTop="16dp"/>

    <CheckBox
        android:id="@+id/checkBoxAgree"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I agree to the terms and conditions"
        android:layout_below="@id/editTextName"
        android:layout_marginTop="16dp"/>

    <Button
        android:id="@+id/buttonSubmit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Submit"
        android:layout_below="@id/checkBoxAgree"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="16dp"/>
</RelativeLayout>
```

Output	Explanation
	• **RelativeLayout:** The root layout that arranges child views relative to each other and to the parent layout. • **TextView:** Positioned at the top and centered horizontally in the parent using 'android:layout_alignParentTop' and 'android:layout_centerHorizontal'. • **EditText:** Positioned below the 'TextView' with a margin on top using 'android:layout_below' and 'android:layout_marginTop'. • **CheckBox:** Positioned below the 'EditText' with a margin on top using 'android:layout_below' and 'android:layout_marginTop'. • **Button:** Positioned below the 'CheckBox' and centered horizontally in the parent using 'android:layout_below', 'android:layout_centerHorizontal', and 'android:layout_marginTop'.

Advantages of Relative Layout

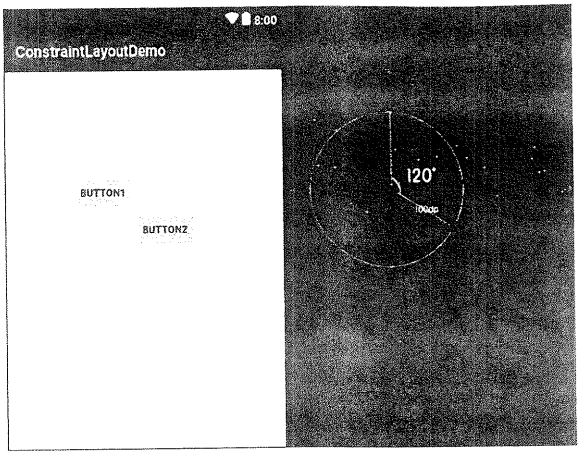
- **Flexibility:** Allows for flexible positioning of child views relative to each other and to the parent.
- **Flat Hierarchy:** Helps maintain a flat view hierarchy, which can improve performance.
- **Dynamic Layouts:** Facilitates the creation of dynamic and responsive layouts that adapt to different screen sizes and orientations.

Disadvantages of Relative Layout

- **Complexity:** Can become complex and difficult to manage with many views and complex relationships.
- **Performance:** While better than deeply nested layouts, it may still incur performance costs if not used judiciously with many views.
- **Readability:** The XML layout can become harder to read and maintain with complex relationships and many attributes.

4.3.3.3 Constraint Layout

ConstraintLayout is a powerful and flexible layout manager available in Android. It allows to create large and complex layouts with a flat view hierarchy, reducing the nesting of view groups, which can improve performance. ConstraintLayout is part of the Android Support Library, which makes it compatible with a wide range of Android versions.



What is a Constraint Layout ?

A ConstraintLayout positions and sizes its child views using a set of constraints. These constraints are rules that define the relationship of a child view to other views or to the parent layout. This allows for flexible and responsive design as views can be positioned relative to each other or the parent container in various ways.

Example | Understanding Constraint Layout

In a room with furniture like chairs and tables, think of Constraint Layout in Android as a special tool that sets rules for the spacing between each piece of furniture and their alignment with walls or other items. For example, we can position a chair 20 inches away from a wall and align it with a table. If the table is moved, the chair will adjust its position automatically based on the rules set. This tool helps in creating a tidy and structured layout similar to arranging furniture in a room but on a digital screen.

Key Features

1. **Flat View Hierarchy:**

   Reduces the need for nested view groups, leading to improved performance and simpler layouts.

2. **Flexible Constraints:**

   Allows for complex positioning and alignment of views using constraints like 'start', 'end', 'top', 'bottom', 'baseline', and 'center'.

3. **Guidelines and Barriers:**

   Supports invisible guidelines to align views and barriers to create flexible group constraints.

4. **Dimension Constraints:**

   Supports 'wrap_content', 'match_constraint' (0dp), and fixed dimensions, providing fine control over view sizes.

5. **Chaining:**

   Enables chaining of views horizontally or vertically to distribute space evenly and create complex layouts with minimal effort.

Attributes

Attribute and Description	Example
app:layout_constraintTop_toTopOf Positions the top edge of a view relative to another view or parent	app:layout_constraintTop_toTopOf="parent"
app:layout_constraintBottom_toBottomOf Positions the bottom edge of a view relative to another view or parent	app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintStart_toStartOf Positions the start edge of a view relative to another view or parent	app:layout_constraintStart_toStartOf="parent"
app:layout_constraintEnd_toEndOf Positions the end edge of a view relative to another view or parent	app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintHorizontal_bias Adjusts the horizontal bias (position) between start and end constraints	app:layout_constraintHorizontal_bias="0.5"
app:layout_constraintVertical_bias Adjusts the vertical bias (position) between top and bottom constraints	app:layout_constraintVertical_bias="0.5"
app:layout_constraintWidth_default Specifies the default width behavior, such as wrap or match constraint (0dp)	app:layout_constraintWidth_default="wrap"
app:layout_constraintHeight_default Specifies the default height behavior, such as wrap or match constraint (0dp)	app:layout_constraintHeight_default="wrap"
app:layout_constraintWidth_min Sets the minimum width of a view	app:layout_constraintWidth_min="50dp"
app:layout_constraintHeight_min Sets the minimum height of a view	app:layout_constraintHeight_min="50dp"

When to Choose ConstraintLayout

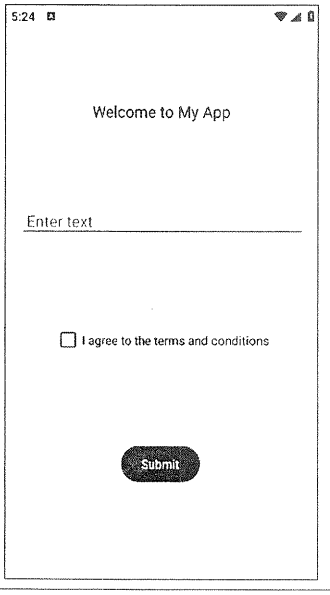
- When designing complex and responsive layouts that need precise control over positioning and alignment.
- When aiming to maintain a flat view hierarchy for better performance.
- When dealing with dynamic content and varying screen sizes, making flexible layouts necessary.

Example | ConstraintLayout

```
<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"
    android:padding="16dp">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to My App"
        android:textSize="18sp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintBottom_toTopOf="@id/editTextName"
        app:layout_constraintHorizontal_bias="0.5"/>

    <EditText
        android:id="@+id/editTextName"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="Enter text"
        android:padding="8dp"
        app:layout_constraintTop_toBottomOf="@id/textView"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintBottom_toTopOf="@id/checkBoxAgree"
        app:layout_constraintVertical_bias="0.2"/>

    <CheckBox
        android:id="@+id/checkBoxAgree"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I agree to the terms and conditions"
        app:layout_constraintTop_toBottomOf="@id/editTextName"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintBottom_toTopOf="@id/buttonSubmit"
        app:layout_constraintVertical_bias="0.5"/>

    <Button
        android:id="@+id/buttonSubmit"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Submit"
        app:layout_constraintTop_toBottomOf="@id/checkBoxAgree"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintVertical_bias="0.7"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Output



Explanation

- **ConstraintLayout:** The root layout that arranges child views using constraints.
- **TextView:** Positioned at the top center of the parent using 'app:layout_constraintTop_toTopOf', 'app:layout_constraintStart_toStartOf', and 'app:layout_constraintEnd_toEndOf'. The horizontal bias is set to '0.5' to center it.
- **EditText:** Positioned below the 'TextView' with its width constrained to the parent's width using '0dp' and 'layout_constraintWidth_default="match"'. The vertical bias is set to '0.2' to provide some space above it.
- **CheckBox:** Positioned below the 'EditText' and centered horizontally with a vertical bias of '0.5'.
- **Button:** Positioned below the 'CheckBox', centered horizontally, and above the bottom of the parent using 'app:layout_constraintBottom_toBottomOf'. The vertical bias is set to '0.7' to align it properly.

Advantages of Constraint Layout

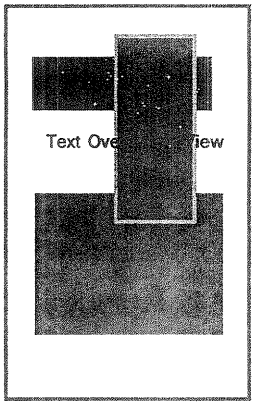
- **Flexibility:** Allows for complex positioning and alignment of views.
- **Performance:** Maintains a flat view hierarchy, which can improve performance.
- **Responsiveness:** Easily adapts to different screen sizes and orientations.
- **Ease of Use:** The visual editor in Android Studio makes it easy to create and manage constraints.

Disadvantages of Constraint Layout

- **Learning Curve:** Can be complex and difficult to learn for beginners.
- **Overhead:** Might add unnecessary complexity for simple layouts.
- **Performance:** While it can improve performance in some cases, improper use of constraints can lead to performance issues.

4.3.3.4 Frame Layout

FrameLayout is one of the simplest and most straightforward layout managers in Android. It is designed to block out an area on the screen to display a single item. FrameLayout is commonly used to hold a single child view, but it can also be used to stack multiple child views, with each new child view added appearing on top of the previous ones. It is like placing one transparent sheet over another, where only the topmost sheet is visible.



What is a Frame Layout ?

FrameLayout in Android is a simple layout that acts as a placeholder to display a single view. It is commonly used when we want to stack views on top of each other or when we need to display a single view at a time.

	Example	Understanding Frame Layout

Imagine different items placed on a shelf, such as books, a vase, and a photo frame. Each item added to the shelf stacks on top of the previous one, similar to arranging objects on a bookshelf. Adding a new item covers the items below it, creating a layered effect. This is how FrameLayout functions in Android. It allow views to be stacked on top of each other to create visually appealing layouts.

Key Features

1. **Simple Overlay:**
   Ideal for displaying a single child or overlapping multiple views.
2. **Stacking Views:**
   Allows multiple views to be stacked on top of each other.
3. **Z-Index Control:**
   The order in which views are added determines their layering, with the last added view on top.
4. **Minimal Attributes:**
   Simplifies layout configuration with fewer attributes compared to other layouts.
5. **Flexible Content:**
   Useful for dynamic content where one view can replace another, such as a progress bar over an image.

Attributes

Attribute	Description	Example
android:layout_gravity	Controls the alignment of the view within the FrameLayout	android:layout_gravity="center"
android:foreground	Sets a drawable to be drawn in front of the children	android:foreground="@drawable/overlay"
android:foregroundGravity	Controls the alignment of the foreground drawable	android:foregroundGravity="center"
android:measureAllChildren	Measures all children when true, regardless of visibility	android:measureAllChildren="true"

When to Choose FrameLayout

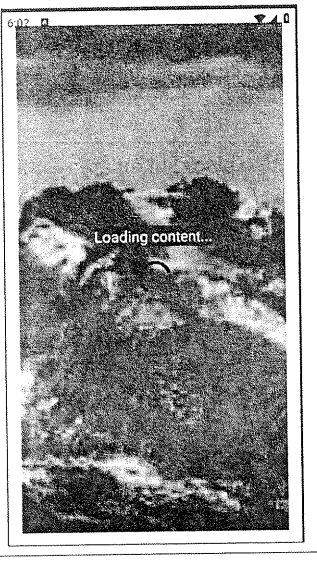
- When a simple layout is needed to block out an area for a single child.
- When overlapping multiple views is required, such as adding text over an image.
- When a lightweight and straightforward layout container is desired.
- When dynamically switching between views within the same container.

Example FrameLayout

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <!-- Background ImageView -->
    <ImageView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:src="@drawable/sample_image"
        android:scaleType="centerCrop"/>

    <!-- TextView Overlay -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Loading content..."
        android:textSize="18sp"
        android:textColor="#FFFFFF"
        android:layout_gravity="center"
        android:layout_marginBottom="50dp"
        android:background="#80000000"/><!-- semi-transparent background -->
```

```
    <!-- Centered ProgressBar -->
    <ProgressBar
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"/>

</FrameLayout>
```

Output



Explanation

- **FrameLayout:**The root layout that serves as a container for overlapping views.
- **ImageView:** Serves as the background image.

  'android:layout_width="match_parent"': The width of the ImageView matches the parent's width.

  'android:layout_height="match_parent"': The height of the ImageView matches the parent's height.

  'android:src="@drawable/sample_image"': Sets the image source.

  'android:scaleType="centerCrop"': Scales the image to fill the view, cropping it as necessary.
- **TextView:** Displays a loading message over the ImageView.

  'android:layout_width="wrap_content"': The width of the TextView is just enough to fit its content.

  'android:layout_height="wrap_content"': The height of the TextView is just enough to fit its content.

  'android:text="Loading content..."': Sets the text to be displayed.

  'android:textSize="18sp"': Sets the size of the text.

  'android:textColor="#FFFFFF"': Sets the color of the text to white for better visibility.

  'android:layout_gravity="center"': Centers the TextView within the FrameLayout.

'android:layout_marginBottom="50dp"': Adds a margin below the TextView to create space between the TextView and the ProgressBar.

'android:background="#80000000"': Sets a semi-transparent black background to make the text more readable against the image.

- **ProgressBar:** Indicates that content is loading, centered over both the ImageView and TextView.

'android:layout_width="wrap_content"': The width of the ProgressBar is just enough to fit its content.

'android:layout_height="wrap_content"': The height of the ProgressBar is just enough to fit its content.

'android:layout_gravity="center"': Centers the ProgressBar within the FrameLayout.

Advantages of FrameLayout

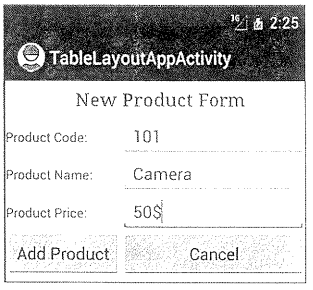
- **Simplicity:** Easy to use and understand, making it ideal for simple overlays.
- **Performance:** Minimal layout overhead, which can improve performance for simple use cases.
- **Versatility:** Can be used to stack multiple views and dynamically switch between them.
- **Dynamic Content:** Suitable for dynamic content updates, such as loading indicators or progress bars over other views.

Disadvantages of FrameLayout

- **Limited Layout Control:** Less control over positioning compared to other layouts like RelativeLayout or ConstraintLayout.
- **Overlapping Views:** Can become difficult to manage when multiple overlapping views are used.
- **No Relative Positioning:** Does not support relative positioning of child views without additional attributes or nesting.

4.3.3.5 Table Layout

TableLayout in Android is a layout manager that arranges views in rows and columns, similar to an HTML table structure. It is used to create grids of rows and columns to organize content in a structured format.



What is a Table Layout ?

A TableLayout is used to create a grid-like structure where views are organized in rows and columns. Each child of a TableLayout is a 'TableRow' that can contain multiple child views, representing the cells of the table. This layout is particularly useful for displaying tabular data.

Example	Understanding Table Layout
Imagine different items placed on a table, such as plates, cups, and utensils. Each item is arranged in a specific spot, like placing objects in a grid on a table. Each row on the table can hold multiple items, and each item fits neatly into its own cell without overlapping others. This is how TableLayout functions in Android. It arranges views in rows and columns, creating a structured, grid-like layout where each view occupies a distinct cell, making it ideal for displaying tabular data.	

Key Features

1. **Row and Column Organization:**
   Views are arranged in a grid of rows and columns, providing a structured layout for displaying data.
2. **Flexibility in Cell Spanning:**
   Cells can span multiple columns, allowing for more complex layouts.
3. **No Borders:**
   TableLayout does not display borders for its rows and cells by default, creating a clean and borderless table.
4. **Layout Weight:**
   Supports layout weight to distribute space among cells within a row.
5. **Column Shrinking and Stretching:**
   Columns can be marked as shrinkable or stretchable to adapt to the available space.

Attributes

Attribute	Description	Example
android:stretchColumns	Specifies which columns can stretch to fill the remaining space	android:stretchColumns="1,2"
android:shrinkColumns	Specifies which columns can shrink to fit the table's width	android:shrinkColumns="1"
android:collapseColumns	Specifies which columns can be collapsed	android:collapseColumns="1"
android:layout_span	Specifies how many columns a cell should span	android:layout_span="2"
android:layout_column	Specifies the column index for a view within a row	android:layout_column="1"
android:layout_gravity	Specifies how a view should be aligned within its cell	android:layout_gravity="center"

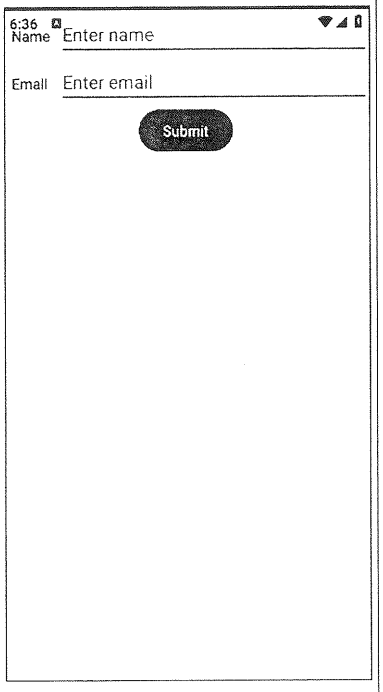
When to Choose TableLayout

- When a structured, grid-like layout is needed to display tabular data.
- When cells need to span multiple columns.
- When a simple way to create rows and columns without using nested layouts is required.
- When creating forms or data-entry interfaces that need a grid structure.

Example TableLayout

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1">

    <TableRow>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Name"
            android:padding="8dp"/>
        <EditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Enter name"/>
    </TableRow>

    <TableRow>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Email"
            android:padding="8dp"/>
        <EditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Enter email"/>
    </TableRow>

    <TableRow>
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Submit"
            android:layout_span="2"
            android:layout_gravity="center"/>
    </TableRow>

</TableLayout>
```

Output



Explanation

- **TableLayout:** The root layout that arranges its children into rows and columns.

  'android:stretchColumns="1"': The second column stretches to fill any remaining space.

- **TableRow:** Each row within the table layout.

Contains child views representing the cells of the table.

1. **First TableRow:**
   - **TextView:** Displays the label "Name".

     'android:padding="8dp"': Adds padding around the text.
   - **EditText:** An input field for entering the name.

     'android:hint="Enter name"': Provides a hint to the user.
2. **Second TableRow:**
   - **TextView:** Displays the label "Email".

     'android:padding="8dp"': Adds padding around the text.
   - **EditText:** An input field for entering the email.

     'android:hint="Enter email"': Provides a hint to the user.
3. **Third TableRow:**
   - **Button:** A submit button.

     'android:layout_span="2"': Spans the button across two columns.

     'android:layout_gravity="center"': Centers the button within the row.

Advantages of TableLayout

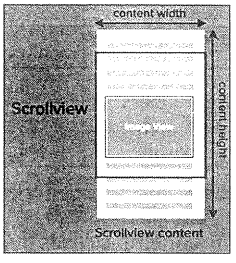
- **Structured Layout:** Provides a clean and structured way to arrange views in rows and columns.
- **Ease of Use:** Simplifies the creation of forms and tables without the need for nested layouts.
- **Cell Spanning:** Supports spanning across multiple columns for more complex layouts.
- **Flexible Columns:** Allows columns to stretch, shrink, or collapse based on the available space.

Disadvantages of TableLayout

- **Performance:** Can become inefficient with a large number of rows and columns, as it measures each child twice.
- **Complexity:** Managing a large and complex table layout can become cumbersome.
- **No Borders:** Does not provide borders by default, requiring additional views or drawables to add lines and borders.

4.3.3.6 Scroll View

ScrollView is a layout container that allows for vertical scrolling of its content. It is particularly useful when the content to be displayed exceeds the available screen space to ensure that users can still access all the information by scrolling up and down.



What is a Scroll View ?

ScrollView in Android is a layout manager that allows its content to be scrolled vertically when it exceeds the screen size. It is used to display a larger amount of content within a limited screen space by enabling scrolling functionality.

Example	Understanding Scroll View

Imagine different items placed on a long shelf, such as books, photo frames, and decorative items. Due to the length of the shelf, all items cannot be viewed at once, and one needs to slide or scroll horizontally to see the entire collection. Each item is arranged linearly, and the shelf ensures that even if the items exceed the visible area, they remain accessible by scrolling. This is how ScrollView functions in Android. It allows views to be placed inside a scrollable container, ensuring that even if the content exceeds the screen's height, it remains accessible through vertical scrolling, making it ideal for displaying lengthy or vertically extensive content.

Key Features

1. **Vertical Scrolling:**
   Enables scrolling vertically through content that exceeds the screen height.
2. **Nested Layouts:**
   Can contain a single layout with multiple views, allowing for complex scrollable interfaces.
3. **Smooth Scrolling:**
   Provides smooth scrolling behavior to enhance user experience.
4. **Scroll Indicators:**
   Automatically shows scroll indicators when content is scrollable.
5. **Programmatic Control:**
   Allows for programmatically controlling scrolling positions and detecting scroll events.

Attributes

Attribute	Description	Example
android:fillViewport	Forces the ScrollView to expand its content to fill the viewport	android:fillViewport="true"

android:scrollbars	Specifies whether scrollbars should be displayed	android:scrollbars="vertical"
android:overScrollMode	Defines the over-scroll mode (e.g., always, if content scrolls, never)	android:overScrollMode="always"
android:fadeScrollbars	Controls whether the scrollbars should fade out	android:fadeScrollbars="true"
android:scrollbarSize	Sets the width of the scrollbars	android:scrollbarSize="5dp"

When to Choose ScrollView

- When the content exceeds the available screen height and vertical scrolling is needed.
- When creating layouts that require displaying a large amount of data or components.
- When ensuring accessibility to off-screen content.
- When a simple and user-friendly way to navigate through extended content is required.

Example ScrollView

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:fillViewport="true">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">

        <ImageView
            android:layout_width="match_parent"
            android:layout_height="500dp"
            android:src="@drawable/sample_image"
            android:scaleType="centerCrop"
            android:contentDescription="Sample Image"/>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="This is some sample text."
            android:padding="16dp"/>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Another piece of text."
            android:padding="16dp"/>
```

```
        <Button
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Click Me"
            android:padding="16dp"/>

        <Button
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Another Button"
            android:padding="16dp"/>

    </LinearLayout>
</ScrollView>
```

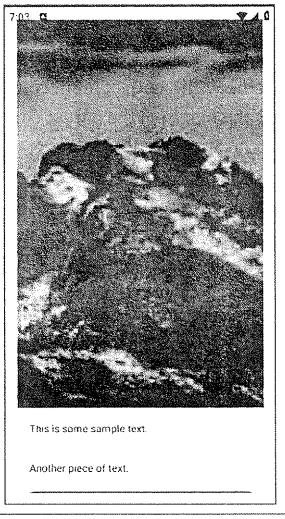
Explanation

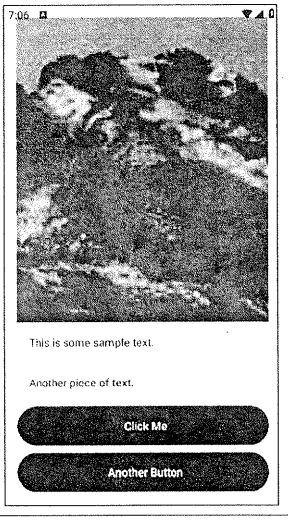
- **ScrollView:** The root layout that enables vertical scrolling.
  - 'android:layout_width="match_parent"': Matches the width of the parent container.
  - 'android:layout_height="match_parent"': Matches the height of the parent container.
  - 'android:padding="16dp"': Adds padding around the ScrollView content.
  - 'android:fillViewport="true"': Forces the ScrollView to expand its content to fill the viewport.
- **LinearLayout:** A single child layout within the ScrollView containing multiple views.
  - 'android:layout_width="match_parent"': Matches the width of the parent container.
  - 'android:layout_height="wrap_content"': The height is determined by the content.
  - 'android:orientation="vertical"': Arranges the child views vertically.

1. **ImageView:** Displays an image at the top.
   - 'android:layout_width="match_parent"': The width of the ImageView matches the parent's width.
   - 'android:layout_height="200dp"': Sets a fixed height for the ImageView.
   - 'android:src="@drawable/sample_image"': Sets the image source.
   - 'android:scaleType="centerCrop"': Scales the image to fill the view, cropping it as necessary.
   - 'android:contentDescription="Sample Image"': Provides a description for accessibility.
2. **TextView:** Displays some sample text.
   - 'android:layout_width="match_parent"': The width of the TextView matches the parent's width.
   - 'ndroid:layout_height="wrap_content"': The height is determined by the content.
   - 'android:text="This is some sample text."': Sets the text to be displayed.
   - 'android:padding="16dp"': Adds padding around the text for better spacing.
3. **TextView:** Displays another piece of text.
   - 'android:layout_width="match_parent"': The width of the TextView matches the parent's width.
   - 'android:layout_height="wrap_content"': The height is determined by the content.
   - 'android:text="Another piece of text."': Sets the text to be displayed.
   - 'android:padding="16dp"': Adds padding around the text for better spacing.

4. **Button: Adds a button for interaction.**
   - 'android:layout_width="match_parent"': The width of the Button matches the parent's width.
   - 'android:layout_height="wrap_content"': The height is determined by the content.
   - 'android:text="Click Me"': Sets the text to be displayed on the button.
   - 'android:padding="16dp"': Adds padding inside the button for better visual appearance.
5. **Button: Adds another button for interaction.**
   - 'android:layout_width="match_parent"': The width of the Button matches the parent's width.
   - 'android:layout_height="wrap_content"': The height is determined by the content.
   - 'android:text="Another Button"': Sets the text to be displayed on the button.
   - 'android:padding="16dp"': Adds padding inside the button for better visual appearance.

Output





Advantages of ScrollView

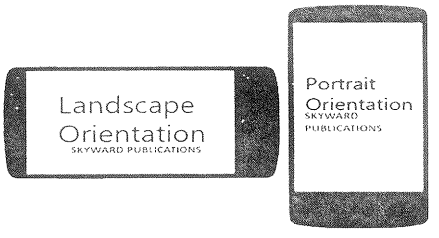
- **Simple Implementation:** Easy to implement and use for adding vertical scrolling to content.
- **Flexible Content:** Can contain complex layouts and a large number of child views.
- **User Experience:** Enhances user experience by providing smooth scrolling for overflow content.
- **Compatibility:** Widely supported across different Android versions.

Disadvantages of ScrollView

- **Single Child Restriction:** Can only contain one direct child, which must be a layout containing other views.
- **Performance:** Performance can degrade if the content inside is too complex or large.
- **Horizontal Scrolling Limitation:** ScrollView is primarily designed for vertical scrolling. For horizontal scrolling, a HorizontalScrollView is needed.

4.4 Adapting to Display Orientation

Adapting to display orientation is crucial in app development to cater to different devices with varying screen sizes and orientations. It ensures that the user interface remains user-friendly regardless of how the device is held. The two main orientations are **portrait** and **landscape**. The portrait and landscape orientations offer distinct layouts and apps should seamlessly transition between them to provide a consistent user experience.



Example : In the YouTube app, when a device is held in portrait mode, the video playback is optimized for a vertical viewing experience, with the video filling most of the screen height. Upon rotating the device to landscape mode, the video automatically adjusts to fit the wider screen to provide a more immersive viewing experience with the video expanding to utilize the additional horizontal space.

What is Display Orientation ?

Display orientation refers to the way content is presented on the screen of a device. The two main orientations are:

- **Portrait Mode:** The display is taller than it is wide. This is the default orientation for most smartphones and is commonly used for reading, browsing, and using most apps.
- **Landscape Mode:** The display is wider than it is tall. This orientation is often used for watching videos, playing games, and using certain types of apps like photo editors or spreadsheet programs.

Techniques and Methods to Adapt Display Orientation:

When an Android device changes orientation, the current activity is recreated and the onCreate() method is called again. This can lead to views being redrawn in their original positions, which may not be optimal for the new orientation. To handle these changes effectively, we can use several techniques. To implement views that adapt to display orientation changes in Android, developers can utilize various techniques and methods:

1. Anchoring Views
2. Resizing and Repositioning Views
3. Creating Separate Layouts for Different Orientations

4.4.1 Anchoring Views

Anchoring views is a layout design technique used to ensure that UI elements maintain their position relative to specific edges of the screen, regardless of the device's orientation. This approach helps create a consistent user experience by keeping important controls, such as buttons and input fields, in predictable locations.

When anchoring views, we need to specify rules or constraints that dictate how a view should be positioned relative to its parent layout or other views. This technique is commonly used to keep elements aligned to specific edges, corners, or positions on the screen to ensure a consistent layout across different screen sizes and orientations.

What is Anchoring Views ?

Anchoring views in Android layout design refers to the technique of fixing the position of UI elements relative to specific edges or positions on the screen. By anchoring views, we can ensure that certain elements remain in a consistent location regardless of changes in screen orientation to provide a stable and predictable layout for the user interface.

How It Works ?

In Android XML layouts, anchoring views is typically achieved using a RelativeLayout, which allows to define the position of views relative to the parent layout or other sibling views. By setting the below attributes, we can anchor views to specific edges of the screen.

Anchoring Attributes:

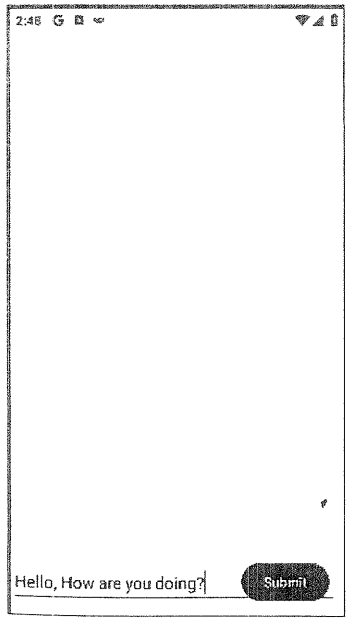
1. **'android:layout_alignParentBottom':** This attribute positions the view at the bottom edge of its parent layout. When set to "true", the view will be anchored to the bottom of the parent layout to ensure it remains at the bottom regardless of other views or screen orientation changes.
2. **'android:layout_alignParentLeft':** Setting this attribute to "true" aligns the view with the left edge of its parent layout. The view will be anchored to the left side of the parent, maintaining its position relative to the left edge even if other views are added or the screen orientation changes.
3. **'android:layout_alignParentRight':** Similar to 'layout_alignParentLeft', this attribute aligns the view with the right edge of its parent layout. By setting it to "true", the view will be anchored to the right side of the parent to ensure its position remains fixed relative to the right edge.
4. **'android:layout_alignParentTop':** When this attribute is set to "true", the view is aligned with the top edge of its parent layout. Anchoring the view to the top ensures it stays at the top of the parent layout, maintaining its position regardless of other elements or orientation adjustments.

Example | Anchoring Views

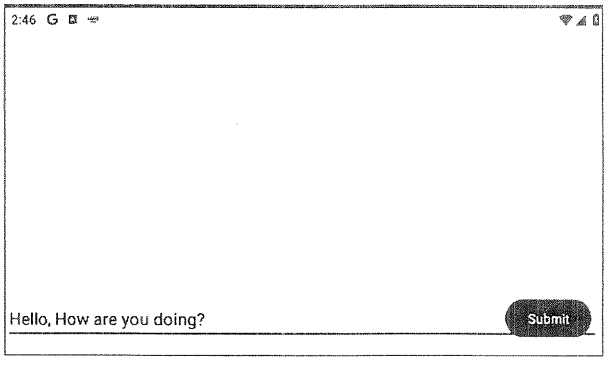
```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <EditText
        android:id="@+id/edit_text"
        android:layout_width="0dp"
```

```
        android:layout_height="wrap_content"
        android:hint="Type something"
        android:layout_alignParentBottom="true"
        android:layout_toStartOf="@id/button"
        android:layout_marginEnd="8dp"
        android:layout_marginBottom="16dp"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"/>

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Submit"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:layout_margin="16dp" />
</RelativeLayout>
```

Output



Portrait Mode



Landscape Mode

Explanation

1. **EditText:** The EditText is anchored to the bottom edge of the parent using android:layout_alignParentBottom="true". It is also constrained to the left and right edges of the parent and to the left of the Button using android:layout_toStartOf="@id/button".
2. **Button:** The Button is anchored to the bottom right edge of the parent using android:layout_alignParentBottom="true" and android:layout_alignParentRight="true".

This anchoring ensures that both the EditText and Button remain at the bottom edge of the screen, regardless of whether the device is in portrait or landscape orientation. This approach provides a consistent and predictable layout for users.

4.4.2 Resizing and Repositioning Views

Resizing and repositioning views is a technique used in Android development to dynamically adjust the size and position of UI elements based on the current screen orientation or available screen space. This ensures that the layout remains functional and aesthetically pleasing regardless of how the user holds the device.

What is Resizing and Repositioning Views ?

Resizing refers to changing the dimensions of a view (width and height), while repositioning involves changing its location on the screen. These adjustments are often necessary to make the best use of the available screen space and to ensure that all important elements are visible and accessible.

How It Works ?

Resizing and repositioning views typically involve using layout attributes that respond to changes in screen size and orientation. Layout managers such as 'ConstraintLayout' or 'LinearLayout' with weights can be used to define flexible, responsive layouts. These layouts can adjust the dimensions and positions of views dynamically.

The below example uses 'ConstraintLayout' to demonstrate resizing and repositioning views. This example will adjust the size and position of an 'EditText' and a 'Button' to make better use of the available screen space in different orientations.

Code Resizing and Repositioning Views

```
<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"
    android:padding="16dp">

    <EditText
        android:id="@+id/username"
        android:layout_width="0dp"
```

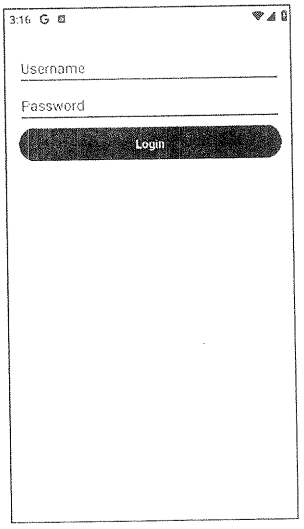
```
        android:layout_height="wrap_content"
        android:hint="Username"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginTop="32dp"
        android:layout_marginBottom="16dp"/>

    <EditText
        android:id="@+id/password"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="Password"
        android:inputType="textPassword"
        app:layout_constraintTop_toBottomOf="@id/username"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginBottom="16dp"/>

    <Button
        android:id="@+id/login_button"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Login"
        app:layout_constraintTop_toBottomOf="@id/password"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Explanation

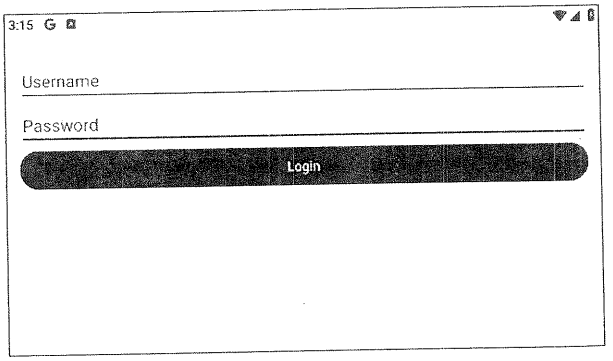
1. **ConstraintLayout:**
   - **Padding:** Added padding to ensure content isn't flush with the screen edges.
   - **Width & Height:** Set to 'match_parent' to fill the entire screen.
2. **EditText for Username:**
   - **Width:** Set to '0dp' to allow dynamic resizing based on constraints.
   - **Height:** Set to 'wrap_content' to fit the content.
   - **Constraints:**
     - 'app:layout_constraintTop_toTopOf="parent"': Anchors the top of the 'EditText' to the top of the parent.
     - 'app:layout_constraintStart_toStartOf="parent"' and 'app:layout_constraintEnd_toEndOf="parent"': Ensures the 'EditText' stretches across the width of the parent.
   - **Margins:** Adds top and bottom margins for spacing.

3. **EditText for Password:**
   - **Width:** Set to '0dp' to allow dynamic resizing based on constraints.
   - **Height:** Set to 'wrap_content' to fit the content.
   - **Constraints:**
     - 'app:layout_constraintTop_toBottomOf="@id/username"': Anchors the top of the 'EditText' to the bottom of the username 'EditText'.
     - 'app:layout_constraintStart_toStartOf="parent"' and 'app:layout_constraintEnd_toEndOf="parent"': Ensures the 'EditText' stretches across the width of the parent.
   - **Margins:** Adds a bottom margin for spacing.
4. **Button for Login:**
   - **Width:** Set to '0dp' to allow dynamic resizing based on constraints.
   - **Height:** Set to 'wrap_content' to fit the content.
   - **Constraints:**
     - 'app:layout_constraintTop_toBottomOf="@id/password"': Anchors the top of the 'Button' to the bottom of the password 'EditText'.
     - 'app:layout_constraintStart_toStartOf="parent"' and 'app:layout_constraintEnd_toEndOf="parent"': Ensures the 'Button' stretches across the width of the parent.
5. **How It Works:**
   - **Portrait Mode:** The layout is vertically aligned, with the username and password fields and login button stretching across the width of the screen. This ensures a clean, centered appearance.
   - **Landscape Mode:** The same constraints ensure that the views stretch horizontally to use the available space efficiently. The vertical alignment is maintained to provide a consistent user experience.

Output



Portrait Mode



Landscape Mode

4.4.3 Creating Separate Layouts for Different Orientations

When a device changes orientation, the screen dimensions and aspect ratio change. A layout designed for portrait mode may not look or function optimally in landscape mode and vice versa. To address this, Android allows to define different layout resources for different orientations.

What do you mean by Creating Separate Layouts for Different Orientation ?

Creating separate layouts for different orientations is a technique used in Android development to provide customized user interfaces for portrait and landscape modes. By designing distinct layouts for each orientation, we can can optimize the UI for the available screen space to ensure a better user experience.

How It Works:

Android's resource system supports providing alternative resources for different device configurations, including screen orientation. By placing layout files in specific resource directories (res/layout for portrait and res/layout-land for landscape), the system automatically loads the appropriate layout based on the current orientation.

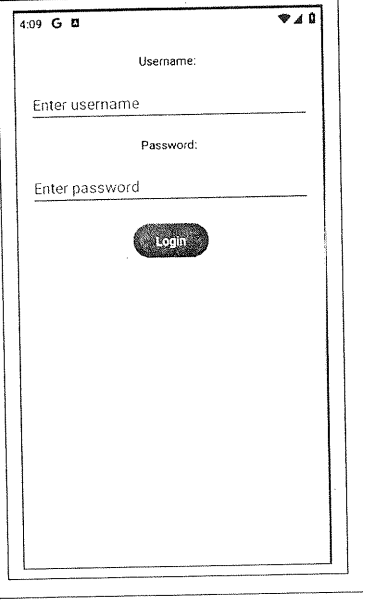
1. **Create Layout Files:** Separate XML layout files for each orientation. For example, have activity_main.xml for portrait mode and activity_main_land.xml for landscape mode.
2. **Organize Layout Resources:** Place portrait layout files in the **res/layout** directory and landscape layout files in the **res/layout-land** directory. Android loads the appropriate layout based on device orientation.
3. **Customize Layouts:** Customize XML files to suit portrait and landscape orientations. Adjust element positions, sizes, and alignments as needed.

Code Portrait Layout (res/layout/activity_main.xml). This layout file will be used when the device is in portrait mode.

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <TextView
        android:id="@+id/username_label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Username:"
        android:layout_marginTop="32dp"
        android:layout_centerHorizontal="true"/>
```

```
    <EditText
        android:id="@+id/username"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter username"
        android:layout_below="@id/username_label"
        android:layout_margirTop="16dp"/>

    <TextView
        android:id="@+id/password_label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Password:'
        android:layout_below="@id/username"
        android:layout_marginTop="16dp"
        android:layout_cente^Horizontal="true"/>

    <EditText
        android:id="@+id/password"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter password"
        android:inputType="textPassword"
        android:layout_below="@id/password_label"
        android:layout_marginTop="16dp"/>

    <Button
        android:id="@+id/login_button"
        android:layout_width="wrap_content"
        android:layout_height='wrap_content"
        android:text="Login"
        android:layout_below="@id/password"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="16dp"/>
</RelativeLayout>
```

Output



Explanation

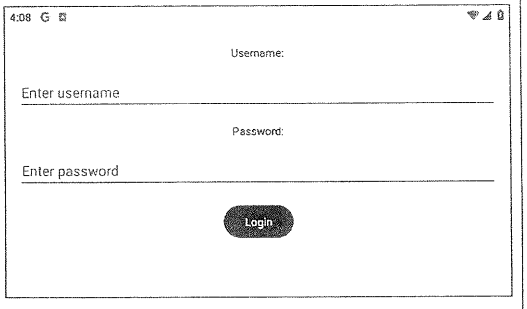
- **RelativeLayout**: Used as the root layout to easily position elements relative to each other.
- **TextView and EditText for Username:** Positioned in the center horizontally, with appropriate margins for spacing.
- **TextView and EditText for Password**: Placed below the username fields, also centered horizontally.
- **Button:** Positioned below the password fields, centered horizontally.

Code Landscape Layout (res/layout-land/activity_main.xml). This layout file will be used when the device is in landscape mode.

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <TextView
        android:id="@+id/username_label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Username:"
        android:layout_marginStart="16dp"
        android:layout_alignParentStart="true"
        android:layout_centerVertical="true"/>

    <EditText
        android:id="@+id/username"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="Enter username"
        android:layout_toEndOf="@id/username_label"
        android:layout_centerVertical="true"
        android:layout_marginStart="8dp"
        android:layout_marginEnd="16dp"
        android:layout_weight="1"/>

    <TextView
        android:id="@+id/password_label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Password:"
        android:layout_below="@id/username"
        android:layout_marginStart="16dp"
        android:layout_alignParentStart="true"
        android:layout_centerVertical="true"/>

    <EditText
        android:id="@+id/password"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
```

```
        android:hint="Enter password"
        android:inputType="textPassword"
        android:layout_toEndOf="@id/password_label"
        android:layout_centerVertical="true"
        android:layout_marginStart="8dp"
        android:layout_marginEnd="16dp"
        android:layout_weight="1"/>

    <Button
        android:id="@+id/login_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Login"
        android:layout_below="@id/password"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="16dp"/>
</RelativeLayout>
```

Output



Explanation

- **RelativeLayout:** Used as the root layout to easily position elements relative to each other.
- **TextView and EditText for Username:** Positioned horizontally next to each other. The 'EditText' is given a '0dp' width and a 'layout_weight' to ensure it stretches to fill the available space.
- **TextView and EditText for Password:** Positioned below the username fields, aligned similarly.
- **Button:** Positioned below the password fields, centered horizontally.

How It Works

1. **Resource Directories:**
   - **res/layout:** Contains the default layout file ('activity_main.xml'), which is used for portrait mode.
   - **res/layout-land:** Contains the landscape-specific layout file ('activity_main.xml'), which is used when the device is in landscape mode.
2. **Automatic Layout Loading:**
   - When the device is in portrait mode, Android loads the layout from 'res/layout/activity_main.xml'.

- When the device switches to landscape mode, Android automatically loads the layout from 'res/layout-land/activity_main.xml'.

3. **Consistent User Experience:**
   - By defining separate layouts for each orientation, the UI elements are arranged optimally for the available screen space. This ensures that the user experience remains consistent and user-friendly regardless of the device's orientation.

Note: Testing Orientation Changes in Android Applications

1. **Android Emulator (Android Studio):**
   - **Open the Project:** Open the Android project in Android Studio.
   - **Run the Emulator:** Click the "Run" button or press 'Shift + F10' to start the application in the emulator.
   - **Rotate the Screen:** Use the rotate buttons in the emulator toolbar:
     - Click the 'Rotate Left' button to rotate the device to the left (landscape mode).
     - Click the 'Rotate Right' button to rotate the device to the right (landscape mode).
2. **Physical Device:**
   - **Using the Device Settings:**
     1. **Enable Auto-Rotate:** Ensure that the auto-rotate feature is enabled on the device. This can usually be found in the quick settings panel or in the display settings.
     2. **Rotate the Device:** Physically rotate the device to switch between portrait and landscape modes.
   - **Using Developer Options:**
     1. **Enable Developer Options:** Go to 'Settings > About phone > Software information' and tap on the "Build number" multiple times until a message appears stating "You are now a developer."
     2. **Developer Options:** Navigate to 'Settings > Developer options' and look for the option to force activities to be resizable. This helps in testing how the app behaves with configuration changes.
3. **In External Emulator - LDPlayer9:**
   - **Enable Forcibly Lock Landscape:**

     To see the output in landscape mode:
     1. **Open LDPlayer:** Start LDPlayer and open the application to be tested.
     2. **Go to Settings:** Click on the settings icon in the toolbar (usually looks like a gear or wrench).
     3. **Navigate to Other Settings Tab:** In the settings window, go to the "Other Settings" tab.
     4. **Enable Forcibly Lock Landscape:** Check the box or toggle the switch for "Forcibly Lock Landscape". Restart the LDPlayer9 and then execute the app from Android Studio.

     The emulator will switch to landscape mode and lock the orientation, allowing observation of the application's behavior in landscape.

- **Disable Forcibly Lock Landscape:**

  To see the output back in portrait mode:

  1. **Open LDPlayer:** If not already open, start LDPlayer and open the application to be tested.
  2. **Go to Settings:** Click on the settings icon in the toolbar.
  3. **Navigate to Other Settings Tab:** In the settings window, go to the "Other Settings" tab.
  4. **Disable Forcibly Lock Landscape:** Uncheck the box or toggle the switch for "Forcibly Lock Landscape". Restart the LDPlayer9 and then execute the app from Android Studio.

  The emulator will switch back to portrait mode, allowing observation of the application's behavior in portrait.

4.5 Managing Changes To Screen Orientation

We have studied how to implement the different techniques for adapting to screen orientation changes. Let's explore what happens to an activity's state when the device changes orientation.

When a device changes orientation (from portrait to landscape or vice versa), the current activity undergoes a configuration change. This involves the activity being destroyed and recreated to adapt to the new configuration. Understanding what happens to an activity's state during this process is crucial for maintaining a seamless user experience.

4.5.1 What Happens to an Activity's State When Orientation Changes?

When a device changes orientation, the activity state undergoes several significant changes. Understanding these changes is crucial for maintaining a seamless user experience.

1. **Activity Destruction and Recreation:**
   - When the device orientation changes, Android destroys the current activity and creates a new instance of it. This process is essential to reload resources that match the new configuration, such as different layout files optimized for landscape or portrait mode.
   - During this process, the 'onCreate()' method is called again, setting up the new activity instance. This ensures that the activity is correctly configured for the new orientation, but it also means that the activity starts from scratch, losing any transient UI state unless it's explicitly saved and restored.

   Example : Imagine an email application where a user is composing an email. If the user changes the device orientation while typing, the activity is destroyed and recreated. Without proper handling, the user might lose the text they were typing, which can be frustrating and disruptive.

2. **Loss of Transient State:**
   - Transient UI states, such as text entered in text fields, selected items in lists, and other temporary data, can be lost during the destruction process. This happens because the default behavior does not automatically save this transient state.
   - Any unsaved data is discarded when the activity is destroyed. The new activity instance created after the orientation change will not have access to the previous state unless mechanisms are in place to save and restore it.

Example: Consider a shopping app where a user is filling out a checkout form. If the user rotates the device mid-process, all the information entered into the form fields could be lost, requiring the user to start over. This could lead to user frustration and potentially abandonment of the checkout process.

3. **Persisting State:**

   To preserve the activity state across configuration changes, Android provides several mechanisms:

   - **onSaveInstanceState() Method:** This method allows the activity to save its current UI state into a Bundle before it is destroyed. When the activity is recreated, the saved state can be restored from the Bundle.
   - **onRestoreInstanceState() Method:** This method can be used to restore the state from the Bundle that was saved in onSaveInstanceState(). It provides an opportunity to restore the UI state after the onCreate() method has completed.

   Example: In a note-taking app, a user may be writing a note. If the orientation changes, the app should save the current note's text using the 'onSaveInstanceState()' method. When the activity is recreated, the app retrieves the text from the 'Bundle' and restores it in the text field.

4.5.2 Persisting State Information during Changes in Configuration

When an activity is destroyed and recreated due to configuration changes, such as screen orientation changes, it may lose its current state. Understanding the lifecycle methods involved in preserving and restoring the state is crucial for maintaining a seamless user experience.

Lifecycle Methods Involved

1. **onPause():** Always called when an activity is killed or pushed into the background. Use this method to save state using different mechanisms like storing in databases or file storage.
2. **onSaveInstanceState():** Called before an activity is killed or pushed into the background. This method is not called when the activity is removed from the stack (e.g., when the back button is pressed). It allows saving state in a 'Bundle' object, which is later restored in 'onCreate()' or 'onRestoreInstanceState()'.

Multiple Approaches to Preserving State

When handling configuration changes, there are multiple approaches to preserving state in an Android application:

- **onSaveInstanceState():** Simple to implement for basic data types but limited to 'Bundle' storage.
- **onRetainNonConfigurationInstance():** Allows saving more complex data types and objects but is deprecated as of API level 28.
- **ViewModel:** The recommended approach for modern Android development. It provides a robust and lifecycle-aware solution for managing UI-related data across configuration changes.

1. Using onSaveInstanceState()

1. **Saving State:**

The 'onSaveInstanceState()' method is used to save the activity's state to a 'Bundle'.

Code

```
@Override
public void onSaveInstanceState(Bundle outState) {
    // Save state data into the Bundle
    outState.putString("ID", "1234567890");
    super.onSaveInstanceState(outState);
}
```

2. **Restoring State:**

The 'onCreate()' method is called first when the activity is recreated, followed by 'onRestoreInstanceState()', which retrieves the saved state from the 'Bundle'.

Code

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

    if (savedInstanceState != null) {
        // Retrieve and use the saved state data
        String ID = savedInstanceState.getString("ID");
    }
}


@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    // Retrieve and use the saved state data
    String ID = savedInstanceState.getString("ID");
}
```

Limitation: The 'onSaveInstanceState()' method only allows saving data in a 'Bundle', which is suitable for simple data types but not for complex data structures.

2. Using onRetainNonConfigurationInstance()

1. **Saving State:**

The 'onRetainNonConfigurationInstance()' method allows saving more complex data when the activity is about to be destroyed due to configuration changes.

Code

```
@Override
public Object onRetainNonConfigurationInstance() {
    // Save state data and return it
    return "Some text to preserve";
}
```

2. **Restoring State:**

The saved data can be retrieved in the 'onCreate()' method using 'getLastNonConfigurationInstance()'.

Code

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

    // Retrieve the saved data
    String str = (String) getLastNonConfigurationInstance();
    if (str != null) {
        // Use the saved data
        Log.d("StateInfo", "Restored data: " + str);
    }
}
```

Example Use Case: When downloading data from a web service, if the user changes the screen orientation, using 'onRetainNonConfigurationInstance()' to temporarily save the downloaded data can prevent the need to re-download the data, making the process more efficient.

3. Using ViewModel

'ViewModel' is part of Android's Architecture Components and is designed to store and manage UI-related data in a lifecycle-conscious way. It survives configuration changes such as screen rotations, ensuring that data persists even if the activity is recreated.

Step 1: Enable dataBinding in build.gradle.kts (Module:app) File

Ensure the buildFeatures entry is added to the 'build.gradle.kts' file:

Code

```
android {
        -----------
        buildFeatures  {
             dataBinding = true
        }
        ----------
}
```

Step 2: Create a ViewModel Class:

Create a 'ViewModel' class to store the UI-related data.

Code

```
public class MyViewModel extends ViewModel {
    private MutableLiveData<String> userText;

    public LiveData<String> getUserText() {
        if (userText == null) {
            userText = new MutableLiveData<>();
        }
        return userText;
    }

    public void setUserText(String text) {
        if (userText == null) {
            userText = new MutableLiveData<>();
        }
        userText.setValue(text);
    }
}
```

Step 3: Use ViewModel in Activity:

In the activity, use the 'ViewModel' to manage UI data.

Code

```
public class MainActivity extends AppCompatActivity {
    private EditText editText;
    private MyViewModel viewModel;

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

        editText = findViewById(R.id.edit_text);
        viewModel = new ViewModelProvider(this).get(MyViewModel.class);

        // Observe changes to the ViewModel
        viewModel.getUserText().observe(this, new Observer<String>() {
            @Override
            public void onChanged(@Nullable String s) {
                editText.setText(s);
            }
        });
```

```
        editText.addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(CharSequence s, int start,
                                          int count, int after) { }

            @Override
            public void onTextChanged(CharSequence s, int start,
                                      int before, int count) {
                viewModel.setUserText(s.toString());
            }

            @Override
            public void afterTextChanged(Editable s) { }
        });
    }
}
```

Explanation

- **ViewModelProvider:** Used to obtain the 'ViewModel' instance. The same 'ViewModel' instance is returned even after configuration changes.
- **LiveData:** The 'ViewModel' uses 'LiveData' to hold the data. 'LiveData' is lifecycle-aware and ensures that UI updates are only made when the activity or fragment is in an appropriate state.
- **TextWatcher:** Updates the 'ViewModel' whenever the text in the 'EditText' changes. This ensures that the latest data is always stored in the 'ViewModel'.

4.5.3 Detecting Orientation Changes

Sometimes, it is necessary to know the device's current orientation during runtime. This can be achieved using the 'WindowManager' class. The following code demonstrates how to programmatically detect the current orientation of an activity:

Step by Step Implemention of Detecting Orientation Changes

Step 1: Create a New Project:

Create an Android project by selecting "Empty Views Activity", naming the project "OrientationMessageApp", and choosing Java as the language.

Step 2: Set Up the Main Activity Layout:

Edit the 'activity_main.xml' file to include a 'TextView' that will display the orientation message.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">
```

```
    <TextView
        android:id="@+id/orientation_message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="24sp"
        android:layout_centerInParent="true"/>
</RelativeLayout>
```

Explanation

- **'RelativeLayout'**: Used as the root layout, allowing child views to be positioned relative to each other.
- **'TextView'**: Added with the ID 'orientation_message' and centered within the layout. This 'TextView' will display the orientation message.

Step 3: Modify the MainActivity Java File:

Edit the 'MainActivity.java' file to detect the orientation and set the text of the 'TextView' accordingly using the 'Point' class to get the display size.

Code MainActivity.java

```
package com.example.orientationchange;

import android.graphics.Point;
import android.os.Bundle;
import android.view.Display;
import android.view.WindowManager;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        TextView orientationMessage = findViewById(R.id.orientation_message);

        //---get the current display info---
        WindowManager wm = (WindowManager) getSystemService(WINDOW_SERVICE);
        Display d = wm.getDefaultDisplay();
        Point size = new Point();
        d.getSize(size);
        int width = size.x;
        int height = size.y;
        if (width > height) {
            //---landscape mode---
            orientationMessage.setText("Landscape mode");
```

```
        } else {
            //---portrait mode---
            orientationMessage.setText("Portrait mode");
        }
    }
}
```

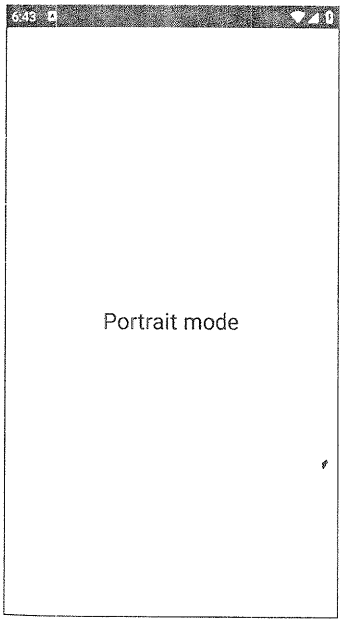
Explanation

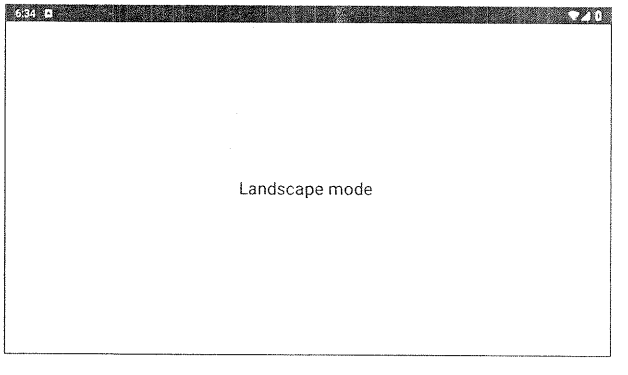
- The 'onCreate' method sets the content view to 'activity_main.xml'.
- A 'TextView' is referenced using 'findViewById'.
- The 'WindowManager' is used to get the current display information.
- The 'Point' class is used to store the display dimensions.
- The width and height of the display are compared using the 'x' and 'y' properties of the 'Point' object to determine the orientation.
- Based on the orientation, the 'TextView' displays either "Landscape mode" or "Portrait mode".

Output

When the application runs, it checks the current screen orientation. If the device is in landscape mode, the 'TextView' will display "Landscape mode". If the device is in portrait mode, the 'TextView' will display "Portrait mode". The orientation message will update each time the activity is created, ensuring the correct message is displayed based on the current orientation.

This app has been tested on LDPlayer 9 by setting "Forcibly Lock Landscape" enabled for landscape mode to ensure proper functionality.





4.5.4 Controlling the Orientation of the Activity

In certain scenarios, it is necessary to ensure that an application is displayed in a specific orientation, such as landscape mode for a game or portrait mode for a reading app. This can be achieved programmatically or by setting attributes in the Android manifest file.

Controlling the orientation of an activity can be crucial for certain applications. This can be done programmatically using the 'setRequestedOrientation()' method or by setting the 'android:screenOrientation' attribute in the manifest file. These methods prevent the activity from being destroyed and recreated when the device orientation changes, thus maintaining the current state and ensuring a seamless user experience.

Programmatically Forcing Orientation Using setRequestedOrientation() Method

The 'setRequestedOrientation()' method of the 'Activity' class allows changing the orientation programmatically.

Code MainActivity.java

```
import android.content.pm.ActivityInfo;
import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        //---change to landscape mode---
        setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
    }
}
```

Explanation

- 'setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE)': Forces the activity to switch to landscape mode.
- To switch to portrait mode, use 'ActivityInfo.SCREEN_ORIENTATION_PORTRAIT'.

Setting Orientation in AndroidManifest.xml

The 'android:screenOrientation' attribute in the '<activity>' element of the manifest file can be used to constrain the activity to a specific orientation.

Code AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="net.learn2develop.Orientations"
    android:versionCode="1"
    android:versionName="1.0">

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name">

        <activity
            android:label="@string/app_name"
            android:name=".OrientationsActivity"
            android:screenOrientation="landscape">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Explanation

- 'android:screenOrientation="landscape"': Constrains the activity to landscape mode.
- Other possible values for 'android:screenOrientation':
- 'portrait': Constrains the activity to portrait mode.
- 'sensor': The orientation is determined based on the accelerometer (default behavior).

4.6 Utilizing The Action Bar

The ActionBar is a primary feature of the modern Android user interface that provides a dedicated space for essential app functionalities. Introduced in Android 3.0 (API level 11), it has become a central UI component for enhancing user experience by providing quick access to important actions, navigation, and other key elements.

What is an Action Bar in Android ?

The ActionBar in Android is a user interface component that typically appears at the top of an app's screen and provides consistent navigation, branding, and actions for the user. The Action Bar in Android is a dedicated space at the top of the screen that provides a consistent navigation and access point for key features and actions within an app. It typically contains the app icon and title, navigation tabs, action icons, and overflow menu. The ActionBar helps users to access key features easily and maintain a consistent user experience across different screens and activities.

Key Concepts and Components of Action Bar

The ActionBar in Android typically consists of the following components:

1. **App Icon:** The application icon is displayed on the left side of the ActionBar. It helps users identify the app they are currently using.
2. **App Title:** The title of the current screen or activity is usually displayed next to the app icon. It provides context to the user about the content they are viewing.
3. **Navigation Tabs:** Tabs can be included in the ActionBar to allow users to switch between different sections or views within the app. Each tab represents a specific category or functionality. Tabs allow users to switch between related content easily.
4. **Action Icons:** Action icons are used to provide quick access to common actions or features within the app. These icons can represent actions like search, settings, refresh, or any other relevant actions.
5. **Overflow Menu:** The overflow menu is a three-dot icon placed at the right end of the Action Bar. It contains additional actions or options that do not fit directly on the Action Bar due to limited space. Users can access these options by tapping on the overflow icon.

Example | **Understanding ActionBar in WhatsApp**

The below image showcases an Action Bar from a WhatsApp messaging application. Let us understand each element in the Action Bar:

1. **Back Button (Left Arrow):**

   The back arrow icon on the far left allows the user to navigate back to the previous screen. This is typically used to exit the current chat and return to the main chat list.

2. **Profile Picture:**

   Next to the back button, there is a small circular profile picture of the contact or group. This visually indicates whom the chat is with.

3. **Contact Name and Status:**
   - **Contact Name:** The name "Manju Sln" displayed prominently indicates the contact or group name.
   - **Status:** Below the contact name, there is a status message "last seen today at 11:58," showing the last seen or the current status of the contact.

4. **Video Call Icon:**

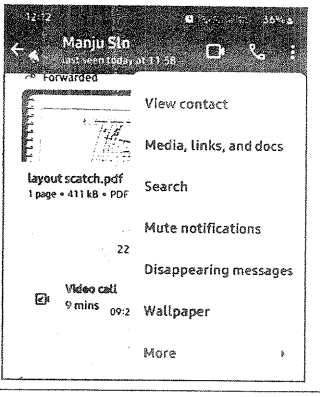
   The video camera icon is used to initiate a video call with the contact. Tapping this icon starts a video call directly from the chat screen.

5. **Voice Call Icon:**

   The phone receiver icon allows the user to make a voice call to the contact. This provides quick access to voice calling functionality.

6. **Overflow Menu (Three Vertical Dots):**

   The three vertical dots represent the overflow menu. Tapping this icon opens a dropdown menu with additional actions and settings related to the chat. This typically includes options such as viewing contact info, select messages, close chat, mute notifications, disappearing messages, and other chat settings.



4.6.1 Creating an ActionBar

The following example explains how to create a simple ActionBar in an Android application. This ActionBar will display a title and a subtitle but will not include action items or a menu.

Step 1: Create a Project:

A new project named SimpleActionBar is created using empty view activity template and Java as a languge.

Step 2: Update 'AndroidManifest.xml':

Modify the 'AndroidManifest.xml' to include the theme setting under the '<application>' tag.

Replace the existing theme in the AndroidManifest.xml file with below

`android:theme="@style/Theme.AppCompat.Light.NoActionBar".`

This ensures that the application uses a custom Toolbar instead of the default ActionBar.

Step 3: Update 'activity_main.xml':

Open 'activity_main.xml' and add the following code to define the layout, which includes a custom Toolbar and a TextView. The Toolbar is used as the ActionBar, and the TextView is added to display a message in the center of the screen.

Code 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"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#168BC34A"
    tools:context=".MainActivity">
```

```
    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        android:elevation="4dp"
        android:theme="@style/ThemeOverlay.AppCompat.ActionBar"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light"
        tools:ignore="MissingConstraints" />

    <!-- Adding a TextView in the activity -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to E-Commerce App"
        android:textColor="#000000"
        android:textSize="24sp"
        android:textStyle="bold"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Explanation

- **'<androidx.appcompat.widget.Toolbar>':** This tag defines the custom Toolbar.
- **'android:layout_width="match_parent"':** The Toolbar spans the entire width of the screen.
- **'android:layout_height="?attr/actionBarSize"':** The height of the Toolbar is set to the standard ActionBar size.
- **'android:background="?attr/colorPrimary"':** The background color of the Toolbar is set to the primary color defined in the theme.
- **'android:theme="@style/ThemeOverlay.AppCompat.ActionBar"':** This line ensures the Toolbar inherits ActionBar styling.
- **'app:popupTheme="@style/ThemeOverlay.AppCompat.Light"':** This sets the theme for the overflow menu.
- **'tools:ignore="MissingConstraints"':** This suppresses warnings about missing constraints in the layout editor.

A TextView is added to display a welcome message " Welcome to E-Commerce App"

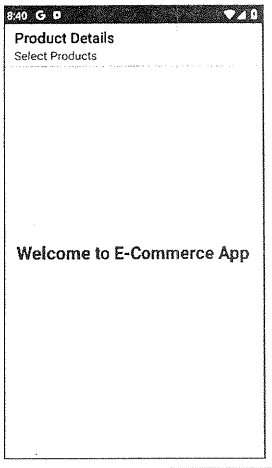
Step 4: Update 'MainActivity.java':

Open 'MainActivity.java' and update the 'onCreate' method to set the custom Toolbar as the ActionBar. This involves finding the Toolbar by its ID and setting it as the ActionBar using 'setSupportActionBar'. Additionally, the title, subtitle, and home display options are configured.

Code MainActivity.java

```
package com.example.simpleactionbar;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;

public class MainActivity extends AppCompatActivity {

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

        // Find the toolbar by its ID
        Toolbar toolbar = findViewById(R.id.toolbar);

        // Set the Toolbar as the ActionBar
        setSupportActionBar(toolbar);

        // Ensure the ActionBar is not null
        if (getSupportActionBar() != null) {
            // Set the title of the ActionBar
            getSupportActionBar().setTitle("Product Details");

            // Set the subtitle of the ActionBar
            getSupportActionBar().setSubtitle("Select Products");
        }
    }
}
```

Output



Explanation

- **'Toolbar toolbar = findViewById(R.id.toolbar);'**: This line finds the Toolbar defined in 'activity_main.xml'.
- **'setSupportActionBar(toolbar);'**: This sets the Toolbar as the ActionBar for the activity.
- **'getSupportActionBar().setTitle("Product Details");'**: This sets the title of the ActionBar to "Product Details".
- **'getSupportActionBar().setSubtitle("Select Products");'**: This sets the subtitle of the ActionBar to "Select Products".

4.6.2 Adding Action Items to the Action Bar

To add action items to the ActionBar in an Android application, follow these detailed steps.

Note : Ensure that Step 1, Step 2 and Step3 of the Previous Section are completed.

Step 1: Creating the Menu Folder:

The menu items for the ActionBar are defined in XML files that reside in the 'menu' directory. This directory should be created under the 'res' directory of the Android project.

Steps:

- Right-click on the 'res' directory.
- Select "New" > "Directory".
- Name the directory 'menu'.

Step 2: Creating 'menu_main.xml' File:

The 'menu_main.xml' file is where the menu items are defined. Each item in this file represents an action button in the ActionBar.

1. **Attributes:**
   - **'android:id':** A unique identifier for the menu item.
   - **'android:icon':** Specifies the icon for the menu item.
   - **'android:title':** The text displayed for the menu item.
   - **'app:showAsAction':** Specifies when and how this item should appear as an action item in the ActionBar.
2. **Common values include:**
   - **'always':** The item always appears as an action item.
   - **'ifRoom':** The item appears as an action item if there is room.
   - **'never':** The item never appears as an action item and is instead in the overflow menu.
   - **'withText':** The item displays both its icon and text.

Step 3: Creating Icons:

Icons for the action items can be created in a couple of ways:

1. **Using Image Asset Studio:**
   - Right-click the 'res' directory.
   - Select "New" > "Image Asset".
   - Follow the prompts to create icons for an action items. This tool helps generate icons of different sizes and densities.
2. **Downloading from Google:**
   - Download the desired icons from internet.
   - Place the downloaded icons in the 'res/drawable' directory.

Step 4: Defining Menu Items in 'menu_main.xml':

Once the icons are ready, they can be referenced in the 'menu_main.xml' file.

Code menu_main.xml

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

    <!-- action button for search -->
    <item android:title="Search"
        android:id="@+id/search"
        android:orderInCategory="100"
        app:showAsAction="ifRoom"
        android:icon="@drawable/search_icon"/>

    <!-- action button for refresh -->
    <item android:title="Refresh"
        android:id="@+id/refresh"
        android:orderInCategory="100"
        app:showAsAction="ifRoom"
        android:icon="@drawable/refresh_icon"/>

    <!-- action button for add to cart -->
    <item android:title="Add to Cart"
        android:id="@+id/cart"
        android:orderInCategory="100"
        app:showAsAction="never"/>

    <!-- action button for settings -->
    <item android:title="Settings"
        android:id="@+id/settings"
        android:orderInCategory="100"
        app:showAsAction="never"/>
</menu>
```

Explanation

- **'xmlns:app="http://schemas.android.com/apk/res-auto"':** This line defines the XML namespace for custom attributes, such as 'app:showAsAction'.
- **'android:title="Search"':** This sets the title of the menu item to "Search".
- **'android:id="@+id/search"':** This gives the menu item a unique identifier.
- **'android:orderInCategory="100"':** This sets the order in which the menu item appears.
- **'app:showAsAction="ifRoom"':** This specifies that the item should appear as an action item if there is room.
- **'android:icon="@drawable/ic_search"':** This sets the icon for the menu item.
- we have used serach_icon and refresh_icon in XML.

Step 5: Implementing Action Items in the Activity:

To make the menu items functional, two methods need to be overridden in the activity:

- **'onCreateOptionsMenu(Menu menu)':**

  This method inflates the menu items defined in the 'menu_main.xml' file into the ActionBar.

- **'onOptionsItemSelected(MenuItem item)':**

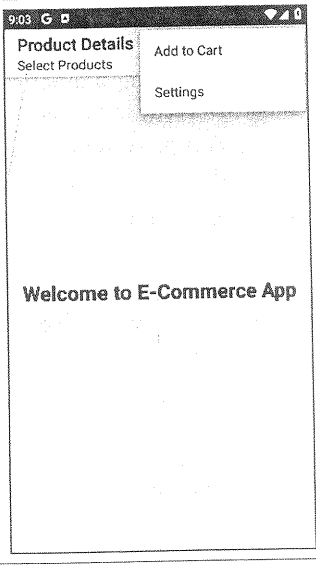
  This method handles click events for the menu items. It checks the ID of the clicked item and performs the corresponding action.

Code MainActivity.java

```
package com.example.simpleactionbar;

import androidx.annotation.NonNull;
import androidx.appcompat.app.ActionBar;
import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;

import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

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

        // Find the toolbar by its ID
        Toolbar toolbar = findViewById(R.id.toolbar);

        // Set the Toolbar as the ActionBar
        setSupportActionBar(toolbar);
        if (getSupportActionBar() != null) {
            // Set the title of the ActionBar
            getSupportActionBar().setTitle("Product Details");
            // Set the subtitle of the ActionBar
            getSupportActionBar().setSubtitle("Select Products");
        }
    }

    // Method to inflate the options menu when the user opens the menu for the first time
    @Override
    public boolean onCreateOptionsMenu( Menu menu ) {
```

```
        // Inflate the menu items from menu_main.xml
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return super.onCreateOptionsMenu(menu);
    }

    // Methods to control the operations that will happen when user clicks on the action buttons
    @Override
    public boolean onOptionsItemSelected(@NonNull MenuItem item) {
        int itemId = item.getItemId();
        if (itemId == R.id.search) {
            Toast.makeText(this, "Search Clicked", Toast.LENGTH_SHORT).show();
            return true;
        } else if (itemId == R.id.refresh) {
            Toast.makeText(this, "Refresh Clicked", Toast.LENGTH_SHORT).show();
            return true;
        } else if (itemId == R.id.cart) {
            Toast.makeText(this, "Add to Cart Clicked", Toast.LENGTH_SHORT).show();
            return true;
        } else if (itemId == R.id.settings) {
            Toast.makeText(this, "Settings Clicked", Toast.LENGTH_SHORT).show();
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
}
```

Output



Explanation

- 'Toolbar toolbar = findViewById(R.id.toolbar);': Finds the Toolbar defined in 'activity_main.xml'.
- 'setSupportActionBar(toolbar);': Sets the Toolbar as the ActionBar for the activity.
- 'getSupportActionBar().setTitle("Product Details");': Sets the title of the ActionBar.
- 'getSupportActionBar().setSubtitle("Select Products");': Sets the subtitle of the ActionBar.
- 'onCreateOptionsMenu(Menu menu)': getMenuInflater().inflate(R.menu.menu_main, menu);': Inflates the menu items from 'menu_main.xml' into the ActionBar.
- 'onOptionsItemSelected(@NonNull MenuItem item)':
- 'int itemId = item.getItemId();': Gets the ID of the clicked menu item.
- 'if (itemId == R.id.search)': Checks if the "Search" menu item was clicked.
- 'Toast.makeText(this, "Search Clicked", Toast.LENGTH_SHORT).show();': Displays a toast message when the "Search" menu item is clicked.
- The same logic is applied for the "Refresh", "Add to Cart", and "Settings" menu items.

4.7 Creating The User Interface Programmatically

So far, all the UIs seen in this chapter are created using XML. Besides using XML, the UI can also be created using code. This approach is useful if the UI needs to be dynamically generated during runtime. For example, consider building a cinema ticket reservation system where the application displays the seats of each cinema using buttons. In this case, the UI would need to be dynamically generated based on the cinema selected by the user.

Creating the UI programmatically involves defining and configuring view objects in the activity's Java instead of using XML layout files. This approach provides greater flexibility, particularly for UIs that need to adapt to changing data or conditions at runtime.

Key concepts:

1. **Creating Views Programmatically:** Views such as buttons, text views, and layouts can be instantiated directly in the code.
2. **Setting Layout Parameters:** Each view needs layout parameters to define its position and size within the parent layout.
3. **Adding Views to Layouts:** Once created and configured, views are added to parent layout containers (like LinearLayout, RelativeLayout, etc.).

Example : Creating a Simple Login Page Programmatically

This example explains how to create a simple login page programmatically in Android. The login page will include two 'EditText' fields (for username and password) and a 'Button' (for submitting the login form).

Step 1: Create a Project:

Create an Android project by selecting "Empty Views Activity", naming the project "DynamicLogin", and choosing Java as the language.

Delete the 'activity_main.xml' file as it is not required for programmatically creating the UI.

Step 2: Write the Code in 'MainActivity.java'

In this step, the UI components will be created and configured programmatically in the 'MainActivity.java' file.

Code MainActivity.java

```
package com.example.dynamiclogin;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.LinearLayout;
import android.widget.Toast;

import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Create a LinearLayout
        LinearLayout linearLayout = new LinearLayout(this);
        linearLayout.setOrientation(LinearLayout.VERTICAL);
        linearLayout.setLayoutParams(new LinearLayout.LayoutParams(
                LinearLayout.LayoutParams.MATCH_PARENT,
                LinearLayout.LayoutParams.MATCH_PARENT));
        linearLayout.setPadding(32, 32, 32, 32);

        // Create EditText for username
        EditText usernameEditText = new EditText(this);
        usernameEditText.setLayoutParams(new LinearLayout.LayoutParams(
                LinearLayout.LayoutParams.MATCH_PARENT,
                LinearLayout.LayoutParams.WRAP_CONTENT));
        usernameEditText.setHint("Enter Username");

        // Create EditText for password
        EditText passwordEditText = new EditText(this);
        passwordEditText.setLayoutParams(new LinearLayout.LayoutParams(
                LinearLayout.LayoutParams.MATCH_PARENT,
                LinearLayout.LayoutParams.WRAP_CONTENT));
        passwordEditText.setHint("Enter Password");
        passwordEditText.setInputType(android.text.InputType.TYPE_TEXT_VARIATION_PASSWORD);
```

```
        // Create Button for login
        Button loginButton = new Button(this);
        loginButton.setLayoutParams(new LinearLayout.LayoutParams(
                LinearLayout.LayoutParams.MATCH_PARENT,
                LinearLayout.LayoutParams.WRAP_CONTENT));
        loginButton.setText("Login");

        // Set button click listener
        loginButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String username = usernameEditText.getText().toString();
                String password = passwordEditText.getText().toString();
                Toast.makeText(MainActivity.this, "Username: " + username +
                        "\nPassword: " + password, Toast.LENGTH_SHORT).show();
            }
        });

        // Add views to the LinearLayout
        linearLayout.addView(usernameEditText);
        linearLayout.addView(passwordEditText);
        linearLayout.addView(loginButton);

        // Set the LinearLayout as the content view
        setContentView(linearLayout);
    }
}
```

Explanation

1. **Creating a LinearLayout:**
   - A 'LinearLayout' is instantiated and configured to be the main layout container for the activity.
   - 'linearLayout.setOrientation(LinearLayout.VERTICAL)': Sets the orientation of the 'LinearLayout' to vertical, stacking child views vertically.
   - 'linearLayout.setLayoutParams(...)': Sets the layout parameters to match the parent dimensions, ensuring the 'LinearLayout' fills the entire screen.
   - 'linearLayout.setPadding(...)': Adds padding to the 'LinearLayout' to provide some spacing around the edges.
2. **Creating 'EditText' for Username:**
   - An 'EditText' is created for the username input.
   - 'usernameEditText.setLayoutParams(...)': Sets the layout parameters to match the parent width and wrap content height.

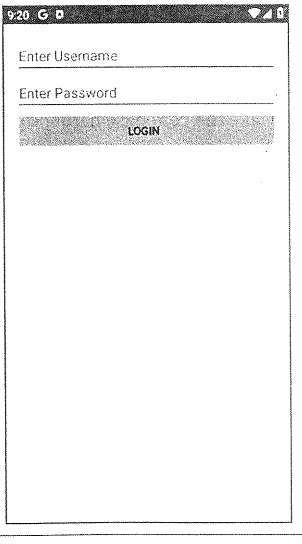
- 'usernameEditText.setHint("Enter Username")': Sets the hint text to guide the user on what to enter.

3. **Creating 'EditText' for Password:**
   - Another 'EditText' is created for the password input.
   - Similar layout parameters are set as for the username.
   - 'passwordEditText.setHint("Enter Password")': Sets the hint text for the password input.
   - 'passwordEditText.setInputType(android.text.InputType.TYPE_TEXT_VARIATION_PASSWORD)': Configures the input type to hide the password text.
4. **Creating 'Button' for Login:**
   - A 'Button' is created for submitting the login form.
   - 'loginButton.setLayoutParams(...)': Sets the layout parameters to match the parent width and wrap content height.
   - 'loginButton.setText("Login")': Sets the text on the button.
5. **Setting Button Click Listener:**
   - An 'OnClickListener' is set on the button to handle click events.
   - Inside the 'onClick' method, the text from the username and password fields is retrieved and displayed in a 'Toast' message.
6. **Adding Views to the LinearLayout:**

   The 'EditText' views and 'Button' are added to the 'LinearLayout' using the 'addView' method.
7. **Setting the LinearLayout as the Content View:**

   Finally, the 'LinearLayout' is set as the content view for the activity using 'setContentView(linearLayout)'.

Output



Advantages of Creating The User Interface Programmatically

1. **Dynamic UI Generation:** Programmatically creating the UI allows for dynamic generation of views based on runtime conditions or user interactions. This flexibility is beneficial for apps with complex or changing UI requirements.
2. **Customization:** Programmatic UI creation provides greater control over the appearance and behavior of UI elements. Developers can customize layouts, styles, and interactions with precision.
3. **Performance Optimization:** In some cases, creating the UI programmatically can lead to better performance compared to XML layouts, especially when dealing with complex UI structures or dynamic content loading.
4. **Modularity and Reusability:** Programmatic UI creation can facilitate modularity and reusability of UI components. Developers can create custom views or layouts that can be easily reused across different parts of the app.
5. **Dynamic Theming:** With programmatic UI creation, it is easier to implement dynamic theming or theming based on user preferences. Views can be styled programmatically based on user-selected themes or preferences.

Disadvantages of of Creating The User Interface Programmatically

1. **Complexity and Readability:** Code for programmatic UI creation can become complex and harder to read, especially for large and intricate layouts. Maintaining and debugging such code may require more effort..
2. **Increased Development Time:** Writing UI code programmatically can be more time-consuming compared to using XML layouts, especially for simple UI designs. It may involve writing more lines of code to achieve the same layout
3. **Limited Visual Editing:** Unlike XML layouts that offer a visual editor in Android Studio, programmatic UI creation lacks a visual representation during design time. Developers need to rely on running the app to see the UI output.
4. **Less Separation of Concerns:** Mixing UI logic with business logic in the code can lead to a lack of separation of concerns. This can make the codebase harder to maintain and scale as the app grows.
5. **Compatibility and Fragmentation:** Programmatic UI creation may require handling different screen sizes, densities, and orientations programmatically, which can be more challenging compared to using XML layouts that offer built-in support for these variations.

4.8 Listening For UI Notifications

Listening for UI notifications involves monitoring and responding to user interactions with the application's user interface. This can occur at different levels within an application, such as the activity level and the view level. By listening for these notifications, an application can provide feedback, perform actions, and generally improve the user experience based on the user's actions.

- **Activity Level Examples:**
  - **Detecting Key Presses:** Handling key events such as volume buttons or back button presses.
  - **Managing Menu Items:** Responding to selections from the app's options menu.

- **Handling Configuration Changes:** Adapting to changes in device orientation or screen size.
- **Intercepting Touch Events:** Customizing touch event handling across the entire activity.

▲ View Level Examples:

- **Button Click:** Submitting a form when a button is pressed.
- **Dropdown Selection:** Performing an action when an item is selected from a dropdown menu.
- **Text Input:** Validating input as the user types into a text field.
- **Item Click in a ListView:** Navigating to a detailed view when an item in a list is clicked.

4.8.1 Listening for UI Notifications at the Activity Level

Users interact with the UI in Android applications at two levels: the activity level and the view level. At the activity level, the Activity class provides various methods that can be overridden to handle specific events and user interactions. These methods allow the application to respond to high-level events such as key presses, menu selections, lifecycle changes, touch events, and window focus changes.

The 'Activity' class in Android provides several methods that can be overridden to handle user interactions at the activity level. These methods include 'onKeyDown', 'onKeyUp', 'onMenuItemSelected', 'onMenuOpened', 'onTouchEvent', 'onWindowFocusChanged', 'onConfigurationChanged', 'onBackPressed', 'onSaveInstanceState', 'onRestoreInstanceState', 'onCreateOptionsMenu', and 'onOptionsItemSelected'. By overriding these methods, an application can respond to key presses, menu selections, touch events, window focus changes, configuration changes, and back key presses, providing a responsive and interactive user experience.

Some common methods that can be overridden in activities are listed below:

Common Methods

1. **onKeyDown:**

   The 'onKeyDown' method is called when a key is pressed down. This method can be overridden to handle key press events that are not already handled by any views within the activity. It provides an opportunity to respond to key events such as the volume keys, back key, or other hardware buttons.

	Example	When the volume down key is pressed during a voice message recording, the app could lower the volume.

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_VOLUME_DOWN) {
        // Code to lower the volume
        return true;  // Event handled
    }
    return super.onKeyDown(keyCode, event);  // Pass other keys to the super class
}
```

Explanation

- Checks if the pressed key is the volume down key.
- Executes code to lower the volume.
- Returns 'true' to indicate that the event has been handled.

2. **onKeyUp:**

The 'onKeyUp' method is called when a key is released. Similar to 'onKeyDown', this method can be overridden to handle key release events that are not already handled by any views within the activity. It provides an opportunity to respond to key release events such as the volume keys, back key, or other hardware buttons.

Example When the volume up key is released during a voice call, the app could increase the volume.

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_VOLUME_UP) {
        // Code to increase the volume
        return true;  // Event handled
    }
    return super.onKeyUp(keyCode, event);  // Pass other keys to the super class
}
```

Explanation

- Checks if the released key is the volume up key.
- Executes code to increase the volume.
- Returns 'true' to indicate that the event has been handled.

3. **onMenuItemSelected:**

The 'onMenuItemSelected' method is called when a menu item in a panel has been selected by the user. This method allows the activity to respond to menu item selections, such as opening a new activity or performing a specific action based on the selected menu item.

Example When the user selects the "Settings" option from the menu.

```
@Override
public boolean onMenuItemSelected(int featureId, MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_settings:
            // Code to open settings
            return true;
        default:
            return super.orMenuItemSelected(featureId, item);
    }
}
```

Explanation

- Checks the ID of the selected menu item.
- Executes code to open settings.
- Returns 'true' to indicate that the event has been handled.

4. **onMenuOpened:**

The 'onMenuOpened' method is called when a panel's menu is opened by the user. This method allows the activity to perform actions or update the UI when the menu is opened, such as highlighting the current selection or updating the menu items.

Example When the user opens the main menu to access options like "New Group" or "Settings".

```
@Override
public boolean onMenuOpened(int featureId, Menu menu) {
    // Code to handle menu opened event
    return true;  // Event handled
}
```

Explanation

- Executes code to handle menu opened event.
- Returns 'true' to indicate that the event has been handled.

5. **onTouchEvent:**

The 'onTouchEvent' method is called when a touch screen motion event occurs. This method allows the activity to respond to touch events, such as taps, swipes, or other gestures. It provides an opportunity to handle touch interactions that are not already handled by any views within the activity.

Example When the user touches the screen to scroll through chat messages.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            // Code to handle touch down event
            return true;
        case MotionEvent.ACTION_UP:
            // Code to handle touch up event
            return true;
    }
    return super.onTouchEvent(event);  // Pass other events to the super class
}
```

Explanation

- Checks the action of the touch event.
- Executes code to handle touch down and touch up events.
- Returns 'true' to indicate that the event has been handled.

6. **onWindowFocusChanged:**

The 'onWindowFocusChanged' method is called when the current window of the activity gains or loses focus. This method allows the activity to respond to changes in window focus, such as updating the UI or performing specific actions when the window gains or loses focus.

Example	When switching between WhatsApp and another app.

```
@Override
public void onWindowFocusChanged(boolean hasFocus) {
    super.onWindowFocusChanged(hasFocus);
    if (hasFocus) {
        // Code to handle window gained focus
    } else {
        // Code to handle window lost focus
    }
}
```

Explanation

- Checks whether the window has gained or lost focus.
- Executes code to handle window focus changes.

7. **onConfigurationChanged:**

The 'onConfigurationChanged' method is called when the device configuration changes while the activity is running. This includes changes such as screen orientation, keyboard availability, and locale. This method allows the activity to respond to configuration changes without being restarted.

Example	When switching between portrait and landscape modes while viewing a chat.

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
        // Code to handle orientation change to landscape
    } else if (newConfig.orientation == Configuration.ORIENTATION_PORTRAIT) {
        // Code to handle orientation change to portrait
    }
}
```

Explanation

- Checks the new configuration's orientation.
- Executes code to handle orientation changes.

3. **onBackPressed**

The 'onBackPressed' method is called when the activity has detected the user's press of the back key. This method allows the activity to handle back press events, such as navigating to the previous screen or exiting the activity.

Example	When the user presses the back key to exit a chat and return to the main chat list.

```
@Override
public void onBackPressed() {
    // Code to handle back press event
    super.onBackPressed();
}
```

Explanation
• Executes code to handle back press event. • Calls 'super.onBackPressed()' to perform the default back press behavior.

9. **onSaveInstanceState:**

The 'onSaveInstanceState' method is called to save the current state of the activity. This method is invoked before the activity may be destroyed so that the state can be restored in 'onRestoreInstanceState'. It is commonly used to save UI state and user data.

Example	When rotating the device while composing a message, the current message text should be saved.

```
@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    EditText messageEditText = findViewById(R.id.messageEditText);
     outState.putString("draftMessage", messageEditText.getText().toString());
// Save the draft message
}
```

Explanation
Saves the current text in the message 'EditText' to the 'Bundle' so it can be restored later.

10. **onRestoreInstanceState:**

The 'onRestoreInstanceState' method is called after 'onCreate' when the activity is being re-initialized from a previously saved state, given here in 'savedInstanceState'. It is commonly used to restore UI state and user data.

Example	When the device is rotated while composing a message, the previously composed text should be restored.

```
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    EditText messageEditText = findViewById(R.id.messageEditText);
    String draftMessage = savedInstanceState.getString("draftMessage");
    if (draftMessage != null) {
        messageEditText.setText(draftMessage);  // Restore the draft message
    }
}
```

Explanation
Retrieves the saved message text from the 'Bundle' and sets it back to the 'EditText'.

11. **onCreateOptionsMenu**

The 'onCreateOptionsMenu' method is called to initialize the contents of the activity's standard options menu. This method is only called once, the first time the options menu is displayed. It allows the activity to inflate menu items from XML resources.

Example	When creating the main menu for options like "New Group", "New Broadcast", "WhatsApp Web", etc.

```
@Override
public boolean onCreateOpticnsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);  // Inflate the menu items
    return true;
}
```

Explanation

Inflates the menu items from 'menu_main.xml' into the options menu.

12. **onOptionsItemSelected:**

The 'onOptionsItemSelected' method is called whenever an item in the options menu is selected. This method allows the activity to handle menu item selections and perform actions based on the selected item.

Example	When the user selects "Settings" from the options menu.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.action_settings) {
        // Code to open settings
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Explanation

- Checks the ID of the selected menu item.
- Executes code to open settings.
- Returns 'true' to indicate that the event has been handled.

4.8.2 Registering Views for Notifications

At the view level, individual UI components (views) such as buttons, text fields, and other interactive elements can be registered for notifications. This involves attaching event listeners to these views to handle specific user actions.

Registering views for notifications involves attaching event listeners to UI components such as buttons, text fields, and other interactive elements. This allows the application to respond to specific user actions like clicks, long clicks, focus changes, touch events, key presses, editor actions, and changes in checked state. By handling these events, applications can provide a responsive and interactive user experience.

Common View Events

1. onClick:

The 'onClick' event is triggered when a view, such as a button, is clicked. This is one of the most common events and is used to perform an action in response to user input.

Example When the user clicks the send button to send a message.

Code XML Layout

```
<Button
    android:id="@+id/sendButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Send" />
```

Code Activity Java Code

```
Button sendButton = findViewById(R.id.sendButton);
sendButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // Code to send message
    }
});
```

Explanation

- Attaches an 'OnClickListener' to the send button.
- Executes code to send the message when the button is clicked.

2. onLongClick:

The 'onLongClick' event is triggered when a view is clicked and held for a longer duration. This is useful for providing additional options or actions.

Example When the user long-clicks a message to bring up options like delete, forward, or copy.

Code XML Layout:

```
<TextView
    android:id="@+id/messageTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, this is a message" />
```

Code Activity Java Code

```
TextView messageTextView = findViewById(R.id.messageTextView);
messageTextView.setOnLongClickListener(new View.OnLongClickListener() {
    @Override
    public boolean onLongClick(View v) {
        // Code to show options like delete, forward, or copy
        return true;  // Event handled
    }
});
```

Explanation

- Attaches an 'OnLongClickListener' to the message text view.
- Executes code to show options when the message is long-clicked.
- Returns 'true' to indicate that the event has been handled.

3. onFocusChange:

The 'onFocusChange' event is triggered when a view gains or loses focus. This can be useful for validating input or updating the UI based on focus changes.

Example When the user focuses on the text input field to start typing a message.

Code XML Layout

```
<EditText
    android:id="@+id/messageEditText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Type a message" />
```

Code Activity Java Code

```
EditText messageEditText = findViewById(R.id.messageEditText);
messageEditText.setOnFocusChangeListener(new View.OnFocusChangeListener() {
    @Override
    public void onFocusChange(View v, boolean hasFocus) {
        if (hasFocus) {
            // Code to handle focus gained
        } else {
            // Code to handle focus lost
        }
    }
});
```

Explanation

- Attaches an 'OnFocusChangeListener' to the message edit text.
- Executes code to handle when the edit text gains or loses focus.

4. **onTouch:**

The 'onTouch' event is triggered when a touch event occurs on a view. This event can be used to handle various touch interactions like tapping, swiping, or dragging.

Example When the user touches and drags a message to perform actions like swiping to reply.

Code XML Layout:

```
<TextView
    android:id="@+id/swipeMessageTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Swipe me" />
```

Code Activity Java Code:

```
TextView swipeMessageTextView = findViewById(R.id.swipeMessageTextView);
swipeMessageTextView.setOnTouchListener(new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                // Code to handle touch down
                return true;
            case MotionEvent.ACTION_MOVE:
                // Code to handle touch move
                return true;
            case MotionEvent.ACTION_UP:
                // Code to handle touch up
                return true;
        }
        return false;
    }
});
```

Explanation

- Attaches an 'OnTouchListener' to the swipe message text view.
- Executes code to handle touch down, move, and up events.
- Returns 'true' to indicate that the event has been handled.

5. onKey:

The 'onKey' event is triggered when a hardware key is pressed while the view has focus. This is useful for handling key events within a specific view.

Example When the user presses the enter key to send a message from the text input field.

Code XML Layout

```
<EditText
    android:id="@+id/messageEditText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Type a message"
    android:imeOptions="actionSend" />
```

Code Activity Java Code

```
EditText messageEditText = findViewById(R.id.messageEditText);
messageEditText.setOnKeyListener(new View.OnKeyListener() {
    @Override
    public boolean onKey(View v, int keyCode, KeyEvent event) {
       if (keyCode == KeyEvent.KEYCODE_ENTER && event.getAction()
                    == KeyEvent.ACTION_DOWN) {
            // Code to send message
            return true;
        }
        return false;
    }
});
```

Explanation

- Attaches an 'OnKeyListener' to the message edit text.
- Executes code to send the message when the enter key is pressed.
- Returns 'true' to indicate that the event has been handled.

6. onCheckedChanged:

The 'onCheckedChanged' event is triggered when the checked state of a compound button, such as a checkbox or radio button, changes. This is useful for handling changes in selection state.

Example When the user selects or deselects a contact for creating a group.

Code XML Layout

```
<CheckBox
    android:id="@+id/contactCheckBox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Contact Name" />
```

Code Activity Java Code

```
CheckBox contactCheckBox = findViewById(R.id.contactCheckBox);
contactCheckBox.setOnCheckedChangeListener(new CompoundButton.
OnCheckedChangeListener() {
    @Override
   public void onCheckedChanged(CompoundButton buttonView, boolean isChecked)
{
        if (isChecked) {
            // Code to add contact to group
        } else {
            // Code to remove contact from group
        }
    }
});
```

Explanation

- Attaches an 'OnCheckedChangeListener' to the contact checkbox.
- Executes code to handle the selection and deselection of the contact.

4.9 Review Questions

Section - A

1. List the Components of a Screen
2. Define View in Android.
3. List Some Common Views.
4. What is a ViewGroup in Android?
5. Give two examples of a ViewGroup.
6. What is a Layout in Android?
7. What is the purpose of an XML layout file in Android?
8. What is the purpose of 'setContentView()' method.
9. What is a 'TextView' in Android?

10. Describe the purpose of an 'EditText' in Android.
11. What is the primary function of a ViewGroup in Android?
12. What is a LinearLayout in Android?
13. Define a RelativeLayout in Android.
14. What attribute is used to set the orientation of a LinearLayout?
15. Explain the use of 'layout_weight' in LinearLayout.
16. What is a RelativeLayout in Android?
17. How can you align a view to the bottom of its parent in a RelativeLayout?
18. What is a ConstraintLayout in Android?
19. Mention two advantages of using ConstraintLayout.
20. What is a FrameLayout in Android?
21. What is the primary use case for a FrameLayout?
22. What is a TableLayout in Android?
23. What is a ScrollView in Android?
24. What is Display Orientation ?
25. What does anchoring views mean in Android layout design?
26. How do you anchor a view to the bottom of its parent in a RelativeLayout?
27. What is Resizing and Repositioning Views ?
28. Why is it important to resize and reposition views dynamically?
29. What do you mean by Creating Separate Layouts for Different Orientation ?
30. What is the 'onSaveInstanceState()' method used for?
31. How does 'onRetainNonConfigurationInstance()' help in preserving activity state?
32. What is an Action Bar in Android?
33. Name two components typically found in an Action Bar.
34. What is the purpose of the overflow menu in the Action Bar?
35. What is the purpose of listening for UI notifications in an Android application?
36. What is the onClick event used for in Android?

Section - B

1. Explain the Components of a Screen.
2. What is a View in Android? Explain its features.
3. What is a ViewGroup in Android? Explain its features.
4. Explain any five common views with XML code.
5. What are the advantages and disadvantages of a View.
6. What are the advantages and disadvantages of a ViewGroup.

7. What is a Layout? Explain its features.
8. Explain Linear Layout with an example.
9. Explain Relative Layout with an example.
10. Explain Constraint Layout with an example.
11. Explain Frame Layout with an example.
12. Explain Table Layout with an example.
13. Explain ScrollView with an example.
14. Explain Anchoring Views to Adapt Display Orientation.
15. Explain Resizing and Repositioning Views to Adapt Display Orientation.
16. Explain Creating Separate Layouts for Different Orientations. How it works?
17. What Happens to an Activity's State When Orientation Changes?
18. Explain How to Detect Orientation Changes.
19. Discuss the different ways to control the orientation of an activity.
20. What is an ActionBar? Describe the components of the Action Bar and their respective roles in an Android application.
21. Explain how to use methods like 'onSaveInstanceState()','onRetainNonConfigurationInstance()', and ViewModel to preserve and restore activity state during orientation changes.
22. Describe the steps involved in programmatically detecting orientation changes in an Android application and provide a code example.
23. Describe how to handle key events at the activity level in Android, including examples of methods that can be overridden.

Section - C

1. Discuss in detail how Views and ViewGroups are utilized in creating user interfaces in Android applications.
2. List and briefly describe common views used in Android applications with their XML definitions.
3. Explain Linear Layout along with its features, attributes, advantages and disadvantages.
4. Explain Relative Layout along with its features, attributes, advantages and disadvantages.
5. Explain Constraint Layout along with its features, attributes, advantages and disadvantages.
6. Explain Frame Layout along with its features, attributes, advantages and disadvantages.
7. Explain Table Layout along with its features, attributes, advantages and disadvantages.
8. Explain ScrollView along with its features, attributes, advantages and disadvantages.
9. Explain Techniques and Methods to Adapt Display Orientation.
10. Explain Persisting State Information during Changes in Configuration.
11. Describe in detail the lifecycle methods involved in handling configuration changes in Android.

12. Explain the steps to create a simple Action Bar in an Android application.
13. Explain the process of creating a simple login page programmatically in Android, including the creation and configuration of views.
14. Explain the concept of registering views for various UI events in Android. Provide detailed examples of handling onClick, onLongClick, onTouch, and onKey events.
15. Discuss in detail the different types of UI notifications that can be listened to at the activity level in Android.

UNIT - II

CHAPTER 5

DESIGNING USER INTERFACE WITH VIEWS

CONTENTS

- Introduction
- Using Basic Views
- Using Picker Views
- Using List Views to Display Long Lists
- Understanding Specialized Fragments
- Using Image Views to Display Pictures
- Using Menus With Views
- Using WebView
- Data Persistence
- Sample Applications
- Review Questions

5.1 Introduction

n Android app development, views are the building blocks of the user interface (UI) to create nteractive and visually appealing applications. Views represent UI components such as buttons, text ields, images, and more to interact with the app's content. Views are crucial for creating a seamless ıser experience and are essential for developing modern Android apps.

ıdding views to an Android app involves defining them either in XML layout files or programmatically n Java or Kotlin code. Views can be customized with various attributes to control their appearance ınd behavior, such as text size, color, margins, and more. Additionally, views can be organized using ayouts to create complex UI designs.

)ne of the key aspects of working with views is handling user interactions. Views can respond to user ıctions, such as clicks or touches, by implementing event listeners. For example, a button view can rigger an action when clicked, allowing developers to create interactive features in their apps.

'iews can also be dynamically updated and modified based on the app's state or user input. For ıxample, a text view can display different messages based on the app's logic to provide the real-time eedback to the user.

n the previous chapter, we have learned about the various layouts that can be used to position views n an activity. We also learned about the techniques to adapt to different screen resolutions and sizes. n this chapter, we will take a look at the various views that can be used to design the user interface or an android applications.

5.1.1 View Attributes in Android

'iew attributes in Android are properties that define the appearance, behavior, and interaction of JI elements within the user interface. These attributes play a crucial role in defining how views are lisplayed on the screen, how users can interact with them, and how they behave in response to user ctions. The attributes are key-value pairs that define various properties of views, which are the ıasic building blocks of the user interface. These attributes are specified in XML layout files and ontrol aspects such as the view's size, position, appearance, and behavior.

Importance of View Attributes

'he importance of View attributes in Android development lies in their ability to customize the ppearance, behavior, and interaction of user interface elements.

1. **Layout Control:** View attributes like 'android:layout_width', 'android:layout_height', 'android:padding', and 'android:margin' are essential for defining the size and spacing of UI elements to ensure that the layout is organized, visually appealing, and responsive across different devices and screen sizes.
2. **Separation of Concerns:** By specifying view properties in XML, developers can keep the UI design separate from the application logic. This separation enhances code readability, maintainability, and allows for easier updates to the UI without impacting the underlying logic.

3. **User Interaction:** Attributes such as 'android:onClick', 'android:enabled', and 'android:focusable' are crucial for managing user interactions. They allow developers to define event handlers, control the interactive state of views, and manage the flow of user actions within the application.
4. **Accessibility:** Accessibility attributes like 'android:contentDescription' ensure that applications are usable by people with disabilities. These attributes help create inclusive designs that cater to a broader audience by providing necessary information to accessibility services.
5. **Dynamic Visibility and Behavior:** Attributes such as 'android:visibility' and 'android:alpha' enable developers to dynamically show, hide, or adjust the transparency of views. This capability is important for creating interactive and engaging user interfaces that can respond to user inputs and application states effectively.
6. **Customization:** View attributes allow developers to tailor the look and feel of UI components to match the design requirements of the application. By setting attributes such as 'background', 'padding', 'textColor', and 'textSize', developers can create visually appealing interfaces that align with the app's branding and user experience goals.

5.1.2 Defining View Attributes in XML Layout Files

View attributes are typically defined in XML layout files located in the res/layout directory of an Android project. Each view element in the XML layout file can have attributes specified using XML attributes like android:layout_width, android:layout_height, etc.

Code

```
<TextView
    android:id="@+id/myTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, World!"
    android:textSize="16sp"
    android:textColor="#000000"
    android:gravity="center"
    />
```

5.1.3 Setting Attributes Programmatically

Attributes can also be set programmatically in Java or Kotlin code by accessing the view object and using methods to set properties. When setting attributes programmatically in Java or Kotlin code, developers can access the view object and use methods to dynamically change the properties of UI elements. This approach allows for real-time updates and customization of views based on user interactions, data changes, or application logic.

Code

```
Button myButton = findViewById(R.id.myButton);
// Set visibility of the Button
myButton.setVisibility(View.VISIBLE); // View.VISIBLE, View.INVISIBLE, or View.GONE
// Enable or disable the Button
myButton.setEnabled(true); // true to enable, false to disable
```

5.1.4 Common Attributes for All Views

Common attributes that are applicable to various views in Android are listed below.

Attribute Name	Description	Example	Explanation
android:id	Specifies a unique identifier for the view within the layout.	android:id="@+id/myView"	Assigns a unique ID to the view, which can be used to reference it in Java or Kotlin code.
android:layout_width	Defines the width of the view (e.g., match_parent, wrap_content, specific dimension).	android:layout_width="wrap_content"	Sets the width of the view. Common values are 'match_parent' and 'wrap_content'.
android:layout_height	Specifies the height of the view (e.g., match_parent, wrap_content, specific dimension).	android:layout_height="wrap_content"	Sets the height of the view. Similar to width, it can be 'match_parent' or 'wrap_content'.
android:layout_margin	Sets the margins around the view to control spacing.	android:layout_margin="16dp"	Adds space around the view. Can also use 'android:layout_marginStart', 'android:layout_marginTop', 'android:layout_marginEnd', and 'android:layout_marginBottom' for specific margins.
android:layout_gravity	Specifies how the view should be positioned within its parent layout.	android:layout_gravity="center"	Determines how the view is aligned within its parent container.
android:visibility	Controls the visibility of the view (e.g., visible, invisible, gone).	android:visibility="gone"	Controls the visibility of the view. Common values are 'visible', 'invisible', and 'gone'.
android:background	Sets the background color or drawable for the view.	android:background="@color/backgroundColor"	Sets the background color or drawable of the view.

4. **onMenuOpened:**

The 'onMenuOpened' method is called when a panel's menu is opened by the user. This method allows the activity to perform actions or update the UI when the menu is opened, such as highlighting the current selection or updating the menu items.

Example When the user opens the main menu to access options like "New Group" or "Settings".

```
@Override
public boolean onMenuOpened(int featureId, Menu menu) {
    // Code to handle menu opened event
    return true;  // Event handled
}
```

Explanation

- Executes code to handle menu opened event.
- Returns 'true' to indicate that the event has been handled.

5. **onTouchEvent:**

The 'onTouchEvent' method is called when a touch screen motion event occurs. This method allows the activity to respond to touch events, such as taps, swipes, or other gestures. It provides an opportunity to handle touch interactions that are not already handled by any views within the activity.

Example When the user touches the screen to scroll through chat messages.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            // Code to handle touch down event
            return true;
        case MotionEvent.ACTION_UP:
            // Code to handle touch up event
            return true;
    }
    return super.onTouchEvent(event);  // Pass other events to the super class
}
```

Explanation

- Checks the action of the touch event.
- Executes code to handle touch down and touch up events.
- Returns 'true' to indicate that the event has been handled.

6. **onWindowFocusChanged:**

The 'onWindowFocusChanged' method is called when the current window of the activity gains or loses focus. This method allows the activity to respond to changes in window focus, such as updating the UI or performing specific actions when the window gains or loses focus.

	Example	When switching between WhatsApp and another app.

```
@Override
public void onWindowFocusChanged(boolean hasFocus) {
    super.onWindowFocusChanged(hasFocus);
    if (hasFocus) {
        // Code to handle window gained focus
    } else {
        // Code to handle window lost focus
    }
}
```

Explanation

- Checks whether the window has gained or lost focus.
- Executes code to handle window focus changes.

7. **onConfigurationChanged:**

The 'onConfigurationChanged' method is called when the device configuration changes while the activity is running. This includes changes such as screen orientation, keyboard availability, and locale. This method allows the activity to respond to configuration changes without being restarted.

	Example	When switching between portrait and landscape modes while viewing a chat.

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
        // Code to handle orientation change to landscape
    } else if (newConfig.orientation == Configuration.ORIENTATION_PORTRAIT) {
        // Code to handle orientation change to portrait
    }
}
```

Explanation

- Checks the new configuration's orientation.
- Executes code to handle orientation changes.

8. **onBackPressed**

The 'onBackPressed' method is called when the activity has detected the user's press of the back key. This method allows the activity to handle back press events, such as navigating to the previous screen or exiting the activity.

	Example	When the user presses the back key to exit a chat and return to the main chat list.

```
@Override
public void onBackPressed() {
    // Code to handle back press event
    super.onBackPressed();
}
```

Explanation
• Executes code to handle back press event. • Calls 'super.onBackPressed()' to perform the default back press behavior.

9. **onSaveInstanceState:**

The 'onSaveInstanceState' method is called to save the current state of the activity. This method is invoked before the activity may be destroyed so that the state can be restored in 'onRestoreInstanceState'. It is commonly used to save UI state and user data.

Example	When rotating the device while composing a message, the current message text should be saved.

```
@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    EditText messageEditText = findViewById(R.id.messageEditText);
     outState.putString("draftMessage", messageEditText.getText().toString());
// Save the draft message
}
```

Explanation
Saves the current text in the message 'EditText' to the 'Bundle' so it can be restored later.

10. **onRestoreInstanceState:**

The 'onRestoreInstanceState' method is called after 'onCreate' when the activity is being re-initialized from a previously saved state, given here in 'savedInstanceState'. It is commonly used to restore UI state and user data.

Example	When the device is rotated while composing a message, the previously composed text should be restored.

```
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    EditText messageEditText = findViewById(R.id.messageEditText);
    String draftMessage = savedInstanceState.getString("draftMessage");
    if (draftMessage != null) {
        messageEditText.setText(draftMessage);  // Restore the draft message
    }
}
```

Explanation
Retrieves the saved message text from the 'Bundle' and sets it back to the 'EditText'.

11. **onCreateOptionsMenu**

The 'onCreateOptionsMenu' method is called to initialize the contents of the activity's standard options menu. This method is only called once, the first time the options menu is displayed. It allows the activity to inflate menu items from XML resources.

Example	When creating the main menu for options like "New Group", "New Broadcast", "WhatsApp Web", etc.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);  // Inflate the menu items
    return true;
}
```

Explanation

Inflates the menu items from 'menu_main.xml' into the options menu.

12. onOptionsItemSelected:

The 'onOptionsItemSelected' method is called whenever an item in the options menu is selected. This method allows the activity to handle menu item selections and perform actions based on the selected item.

Example	When the user selects "Settings" from the options menu.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.action_settings) {
        // Code to open settings
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Explanation

- Checks the ID of the selected menu item.
- Executes code to open settings.
- Returns 'true' to indicate that the event has been handled.

4.8.2 Registering Views for Notifications

At the view level, individual UI components (views) such as buttons, text fields, and other interactive elements can be registered for notifications. This involves attaching event listeners to these views to handle specific user actions.

Registering views for notifications involves attaching event listeners to UI components such as buttons, text fields, and other interactive elements. This allows the application to respond to specific user actions like clicks, long clicks, focus changes, touch events, key presses, editor actions, and changes in checked state. By handling these events, applications can provide a responsive and interactive user experience.

COMPLIMENTARY COPY
NOT FOR SALE

Common View Events

1. onClick:

The 'onClick' event is triggered when a view, such as a button, is clicked. This is one of the most common events and is used to perform an action in response to user input.

Example When the user clicks the send button to send a message.

Code XML Layout

```
<Button
    android:id="@+id/sendButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Send" />
```

Code Activity Java Code

```
Button sendButton = findViewById(R.id.sendButton);
sendButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // Code to send message
    }
});
```

Explanation

- Attaches an 'OnClickListener' to the send button.
- Executes code to send the message when the button is clicked.

2. onLongClick:

The 'onLongClick' event is triggered when a view is clicked and held for a longer duration. This is useful for providing additional options or actions.

Example When the user long-clicks a message to bring up options like delete, forward, or copy.

Code XML Layout:

```
<TextView
    android:id="@+id/messageTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, this is a message" />
```

Code Activity Java Code

```
TextView messageTextView = findViewById(R.id.messageTextView);
messageTextView.setOnLongClickListener(new View.OnLongClickListener() {
    @Override
    public boolean onLongClick(View v) {
        // Code to show options like delete, forward, or copy
        return true;  // Event handled
    }
});
```

Explanation

- Attaches an 'OnLongClickListener' to the message text view.
- Executes code to show options when the message is long-clicked.
- Returns 'true' to indicate that the event has been handled.

3. **onFocusChange:**

The 'onFocusChange' event is triggered when a view gains or loses focus. This can be useful for validating input or updating the UI based on focus changes.

Example When the user focuses on the text input field to start typing a message.

Code XML Layout

```
<EditText
    android:id="@+id/messageEditText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Type a message" />
```

Code Activity Java Code

```
EditText messageEditText = findViewById(R.id.messageEditText);
messageEditText.setOnFocusChangeListener(new View.OnFocusChangeListener() {
    @Override
    public void onFocusChange(View v, boolean hasFocus) {
        if (hasFocus) {
            // Code to handle focus gained
        } else {
            // Code to handle focus lost
        }
    }
});
```

Explanation
• Attaches an 'OnFocusChangeListener' to the message edit text. • Executes code to handle when the edit text gains or loses focus.

4. **onTouch:**

The 'onTouch' event is triggered when a touch event occurs on a view. This event can be used to handle various touch interactions like tapping, swiping, or dragging.

Example	When the user touches and drags a message to perform actions like swiping to reply.

Code XML Layout:

```
<TextView
    android:id="@+id/swipeMessageTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Swipe me" />
```

Code Activity Java Code:

```
TextView swipeMessageTextView = findViewById(R.id.swipeMessageTextView);
swipeMessageTextView.setOnTouchListener(new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                // Code to handle touch down
                return true;
            case MotionEvent.ACTION_MOVE:
                // Code to handle touch move
                return true;
            case MotionEvent.ACTION_UP:
                // Code to handle touch up
                return true;
        }
        return false;
    }
});
```

Explanation
• Attaches an 'OnTouchListener' to the swipe message text view. • Executes code to handle touch down, move, and up events. • Returns 'true' to indicate that the event has been handled.

5. **onKey:**

The 'onKey' event is triggered when a hardware key is pressed while the view has focus. This is useful for handling key events within a specific view.

Example When the user presses the enter key to send a message from the text input field.

Code XML Layout

```
<EditText
    android:id="@+id/messageEditText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Type a message"
    android:imeOptions="actionSend" />
```

Code Activity Java Code

```
EditText messageEditText = findViewById(R.id.messageEditText);
messageEditText.setOnKeyListener(new View.OnKeyListener() {
    @Override
    public boolean onKey(View v, int keyCode, KeyEvent event) {
       if (keyCode == KeyEvent.KEYCODE_ENTER && event.getAction()
                    == KeyEvent.ACTION_DOWN) {
            // Code to send message
            return true;
        }
        return false;
    }
});
```

Explanation

- Attaches an 'OnKeyListener' to the message edit text.
- Executes code to send the message when the enter key is pressed.
- Returns 'true' to indicate that the event has been handled.

6. **onCheckedChanged:**

The 'onCheckedChanged' event is triggered when the checked state of a compound button, such as a checkbox or radio button, changes. This is useful for handling changes in selection state.

Example When the user selects or deselects a contact for creating a group.

Code XML Layout

```
<CheckBox
    android:id="@+id/contactCheckBox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Contact Name" />
```

Code Activity Java Code

```
CheckBox contactCheckBox = findViewById(R.id.contactCheckBox);
contactCheckBox.setOnCheckedChangeListener(new CompoundButton.
OnCheckedChangeListener() {
    @Override
   public void onCheckedChanged(CompoundButton buttonView, boolean isChecked)
{
        if (isChecked) {
            // Code to add contact to group
        } else {
            // Code to remove contact from group
        }
    }
});
```

Explanation

- Attaches an 'OnCheckedChangeListener' to the contact checkbox.
- Executes code to handle the selection and deselection of the contact.

4.9 Review Questions

Section - A

1. List the Components of a Screen
2. Define View in Android.
3. List Some Common Views.
4. What is a ViewGroup in Android?
5. Give two examples of a ViewGroup.
6. What is a Layout in Android?
7. What is the purpose of an XML layout file in Android?
8. What is the purpose of 'setContentView()' method.
9. What is a 'TextView' in Android?

10. Describe the purpose of an 'EditText' in Android.
11. What is the primary function of a ViewGroup in Android?
12. What is a LinearLayout in Android?
13. Define a RelativeLayout in Android.
14. What attribute is used to set the orientation of a LinearLayout?
15. Explain the use of 'layout_weight' in LinearLayout.
16. What is a RelativeLayout in Android?
17. How can you align a view to the bottom of its parent in a RelativeLayout?
18. What is a ConstraintLayout in Android?
19. Mention two advantages of using ConstraintLayout.
20. What is a FrameLayout in Android?
21. What is the primary use case for a FrameLayout?
22. What is a TableLayout in Android?
23. What is a ScrollView in Android?
24. What is Display Orientation ?
25. What does anchoring views mean in Android layout design?
26. How do you anchor a view to the bottom of its parent in a RelativeLayout?
27. What is Resizing and Repositioning Views ?
28. Why is it important to resize and reposition views dynamically?
29. What do you mean by Creating Separate Layouts for Different Orientation ?
30. What is the 'onSaveInstanceState()' method used for?
31. How does 'onRetainNonConfigurationInstance()' help in preserving activity state?
32. What is an Action Bar in Android?
33. Name two components typically found in an Action Bar.
34. What is the purpose of the overflow menu in the Action Bar?
35. What is the purpose of listening for UI notifications in an Android application?
36. What is the onClick event used for in Android?

Section - B

1. Explain the Components of a Screen.
2. What is a View in Android? Explain its features.
3. What is a ViewGroup in Android? Explain its features.
4. Explain any five common views with XML code.
5. What are the advantages and disadvantages of a View.
6. What are the advantages and disadvantages of a ViewGroup.

7. What is a Layout? Explain its features.
8. Explain Linear Layout with an example.
9. Explain Relative Layout with an example.
10. Explain Constraint Layout with an example.
11. Explain Frame Layout with an example.
12. Explain Table Layout with an example.
13. Explain ScrollView with an example.
14. Explain Anchoring Views to Adapt Display Orientation.
15. Explain Resizing and Repositioning Views to Adapt Display Orientation.
16. Explain Creating Separate Layouts for Different Orientations. How it works?
17. What Happens to an Activity's State When Orientation Changes?
18. Explain How to Detect Orientation Changes.
19. Discuss the different ways to control the orientation of an activity.
20. What is an ActionBar? Describe the components of the Action Bar and their respective roles in an Android application.
21. Explain how to use methods like 'onSaveInstanceState()','onRetainNonConfigurationInstance()', and ViewModel to preserve and restore activity state during orientation changes.
22. Describe the steps involved in programmatically detecting orientation changes in an Android application and provide a code example.
23. Describe how to handle key events at the activity level in Android, including examples of methods that can be overridden.

Section - C

1. Discuss in detail how Views and ViewGroups are utilized in creating user interfaces in Android applications.
2. List and briefly describe common views used in Android applications with their XML definitions.
3. Explain Linear Layout along with its features, attributes, advantages and disadvantages.
4. Explain Relative Layout along with its features, attributes, advantages and disadvantages.
5. Explain Constraint Layout along with its features, attributes, advantages and disadvantages.
6. Explain Frame Layout along with its features, attributes, advantages and disadvantages.
7. Explain Table Layout along with its features, attributes, advantages and disadvantages.
8. Explain ScrollView along with its features, attributes, advantages and disadvantages.
9. Explain Techniques and Methods to Adapt Display Orientation.
10. Explain Persisting State Information during Changes in Configuration.
11. Describe in detail the lifecycle methods involved in handling configuration changes in Android.

12. Explain the steps to create a simple Action Bar in an Android application.
13. Explain the process of creating a simple login page programmatically in Android, including the creation and configuration of views.
14. Explain the concept of registering views for various UI events in Android. Provide detailed examples of handling onClick, onLongClick, onTouch, and onKey events.
15. Discuss in detail the different types of UI notifications that can be listened to at the activity level in Android.

UNIT - II

CHAPTER 5

DESIGNING USER INTERFACE WITH VIEWS

CONTENTS

- Introduction
- Using Basic Views
- Using Picker Views
- Using List Views to Display Long Lists
- Understanding Specialized Fragments
- Using Image Views to Display Pictures
- Using Menus With Views
- Using WebView
- Data Persistence
- Sample Applications
- Review Questions

5.1 Introduction

n Android app development, views are the building blocks of the user interface (UI) to create nteractive and visually appealing applications. Views represent UI components such as buttons, text ields, images, and more to interact with the app's content. Views are crucial for creating a seamless ıser experience and are essential for developing modern Android apps.

ıdding views to an Android app involves defining them either in XML layout files or programmatically n Java or Kotlin code. Views can be customized with various attributes to control their appearance ınd behavior, such as text size, color, margins, and more. Additionally, views can be organized using ayouts to create complex UI designs.

)ne of the key aspects of working with views is handling user interactions. Views can respond to user ıctions, such as clicks or touches, by implementing event listeners. For example, a button view can rigger an action when clicked, allowing developers to create interactive features in their apps.

'iews can also be dynamically updated and modified based on the app's state or user input. For :xample, a text view can display different messages based on the app's logic to provide the real-time eedback to the user.

n the previous chapter, we have learned about the various layouts that can be used to position views n an activity. We also learned about the techniques to adapt to different screen resolutions and sizes. n this chapter, we will take a look at the various views that can be used to design the user interface or an android applications.

5.1.1 View Attributes in Android

'iew attributes in Android are properties that define the appearance, behavior, and interaction of JI elements within the user interface. These attributes play a crucial role in defining how views are lisplayed on the screen, how users can interact with them, and how they behave in response to user ıctions. The attributes are key-value pairs that define various properties of views, which are the ıasic building blocks of the user interface. These attributes are specified in XML layout files and ontrol aspects such as the view's size, position, appearance, and behavior.

Importance of View Attributes

'he importance of View attributes in Android development lies in their ability to customize the ıppearance, behavior, and interaction of user interface elements.

1. **Layout Control:** View attributes like 'android:layout_width', 'android:layout_height', 'android:padding', and 'android:margin' are essential for defining the size and spacing of UI elements to ensure that the layout is organized, visually appealing, and responsive across different devices and screen sizes.
2. **Separation of Concerns:** By specifying view properties in XML, developers can keep the UI design separate from the application logic. This separation enhances code readability, maintainability, and allows for easier updates to the UI without impacting the underlying logic.

3. **User Interaction:** Attributes such as 'android:onClick', 'android:enabled', and 'android:focusable' are crucial for managing user interactions. They allow developers to define event handlers, control the interactive state of views, and manage the flow of user actions within the application.
4. **Accessibility:** Accessibility attributes like 'android:contentDescription' ensure that applications are usable by people with disabilities. These attributes help create inclusive designs that cater to a broader audience by providing necessary information to accessibility services.
5. **Dynamic Visibility and Behavior:** Attributes such as 'android:visibility' and 'android:alpha' enable developers to dynamically show, hide, or adjust the transparency of views. This capability is important for creating interactive and engaging user interfaces that can respond to user inputs and application states effectively.
6. **Customization:** View attributes allow developers to tailor the look and feel of UI components to match the design requirements of the application. By setting attributes such as 'background', 'padding', 'textColor', and 'textSize', developers can create visually appealing interfaces that align with the app's branding and user experience goals.

5.1.2 Defining View Attributes in XML Layout Files

View attributes are typically defined in XML layout files located in the res/layout directory of an Android project. Each view element in the XML layout file can have attributes specified using XML attributes like android:layout_width, android:layout_height, etc.

Code

```
<TextView
    android:id="@+id/myTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, World!"
    android:textSize="16sp"
    android:textColor="#000000"
    android:gravity="center"
    />
```

5.1.3 Setting Attributes Programmatically

Attributes can also be set programmatically in Java or Kotlin code by accessing the view object and using methods to set properties. When setting attributes programmatically in Java or Kotlin code, developers can access the view object and use methods to dynamically change the properties of UI elements. This approach allows for real-time updates and customization of views based on user interactions, data changes, or application logic.

Code

```
Button myButton = findViewById(R.id.myButton);
// Set visibility of the Button
myButton.setVisibility(View.VISIBLE); // View.VISIBLE, View.INVISIBLE, or View.GONE
// Enable or disable the Button
myButton.setEnabled(true); // true to enable, false to disable
```

5.1.4 Common Attributes for All Views

Common attributes that are applicable to various views in Android are listed below.

Attribute Name	Description	Example	Explanation
android:id	Specifies a unique identifier for the view within the layout.	android:id="@+id/myView"	Assigns a unique ID to the view, which can be used to reference it in Java or Kotlin code.
android:layout_width	Defines the width of the view (e.g., match_parent, wrap_content, specific dimension).	android:layout_width="wrap_content"	Sets the width of the view. Common values are 'match_parent' and 'wrap_content'.
android:layout_height	Specifies the height of the view (e.g., match_parent, wrap_content, specific dimension).	android:layout_height="wrap_content"	Sets the height of the view. Similar to width, it can be 'match_parent' or 'wrap_content'.
android:layout_margin	Sets the margins around the view to control spacing.	android:layout_margin="16dp"	Adds space around the view. Can also use 'android:layout_marginStart', 'android:layout_marginTop', 'android:layout_marginEnd', and 'android:layout_marginBottom' for specific margins.
android:layout_gravity	Specifies how the view should be positioned within its parent layout.	android:layout_gravity="center"	Determines how the view is aligned within its parent container.
android:visibility	Controls the visibility of the view (e.g., visible, invisible, gone).	android:visibility="gone"	Controls the visibility of the view. Common values are 'visible', 'invisible', and 'gone'.
android:background	Sets the background color or drawable for the view.	android:background="@color/backgroundColor"	Sets the background color or drawable of the view.

android:padding	Defines the padding inside the view to create space between the content and the view boundaries.	android:padding="8dp"'	Adds space inside the view, between its edges and its content. Can also use 'android:paddingStart', 'android:paddingTop', 'android:paddingEnd', and 'android:paddingBottom' for specific paddings.
android:onClick	Specifies the method to be called when the view is clicked.	android:onClick ="handleClick"	Defines the name of the method in the activity that will be invoked when the view is clicked.
android:enabled	Enables or disables the view for user interaction.	android:enabled="false"	If false, the view is disabled and cannot be interacted with.
android:focusable	Determines if the view can gain focus.	android:focusable="true"	If true, the view can receive focus.
android:clickable	Indicates if the view is clickable.	android:clickable="true"	If true, the view is clickable.
android:tag	Associates an arbitrary object with the view.	android:tag="myTag"	Used to store additional information about the view.
android:content Description	Provides a description of the view for accessibility purposes.	android:contentDescription ="Description"	Used by accessibility services to describe the view's content to users with disabilities.
android:elevation	Sets the elevation of the view, which affects its shadow.	android:elevation="4dp"	Specifies the elevation of the view in the Z-axis, creating a visual shadow effect.
android:alpha	Sets the opacity of the view.	android:alpha="0.5"	Defines the transparency level of the view, with 1.0 being fully opaque and 0.0 being fully transparent.
android:minWidth	Sets the minimum width of the view.	android:minWidth ="100dp"	Ensures the view is at least the specified width, regardless of its content.
android:minHeight	Sets the minimum height of the view.	android:minHeight ="50dp"	Ensures the view is at least the specified height, regardless of its content.
android:layout_ weight	Specifies how much of the extra space in the layout should be allocated to the view.	android:layout_weight ="1"	Defines the distribution of available space in a LinearLayout among child views.

5.2 Using Basic Views

In Android development, Views are fundamental building blocks used to create the user interface (UI) of an application. A View represents a rectangular area on the screen and is responsible for drawing and handling user interactions. Understanding the basics of Views is essential for designing visually appealing and interactive Android applications.

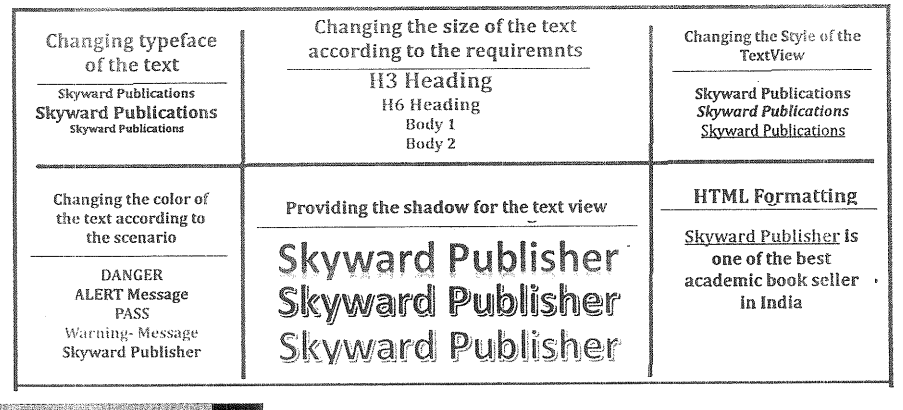
List of Commonly Used Basic Views

Some fundamental views that are essential for designing the user interface (UI) of Android applications are listed below:

1. TextView
2. EditText
3. Button
4. ImageButton
5. CheckBox
6. ToggleButton
7. RadioButton
8. ProgressBar
9. AutoCompleteTextView

5.2.1 TextView

The TextView view in Android is used to display text to the user. It is one of the most fundamental and commonly used views in Android app development. The TextView component that can be styled and customized to fit various design requirements. For example, we can italicize or bold specific parts of the text to highlight important information. Additionally, if the text contains a hyperlink directing to a specific web URL, we can span it with a hyperlink and underline it.



Key Features of TextView

1. **Display Text:** TextView is primarily used to display text on the screen, supporting plain text, formatted text (HTML), and styled text (bold, italic, underline).
2. **Text Styling:** TextView provides options for text styling, including setting text color, text size, text style (bold, italic), text alignment, and text appearance customization.
3. **Text Formatting:** Developers can use HTML-like tags to format text in TextView for the inclusion of links, lists, and other formatted content within the text display.

4. **Accessibility:** TextView supports accessibility features to create apps that are accessible to users with disabilities by specifying content descriptions and other accessibility attributes.
5. **Event Handling:** TextView can respond to user interactions such as clicks and long presses by implementing event listeners for interactive elements within the text display.
6. **Read-Only:** By default, TextView is read-only, preventing users from modifying the displayed text. This feature is beneficial for presenting static content that should not be edited.
7. **Auto-Linking:** TextView can automatically detect web URLs, email addresses, phone numbers, and map addresses within the text and convert them into clickable links for easy access.
8. **Ellipsizing:** TextView supports ellipsizing, which truncates text that is too long to fit within the view and adds an ellipsis ("...") to indicate that the text has been truncated.

Key Attributes of TextView

Attribute Name	Description	Example	Explanation
android:text	Sets the text to be displayed in the TextView.	android:text="Hello, World!"	Specifies the text content of the TextView.
android:textSize	Sets the size of the text.	android:textSize="16sp"	Defines the size of the text. Using 'sp' ensures proper scaling for different screen sizes.
android:textColor	Sets the color of the text.	android:textColor="#FF0000"	Specifies the color of the text using a hexadecimal color code.
android:textStyle	Sets the style of the text (bold, italic, etc.).	android:textStyle="bold"	Applies a style to the text, such as 'bold', 'italic', or 'normal'.
android:fontFamily	Sets the font family for the text.	android:fontFamily="sans-serif"	Specifies the font family to be used for the text, such as 'sans-serif', 'serif', 'monospace', etc.
android:ellipsize	Controls how the text is truncated when it overflows the TextView's width.	android:ellipsize="end"	Specifies how to handle text overflow, such as 'end', 'start', 'middle', or 'marquee'.
android:maxLines	Sets the maximum number of lines for the TextView.	android:maxLines="2"	Limits the number of lines of text displayed in the TextView.
android:hint	Sets the hint text to be displayed when the TextView is empty.	android:hint="Enter your name"	Displays a hint or placeholder text when the TextView is empty.

Defining TextView in XML

To begin working with TextView in Android Studio we need to create a new project and navigate to the layout file to work with the TextView element. By default, Android Studio creates a layout file named activity_main.xml in the res/layout folder. In activity_main.xml file, we can add a TextView element within a layout container, such as LinearLayout or RelativeLayout as shown below.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/myTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, World!"
        android:textSize="24sp"
        android:textStyle="bold"
        android:layout_gravity="center"/>

</LinearLayout>
```

In this example, we've added a TextView with the id myTextView that displays the text "Hello, World!" in a bold style and with a font size of 24sp. The layout_gravity attribute is set to "center" to horizontally center the TextView within its parent layout.

Using TextView in Activity Code

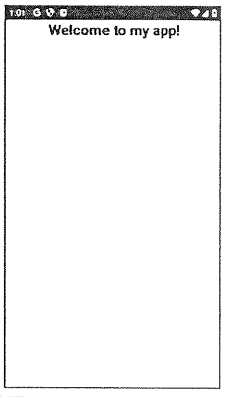
Next, we need to update MainActivity.java file to interact with this TextView. We can do this by finding the TextView using its id and then setting or modifying its text.

Code MainActivity.java

```
package com.example.myfirstapp;
import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;
import android.widget.TextView;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        // Find the TextView
        TextView myTextView = findViewById(R.id.myTextView);
        // Update the text of the TextView
        myTextView.setText("Welcome to my app!");
    }
}
```

Output



Explanation

In this MainActivity.java file, we first find the TextView using findViewById(R.id.myTextView) and then update its text using setText("Welcome to my app!").

When we run app, we should see the updated text displayed in the TextView. This is a basic example of how to work with TextViews in Android Studio.

5.2.2. EditText View

EditText is a key component in Android development, designed to allow users to input and modify text. The EditText view in Android is used to provide an input field where users can enter text. It allows users to input alphanumeric characters, numbers, symbols, and more. It extends from the TextView class, inheriting its properties while adding functionality specific to user input. EditText can handle various types of data, support validation, and be customized extensively to fit the design requirements of an application.

Features of EditText

1. **User Input:** EditText allows users to input and edit text within the view capturing user-generated content.
2. **Input Types:** Supports various input types such as text, number, password, email, phone, and more to specify the type of data expected from users.
3. **Validation:** EditText is used to validate user input and display error messages if the input does not meet specified criteria or constraints.
4. **Auto-complete:** EditText can provide suggestions and auto-complete capabilities to enhance the user experience by predicting and completing text input based on previous input or predefined options.
5. **Customization:** It offers extensive customization options for text size, color, background, padding, and other visual attributes to tailor the appearance of EditText to suit the app's design.

6. **Formatting:** It supports text formatting features such as bold, italic, underline, and more to format text input within the EditText view.
7. **Event Handling:** EditText can respond to user interactions like clicks and long presses by implementing event listeners to create interactive input fields.
8. **Accessibility:** It supports accessibility features to ensure apps are accessible to users with disabilities. Developers can provide content descriptions and other accessibility attributes for EditText to enhance usability for all users.

Key Attributes

Attribute Name	Description	Example	Explanation
android:hint	Sets a hint to be displayed when the EditText is empty.	android:hint="Enter your name"	Displays a hint or placeholder text to guide the user on what to enter.
android:inputType	Specifies the type of input expected (e.g., text, number, password).	android:inputType ="textPassword"	Defines the type of data that the EditText expects, such as 'text', 'number', 'phone', etc.
android:ems	Sets the EditText width to a specified number of ems (width of 'M' characters).	android:ems="10"	Specifies the width of the EditText in terms of ems (the width of the letter 'M').
android:maxLength	Sets the maximum number of characters allowed in the EditText.	android:maxLength="30"	Limits the number of characters the user can enter.
android:singleLine	Restricts the EditText to a single line of text.	android:singleLine="true"	If true, forces the EditText to be single-line, preventing line breaks.
android:textColorHint	Sets the color of the hint text.	android:textColorHint ="#888888"	Specifies the color of the hint text to improve visibility.

Defining EditText in XML

To begin working with EditText in Android Studio, we need to create a new project and navigate to the layout file to work with the EditText element. By default, Android Studio creates a layout file named activity_main.xml in the res/layout folder. In activity_main.xml file, we can add an EditText element within a layout container, such as LinearLayout or RelativeLayout as shown below.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:padding="16dp">

    <EditText
        android:id="@+id/usernameEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter your username"
        android:inputType="textPersonName"
        android:textColorHint="#888888"
        android:textSize="16sp"
        android:padding="10dp"
        android:layout_marginBottom="16dp" />

    <EditText
        android:id="@+id/passwordEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter your password"
        android:inputType="textPassword"
        android:textColorHint="#888888"
        android:textSize="16sp"
        android:padding="10dp" />
</LinearLayout>
```

This XML code defines a vertical LinearLayout containing two EditText views for user input: one for entering a username and one for entering a password. Each EditText has a hint to guide the user, specified input types to ensure appropriate keyboard behavior, and additional styling attributes for text size, hint color, padding, and layout margins to enhance readability and user experience.

Using EditText in Activity Code

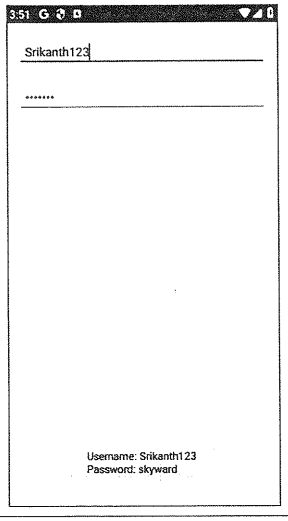
To use EditText in an activity code, we need to reference the EditText elements defined in a XML layout and interact with them in an activity.

Code MainActivity.java

```
package com.example.viewdemo;

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

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Find the EditText views by their IDs
        EditText usernameEditText = findViewById(R.id.usernameEditText);
        EditText passwordEditText = findViewById(R.id.passwordEditText);

        // Setting default text (this can be removed if not needed)
        usernameEditText.setText("Srikanth123");
        passwordEditText.setText("skyward");

        // Getting text from EditText
        String username = usernameEditText.getText().toString();
        String password = passwordEditText.getText().toString();

        // Displaying a toast message with the entered username and password
        showToast("Username: " + username + "\nPassword: " + password);
    }

    // Method to show toast message
    private void showToast(String message) {
        Toast.makeText(this, message, Toast.LENGTH_LONG).show();
    }
}
```

Output



Explanation

This code defines an Android activity that demonstrates the interaction with EditText views for a username and password input. In the onCreate method, the EditText views are initialized by finding them through their IDs defined in the layout XML. Default text values are set for both the username ("Srikanth123") and password ("skyward") fields. The text entered in these EditText views is then retrieved and stored in string variables. A custom method showToast is used to display a Toast message containing the entered username and password.

5.2.3 Button

'Button' is a fundamental UI component in Android development and it is designed to trigger actions when clicked. The 'Button' view in Android provides a clickable interface element that users can interact with to initiate specific actions such as submitting a form, navigating to another screen, or performing any other task. It extends from the 'TextView' class, inheriting its properties while adding functionality specific to user interactions.

Features of Button

1. **User Interaction:** 'Button' allows users to initiate actions within the application by clicking the button.
2. **Customization:** Supports extensive customization options for text size, color, background, padding, and other visual attributes to match the app's design.
3. **Text and Icons:** 'Button' can display text and icons for versatile design options.
4. **Event Handling:** Responds to user interactions like clicks by implementing event listeners to perform specific actions.
5. **Accessibility:** Supports accessibility features to ensure apps are accessible to users with disabilities. Developers can provide content descriptions and other accessibility attributes for 'Button' to enhance usability for all users.

Key Attributes

Attribute Name	Description	Example	Explanation
android:text	Sets the text to be displayed on the button.	android:text="Click Me"	Specifies the text label of the button.
android:textColor	Sets the color of the text on the button.	android:textColor="#FFFFFF"	Defines the color of the button text.
android:textSize	Sets the size of the text on the button.	android:textSize="18sp"	Specifies the size of the text displayed on the button.
android:drawableLeft	Sets a drawable (icon) to be displayed to the left of the button text.	android:drawableLeft="@ drawable/icon"	Adds an icon to the left of the button text.
android:background	Sets the background color or drawable of the button.	android:background="@color/ buttonColor"	Specifies the background color or drawable for the button.
android:onClick	Specifies the method to be called when the button is clicked.	android:onClick="handleClick"	Defines the name of the method in the activity that will be invoked when the button is clicked.
android:enabled	Enables or disables the button for user interaction.	android:enabled="false"	If false, the button is disabled and cannot be interacted with.

Defining Buttons in XML

To begin working with 'Button' in Android Studio, create a new project and navigate to the layout file to work with the 'Button' element. By default, Android Studio creates a layout file named 'activity_main.xml' in the 'res/layout' folder. In the 'activity_main.xml' file, add 'Button' elements within a layout container, such as 'LinearLayout' or 'RelativeLayout' as shown below.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button 1"
        android:textSize="18sp"
        android:textColor="#FFFFFF"
        android:padding="10dp"
        android:layout_gravity="center"
        android:onClick="onButton1Click" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button 2"
        android:textSize="18sp"
        android:textColor="#FFFFFF"
        android:padding="10dp"
        android:layout_gravity="center"
        android:onClick="onButton2Click" />
</LinearLayout>
```

Explanation :

LinearLayout: Used as the root layout with vertical orientation and padding.

- **Button 1:** A button with an ID button1, centered horizontally, with specified text, text size, text color, and padding. The onClick attribute is set to onButton1Click.
- **Button 2:** A button with an ID button2, centered horizontally, with specified text, text size, text color, and padding. The onClick attribute is set to onButton2Click.

Using Buttons in Activity Code

To use 'Button' in an activity code, reference the 'Button' elements defined in the XML layout and interact with them in an activity.

Code MainActivity.java

```
package com.example.viewdemo;

import android.os.Bundle;
import android.view.View;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

    // Method to handle Button 1 click event
    public void onButton1Click(View view) {
        showToast("Button 1 clicked!");
    }

    // Method to handle Button 2 click event
    public void onButton2Click(View view) {
        showToast("Button 2 clicked!");
    }

    // Method to show toast message
    private void showToast(String message) {
        Toast.makeText(this, message, Toast.LENGTH_LONG).show();
    }
}
```

Explanation :	

- **onCreate Method:** Sets the content view to activity_main.xml.
- **onButton1Click Method:** Handles the click event for Button 1 and displays a toast message "Button 1 clicked!".
- **onButton2Click Method:** Handles the click event for Button 2 and displays a toast message "Button 2 clicked!".
- **showToast Method:** Displays a toast message with the provided text.

5.2.4 ImageButton

'ImageButton' is a fundamental UI component in Android development, designed to provide a clickable button with an image. The 'ImageButton' view in Android provides an interface element that users can interact with to initiate specific actions, similar to a regular button but with the added capability of displaying an image instead of text. It extends from the 'ImageView' class, inheriting its properties while adding functionality specific to user interactions.

Features of ImageButton

1. **User Interaction:** 'ImageButton' allows users to initiate actions within the application by clicking the button.
2. **Image Display:** Supports displaying images as the button's visual content, making it suitable for icons, graphics, and other visual elements.
3. **Customization:** Offers extensive customization options for image resources, background, padding, and other visual attributes to match the app's design.
4. **Event Handling:** Responds to user interactions like clicks by implementing event listeners to perform specific actions.
5. **Accessibility:** Supports accessibility features to ensure apps are accessible to users with disabilities. Developers can provide content descriptions and other accessibility attributes for 'ImageButton' to enhance usability for all users.

Key Attributes

Attribute Name	Description	Example	Explanation
android:src	Sets the image to be displayed on the button.	android:src="@drawable/icon"	Specifies the drawable resource to be used as the button's image.
android:content Description	Provides a description of the image for accessibility purposes.	android:contentDescription ="Button image"	Helps accessibility services describe the button's image to users with disabilities.
android:background	Sets the background color or drawable of the ImageButton.	android:background="@color/ buttonColor"	Specifies the background color or drawable for the ImageButton.

android:scaleType	Controls how the image should be resized or moved to match the size of the ImageButton.	android:scaleType ="fitCenter"	Determines how the image is scaled within the button (e.g., 'fitCenter', 'centerCrop').
android:adjustView Bounds	Allows the ImageButton to adjust its bounds to preserve the aspect ratio of the drawable.	android:adjustViewBounds ="true"	If true, the button adjusts its bounds to maintain the aspect ratio of the image.
android:alpha	Sets the opacity of the ImageButton.	android:alpha="0.5"	Defines the transparency level of the button, with 1.0 being fully opaque and 0.0 being fully transparent.
android:onClick	Specifies the method to be called when the ImageButton is clicked.	android:onClick="handleImage ButtonClick"	Defines the name of the method in the activity that will be invoked when the button is clicked.

Defining ImageButton in XML

To begin working with 'ImageButton' in Android Studio, create a new project and navigate to the layout file to work with the 'ImageButton' element. By default, Android Studio creates a layout file named 'activity_main.xml' in the 'res/layout' folder. In the 'activity_main.xml' file, add 'ImageButton' elements within a layout container, such as 'LinearLayout' or 'RelativeLayout' as shown below.

Note: Download icons and save them as 'icon1.png' and 'icon2.png' in the 'res/drawable' folder.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <ImageButton
        android:id="@+id/imageButton1"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:src="@drawable/icon1"
        android:contentDescription="First Icon Button"
        android:padding="20dp"
```

```
        android:layout_gravity="center"
        android:onClick="onImageButton1Click" />

    <ImageButton
        android:id="@+id/imageButton2"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:src="@drawable/icon2"
        android:contentDescription="Second Icon Button"
        android:padding="20dp"
        android:layout_gravity="center"
        android:onClick="onImageButton2Click" />
</LinearLayout>
```

This XML code defines a vertical 'LinearLayout' containing two 'ImageButton' views for user interaction. Each 'ImageButton' has an image resource to display, specified padding, content description for accessibility, and layout gravity to enhance usability and user experience.

Using ImageButtons in Activity Code

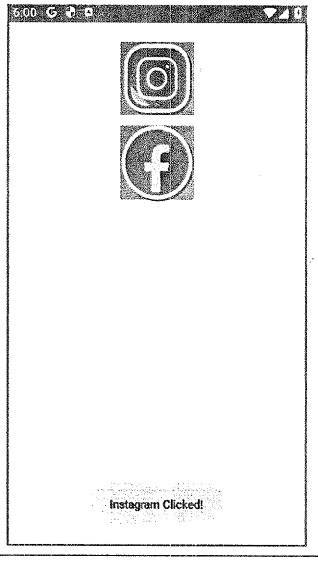
To use 'ImageButton' in an activity code, reference the 'ImageButton' elements defined in the XML layout and interact with them in an activity.

Code MainActivity.java

```
package com.example.viewdemo;

import android.os.Bundle;
import android.view.View;
import android.widget.ImageButton;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        // Find the ImageButton views by their IDs
        ImageButton imageButton1 = findViewById(R.id.imageButton1);
        ImageButton imageButton2 = findViewById(R.id.imageButton2);
```

```
    // Method to handle ImageButton 1 click event
    public void onImageButton1Click(View view) {
        showToast("ImageButton 1 clicked!");
    }

    // Method to handle ImageButton 2 click event
    public void onImageButton2Click(View view) {
        showToast("ImageButton 2 clicked!");
    }

    // Method to show toast message
    private void showToast(String message) {
        Toast.makeText(this, message, Toast.LENGTH_LONG).show();
    }
}
```

Output



Explanation

This code defines an Android activity that demonstrates interaction with two ImageButton views representing Instagram and Facebook icons. In the onCreate method, the ImageButton views are initialized by finding them through their IDs defined in the XML layout (imageButton1 and imageButton2). Click listeners are set up for each ImageButton, where clicking the first button triggers a toast message displaying "Instagram Clicked!" and clicking the second button triggers a toast message displaying "Facebook Clicked!". The showToast method is a custom method used to encapsulate the logic for displaying toast messages.

5.2.5 CheckBox

'CheckBox' is a UI component in Android that allows users to make a binary choice, typically indicating a choice between true and false. The 'CheckBox' view in Android provides a checkbox that can be checked or unchecked by the user. It extends from the 'CompoundButton' class, inheriting its properties while adding functionality specific to checkboxes.

Features of CheckBox

1. **User Interaction:** Allows users to select or deselect an option by checking or unchecking the box.
2. **State Management:** Can be programmatically set to checked or unchecked, allowing dynamic changes in the application's logic.
3. **Event Handling:** Responds to user interactions by implementing event listeners to perform specific actions based on the checked state.
4. **Customization:** Supports customization options for text, text size, text color, background, padding, and other visual attributes to match the app's design.
5. **Accessibility:** Supports accessibility features to ensure apps are accessible to users with disabilities. Developers can provide content descriptions and other accessibility attributes for 'CheckBox' to enhance usability for all users.

Key Attributes

Attribute Name	Description	Example	Explanation
android:text	Sets the text to be displayed next to the CheckBox.	'android:text="Accept Terms"'	Specifies the label text of the CheckBox.
android:checked	Sets the initial checked state of the CheckBox.	'android:checked="true"'	Defines whether the CheckBox is initially checked ('true') or unchecked ('false').
android:buttonTint	Sets the tint color for the CheckBox button.	'android:buttonTint="@color/ checkboxColor"'	Specifies the color used to tint the CheckBox button.
android:onClick	Specifies the method to be called when the CheckBox is clicked.	'android:onClick="handleCheck BoxClick"'	Defines the name of the method in the activity that will be invoked when the CheckBox is clicked.
android:textColor	Sets the color of the text displayed next to the CheckBox.	'android:textColor="#000000"'	Defines the color of the CheckBox text.
android:textSize	Sets the size of the text displayed next to the CheckBox.	'android:textSize="14sp"'	Specifies the size of the text next to the CheckBox.
android:button Drawable	Sets a custom drawable for the CheckBox button.	'android:buttonDrawable="@ drawable/custom_checkbox"'	Allows using a custom drawable as the CheckBox button.

Defining CheckBox in XML

To begin working with 'CheckBox' in Android Studio, create a new project and navigate to the layout file to work with the 'CheckBox' element. By default, Android Studio creates a layout file named 'activity_main.xml' in the 'res/layout' folder. In the 'activity_main.xml' file, add 'CheckBox' elements within a layout container, such as 'LinearLayout' or 'RelativeLayout' as shown below.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:gravity="center_horizontal">

    <CheckBox
        android:id="@+id/checkBox1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 1"
        android:textSize="16sp"
        android:textColor="#000000"
        android:padding="10dp"
        android:layout_marginBottom="20dp" />

    <CheckBox
        android:id="@+id/checkBox2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 2"
        android:textSize="16sp"
        android:textColor="#000000"
        android:padding="10dp" />
</LinearLayout>
```

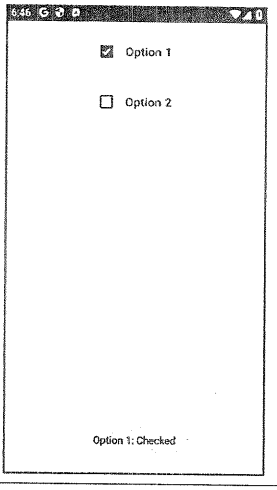
This XML code defines a vertical 'LinearLayout' containing two 'CheckBox' views for user interaction. Each 'CheckBox' has text to guide the user, specified text size, text color, padding, and layout margins to enhance usability and user experience.

Using CheckBox in Activity Code

To use 'CheckBox' in an activity code, reference the 'CheckBox' elements defined in the XML layout and interact with them in an activity.

Code MainActivity.java

```
package com.example.viewdemo;

import android.os.Bundle;
import android.view.View;
import android.widget.CheckBox;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        // Find the CheckBox views by their IDs
        CheckBox checkBox1 = findViewById(R.id.checkBox1);
        CheckBox checkBox2 = findViewById(R.id.checkBox2);

        // Setting up click listeners for the CheckBoxes
        checkBox1.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
              showToast("Option 1: " + (checkBox1.isChecked() ?
                          "Checked" : "Unchecked"));
            }
        });

        checkBox2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
              showToast("Option 2: " + (checkBox2.isChecked()
                          ? "Checked" : "Unchecked"));
            }
        });
    }

    // Method to show toast message
    private void showToast(String message) {
        Toast.makeText(this, message, Toast.LENGTH_LONG).show();
    }
}
```

Output	Explanation
	This code defines an Android activity that demonstrates interaction with two 'CheckBox' views. In the 'onCreate' method, the 'CheckBox' views are initialized by finding them through their IDs defined in the layout XML. Click listeners are set up for each 'CheckBox', where clicking a checkbox triggers a toast message displaying whether the option is checked or unchecked. The 'showToast' method encapsulates the logic for displaying toast messages.

5.2.6 ToggleButton

'ToggleButton' is a UI component in Android that allows users to switch between two states: checked and unchecked. It provides a visual indicator for the on/off state, making it useful for settings and options that can be toggled. The 'ToggleButton' extends from the 'CompoundButton' class, inheriting its properties while adding functionality specific to toggle switches.

ToggleButton is suitable for scenarios where users need to switch between two distinct states or options. ToggleButton can be used for enabling/disabling a feature, toggling a setting, or switching between modes.

Features of ToggleButton

1. **User Interaction:** Allows users to toggle between two states (checked and unchecked) by clicking the button.
2. **State Management:** Can be programmatically set to checked or unchecked, allowing dynamic changes in the application's logic.
3. **Event Handling:** Responds to user interactions by implementing event listeners to perform specific actions based on the checked state.
4. **Customization:** Supports customization options for text, text size, text color, background, padding, and other visual attributes to match the app's design.
5. **Accessibility:** Supports accessibility features to ensure apps are accessible to users with disabilities. Developers can provide content descriptions and other accessibility attributes for 'ToggleButton' to enhance usability for all users.

Key Attributes

Attribute Name	Description	Example	Explanation
android:textOn	Sets the text to be displayed when the ToggleButton is in the "on" state.	android:textOn="ON"	Specifies the label text shown when the ToggleButton is checked.
android:textOff	Sets the text to be displayed when the ToggleButton is in the "off" state.	android:textOff="OFF"	Specifies the label text shown when the ToggleButton is unchecked.
android:checked	Sets the initial checked state of the ToggleButton.	android:checked="true"	Defines whether the ToggleButton is initially in the "on" ('true') or "off" ('false') state.
android:text	Sets the text to be displayed on the ToggleButton.	android:text="Toggle"	Specifies the default text of the ToggleButton.
android:textColor	Sets the color of the text on the ToggleButton.	android:textColor="#FFFFFF"	Defines the color of the text displayed on the ToggleButton.
android:textSize	Sets the size of the text on the ToggleButton.	android:textSize="18sp"	Specifies the size of the text displayed on the ToggleButton.
android:background	Sets the background color or drawable of the ToggleButton.	android:background="@color/buttonColor"	Specifies the background color or drawable for the ToggleButton.
android:onClick	Specifies the method to be called when the ToggleButton is clicked.	android:onClick="handle Toggle Click"	Defines the name of the method in the activity that will be invoked when the ToggleButton is clicked.

Defining ToggleButton in XML

To begin working with 'ToggleButton' in Android Studio, create a new project and navigate to the layout file to work with the 'ToggleButton' element. By default, Android Studio creates a layout file named 'activity_main.xml' in the 'res/layout' folder. In the 'activity_main.xml' file, add 'ToggleButton' elements within a layout container, such as 'LinearLayout' or 'RelativeLayout' as shown below.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:gravity="center_horizontal">

    <ToggleButton
        android:id="@+id/toggleButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textOn="ON"
        android:textOff="OFF"
        android:checked="false"
        android:textSize="16sp"
        android:textColor="#000000"
        android:padding="10dp"
        android:layout_marginBottom="20dp"
        android:onClick="onToggleClick" />

    <ToggleButton
        android:id="@+id/toggleButton2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textOn="YES"
        android:textOff="NO"
        android:checked="false"
        android:textSize="16sp"
        android:textColor="#000000"
        android:padding="10dp"
        android:onClick="onToggleClick" />
</LinearLayout>
```

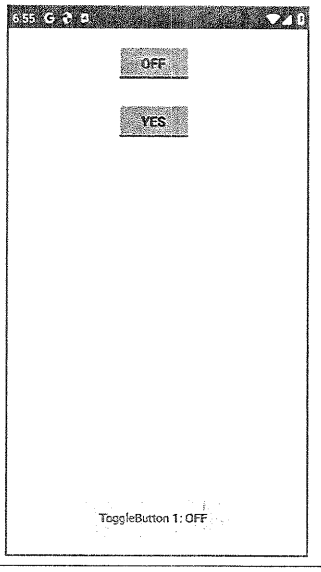
This XML code defines a vertical 'LinearLayout' containing two 'ToggleButton' views for user interaction. Each 'ToggleButton' has text to guide the user, specified text size, text color, padding, and layout margins to enhance usability and user experience.

Using ToggleButton in Activity Code

To use 'ToggleButton' in an activity code, reference the 'ToggleButton' elements defined in the XML layout and interact with them in an activity.

Code MainActivity.java

```
package com.example.viewdemo;

import android.os.Bundle;
import android.view.View;
import android.widget.ToggleButton;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        // Find the ToggleButton views by their IDs
        ToggleButton toggleButton1 = findViewById(R.id.toggleButton1);
        ToggleButton toggleButton2 = findViewById(R.id.toggleButton2);

        // Setting up click listeners for the ToggleButtons
        toggleButton1.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                showToast("ToggleButton 1: " + (toggleButton1.isChecked() ?
                        "ON" : "OFF"));
            }
        });

        toggleButton2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                showToast("ToggleButton 2: " + (toggleButton2.isChecked() ?
                        "YES" : "NO"));
            }
        });
    }

   // Method to handle ToggleButton click events
    public void onToggleClick(View view) {
        ToggleButton toggleButton = (ToggleButton) view;
        String state = toggleButton.isChecked() ?
         toggleButton.getTextOn().toString() : toggleButton.getTextOff().toString();
        showToast("ToggleButton state: " + state);
    }
    // Method to show toast message
    private void showToast(String message) {
        Toast.makeText(this, message, Toast.LENGTH_LONG).show();
    }
}
```

Output	Explanation
	This code defines an Android activity that demonstrates interaction with two 'ToggleButton' views. In the 'onCreate' method, the 'ToggleButton' views are initialized by finding them through their IDs defined in the layout XML. Click listeners are set up for each 'ToggleButton', where clicking a button triggers a toast message displaying whether the toggle button is in the on or off state. The 'showToast' method encapsulates the logic for displaying toast messages.

5.2.7 RadioButton

'RadioButton' is a UI component in Android that allows users to select one option from a set of mutually exclusive options. When a 'RadioButton' is checked, all other 'RadioButton' views within the same 'RadioGroup' are automatically unchecked. It extends from the 'CompoundButton' class, inheriting its properties while adding functionality specific to radio buttons.

Features of RadioButton

1. **User Interaction:** Allows users to select a single option from a group of choices.
2. **Mutual Exclusivity:** Only one 'RadioButton' can be checked at a time within a 'RadioGroup'.
3. **State Management:** Can be programmatically set to checked or unchecked, allowing dynamic changes in the application's logic.
4. **Event Handling:** Responds to user interactions by implementing event listeners to perform specific actions based on the checked state.
5. **Customization:** Supports customization options for text, text size, text color, background, padding, and other visual attributes to match the app's design.
6. **Accessibility:** Supports accessibility features to ensure apps are accessible to users with disabilities. Developers can provide content descriptions and other accessibility attributes for 'RadioButton' to enhance usability for all users.

Key Attributes

Attribute Name	Description	Example	Explanation
android:text	Sets the text to be displayed next to the RadioButton.	android:text="Option 1"	Specifies the label text of the RadioButton.
android:checked	Sets the initial checked state of the RadioButton.	android:checked="true"	Defines whether the RadioButton is initially selected ('true') or not ('false').
android:buttonTint	Sets the tint color for the RadioButton button.	android:buttonTint="@color/radioColor"	Specifies the color used to tint the RadioButton button.
android:onClick	Specifies the method to be called when the RadioButton is clicked.	android:onClick="handle RadioClick"	Defines the name of the method in the activity that will be invoked when the RadioButton is clicked.
android:textColor	Sets the color of the text displayed next to the RadioButton.	android:textColor="#000000"	Defines the color of the RadioButton text.
android:textSize	Sets the size of the text displayed next to the RadioButton.	android:textSize="14sp"	Specifies the size of the text next to the RadioButton.
android:button Drawable	Sets a custom drawable for the RadioButton button.	android:buttonDrawable="@drawable/custom_radio"	Allows using a custom drawable as the RadioButton button.

Defining RadioButton in XML

To begin working with 'RadioButton' in Android Studio, create a new project and navigate to the layout file to work with the 'RadioButton' element. By default, Android Studio creates a layout file named 'activity_main.xml' in the 'res/layout' folder. In the 'activity_main.xml' file, add 'RadioButton' elements within a 'RadioGroup' container to ensure mutual exclusivity, as shown below.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    android:gravity="center_horizontal">
```

```
    <RadioGroup
        android:id="@+id/radioGroup"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical">

        <RadioButton
            android:id="@+id/radioButton1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Male"
            android:textSize="16sp"
            android:textColor="#000000"
            android:padding="10dp"
            android:layout_marginBottom="20dp" />

        <RadioButton
            android:id="@+id/radioButton2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Female"
            android:textSize="16sp"
            android:textColor="#000000"
            android:padding="10dp" />
    </RadioGroup>
</LinearLayout>
```

This XML code defines a vertical 'LinearLayout' containing a 'RadioGroup' with two 'RadioButton' views for user interaction. Each 'RadioButton' has text to guide the user, specified text size, text color, padding, and layout margins to enhance usability and user experience.

Using RadioButton in Activity Code

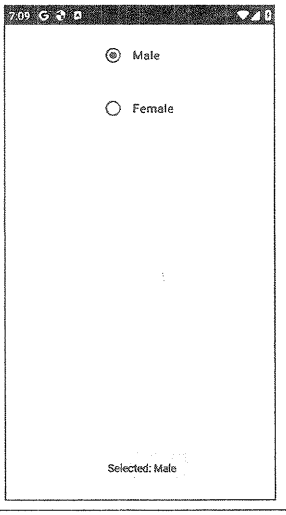
To use 'RadioButton' in an activity code, reference the 'RadioButton' elements defined in the XML layout and interact with them in an activity.

Code MainActivity.java

```
package com.example.viewdemo;

import android.os.Bundle;
import android.widget.RadioButton;
import android.widget.RadioGroup;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;
```

```
public class MainActivity extends AppCompatActivity {

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

        // Find the RadioGroup view by its ID
        RadioGroup radioGroup = findViewById(R.id.radioGroup);

        // Setting up a checked change listener for the RadioGroup
        radioGroup.setOnCheckedChangeListener
                    (new RadioGroup.OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(RadioGroup group, int checkedId) {
                // Find the selected RadioButton by its ID
                RadioButton selectedRadioButton = findViewById(checkedId);
                // Display a toast message with the selected option
                showToast("Selected: " + selectedRadioButton.getText());
            }
        });
    }

    // Method to show toast message
    private void showToast(String message) {
        Toast.makeText(this, message, Toast.LENGTH_LONG).show();
    }
}
```

Output



Explanation

This code defines an Android activity that demonstrates interaction with a 'RadioGroup' containing two 'RadioButton' views. In the 'onCreate' method, the 'RadioGroup' is initialized by finding it through its ID defined in the layout XML. A checked change listener is set up for the 'RadioGroup', which triggers a toast message displaying the selected option when a 'RadioButton' is checked. The 'showToast' method encapsulates the logic for displaying toast messages.

5.2.8 ProgressBar

'ProgressBar' is a UI component in Android that visually indicates the progress of a task. It is commonly used to show the status of ongoing operations, such as loading data, downloading files, or processing tasks. Progress bars enhance user experience by providing feedback on the progress of a long-running task.

Features of ProgressBar

1. **Indeterminate and Determinate Modes:** 'ProgressBar' can operate in two modes. Indeterminate mode indicates an ongoing process without a specific endpoint, while determinate mode shows specific progress.
2. **Customization:** 'ProgressBar' supports extensive customization options, including size, color, orientation, and style.
3. **Accessibility:** Supports accessibility features to ensure apps are accessible to users with disabilities. Developers can provide content descriptions and other accessibility attributes for 'ProgressBar' to enhance usability for all users.

Types of ProgressBar with Example Scenarios

1. **Indeterminate ProgressBar:**

   Meaning: An indeterminate 'ProgressBar' indicates an ongoing process where the duration is unknown. It typically uses a looping animation.

   Example : Loading data from a server where the exact time to complete is not known.

Code XML

```
<ProgressBar
    android:id="@+id/indeterminateProgressBar"
    style="?android:attr/progressBarStyleLarge"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:indeterminate="true" />
```

Code Java

```
ProgressBar indeterminateProgressBar = findViewById(R.id.indeterminateProgressBar);
indeterminateProgressBar.setVisibility(View.VISIBLE); // To show the progress bar
indeterminateProgressBar.setVisibility(View.GONE); // To hide the progress bar when done
```

2. **Determinate ProgressBar:**

   Meaning: A determinate 'ProgressBar' indicates progress toward a known goal. It shows a percentage of completion.

   Example : File download progress where the total size is known and progress can be measured.

Code XML

```
<ProgressBar
    android:id="@+id/determinateProgressBar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:progress="0"
    android:max="100" />
```

Code Java

```
ProgressBar determinateProgressBar = findViewById(R.id.determinateProgressBar);
determinateProgressBar.setProgress(50); // Update progress to 50%
```

Key Attributes

Attribute Name	Description	Example	Explanation
android:indeterminate	Determines whether the ProgressBar is indeterminate (continuous animation) or determinate.	android:indeterminate ="true"	If true, the ProgressBar shows an ongoing animation without a defined endpoint.
android:max	Sets the maximum value of the ProgressBar when in determinate mode.	android:max="100"	Defines the maximum value that represents the completion of the progress.
android:progress	Sets the current progress value of the ProgressBar when in determinate mode.	android:progress="50"	Specifies the current progress value, which should be between 0 and the maximum value.
android:secondary Progress	Sets the secondary progress value, which is often used to show intermediate progress.	android:secondary Progress="75"	Represents a secondary progress value, often used to show buffering progress.
android:progress Drawable	Sets a custom drawable for the progress indicator.	android:progress Drawable="@drawable/ custom_progress"	Allows using a custom drawable to visually represent the progress.
android:indeterminate Drawable	Sets a custom drawable for the indeterminate progress indicator.	android:indeterminate Drawable="@drawable/ custom_indeterminate"	Allows using a custom drawable to visually represent the indeterminate progress.

android:minHeight	Sets the minimum height of the ProgressBar.	android:minHeight ="10dp"	Ensures the ProgressBar is at least the specified height.
android:minWidth	Sets the minimum width of the ProgressBar.	android:minWidth ="200dp"	Ensures the ProgressBar is at least the specified width.

Example : File Download ProgressBar

An example scenario demonstrating the use of a determinate 'ProgressBar' for a file download:

Code XML

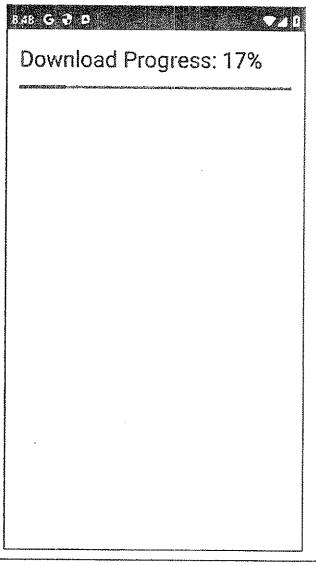
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <TextView
        android:id="@+id/progressText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Download Progress"
        android:paddingBottom="8dp" />

    <ProgressBar
        android:id="@+id/downloadProgressBar"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:progress="0"
        android:max="100" />
</LinearLayout>
```

The ProgressBar in the given XML layout is a horizontal progress bar defined with the style ?android:attr/progressBarStyleHorizontal. It spans the full width of the parent view (match_parent) and has a default height (wrap_content). The progress bar is initialized with a progress of 0 and a maximum value of 100, making it suitable for tasks like displaying download progress.

Code Java

```
package com.example.progressbardemo;

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

```
public class MainActivity extends AppCompatActivity {

    private ProgressBar downloadProgressBar;
    private TextView progressText;

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

        downloadProgressBar = findViewById(R.id.downloadProgressBar);
        progressText = findViewById(R.id.progressText);

        // Simulate download progress
        simulateDownload();
    }

    private void simulateDownload() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int progress = 0; progress <= 100; progress++) {
                    final int currentProgress = progress;
                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            downloadProgressBar.setProgress(currentProgress);
                            progressText.setText
                                    ("Download Progress: " + currentProgress + "%");
                        }
                    });

                    try {
                        Thread.sleep(100); // Simulate time taken for download
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();
    }
}
```

Output	Explanation
	This 'MainActivity' class in an Android application demonstrates the use of a 'ProgressBar' to simulate download progress. The 'ProgressBar' and a 'TextView' for displaying progress percentage are initialized in the 'onCreate' method by finding their respective IDs from the layout. The 'simulateDownload' method runs a separate thread to increment the progress value from 0 to 100, simulating the download process. For each increment, it updates the 'ProgressBar' and 'TextView' on the main UI thread using 'runOnUiThread'. The 'ProgressBar' visually shows the progress while the 'TextView' displays the current percentage of the download. The 'Thread.sleep(100)' call is used to simulate the time taken for each step of the download.

5.2.9 AutoCompleteTextView

'AutoCompleteTextView' is a UI component in Android that provides suggestions to users as they type. It extends from 'EditText' and offers an auto-completion feature, which displays a dropdown list of suggestions based on the text entered by the user. This component is useful in scenarios where users need to select from a predefined list of options, such as entering an email address, selecting a city, or typing a search query.

Features of AutoCompleteTextView

1. **User Input Assistance:** Provides suggestions to users as they type, making it easier to enter data quickly and accurately.
2. **Customizable Dropdown:** The dropdown list of suggestions can be customized in terms of appearance and behavior.
3. **Filterable Data Source:** Automatically filters the list of suggestions based on the user's input.
4. **Integration with Adapters:** Works seamlessly with adapters to populate the list of suggestions.

Key Attributes

Attribute Name	Description	Example	Explanation
android:completion Hint	Sets the hint text to be displayed when the drop-down list is shown.	android:completionHint ="Select a name"	Provides a hint to the user about the type of content expected in the drop-down list.

android:completion Threshold	Defines the minimum number of characters the user has to type before the drop-down list is shown.	android:completion Threshold="1"	Specifies the number of characters needed to trigger the drop-down list for suggestions.
android:dropDown Height	Sets the height of the drop-down list.	android:dropDown Height="200dp"	Controls the height of the suggestions drop-down list.
android:dropDown Width	Sets the width of the drop-down list.	android:dropDownWidth ="match_parent"	Controls the width of the suggestions drop-down list.
android:dropDown VerticalOffset	Sets the vertical offset of the drop-down list.	android:dropDownVertical Offset="10dp"	Adjusts the vertical positioning of the drop-down list relative to the 'AutoCompleteTextView'.
android:dropDown HorizontalOffset	Sets the horizontal offset of the drop-down list.	android:dropDownHorizontal Offset="10dp"	Adjusts the horizontal positioning of the drop-down list relative to the 'AutoCompleteTextView'.
android:popup Background	Sets the background drawable for the drop-down list.	android:popupBackground ="@drawable/popup_ background"	Allows customizing the background of the drop-down list with a drawable.
android:inputType	Sets the type of input expected (e.g., text, number, email).	android:inputType="text"	Defines the type of data expected, ensuring the appropriate keyboard is displayed for input.

Defining AutoCompleteTextView in XML

To begin working with 'AutoCompleteTextView' in Android Studio, create a new project and navigate to the layout file to work with the 'AutoCompleteTextView' element. By default, Android Studio creates a layout file named 'activity_main.xml' in the 'res/layout' folder. In the 'activity_main.xml' file, add the 'AutoCompleteTextView' element within a layout container, such as 'LinearLayout' or 'RelativeLayout', as shown below.

Code XML

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">
```

```
    <AutoCompleteTextView
        android:id="@+id/autoCompleteTextView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Type to search..."
        android:completionThreshold="1" />
</LinearLayout>
```

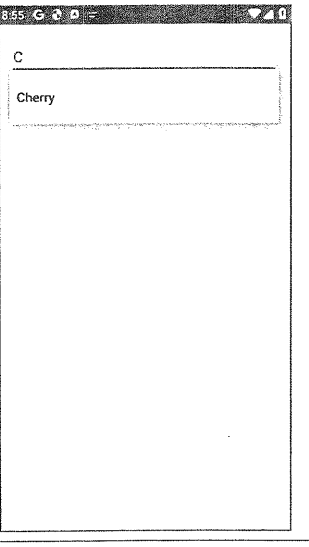
This XML code defines a vertical 'LinearLayout' containing an 'AutoCompleteTextView' view for user interaction. The 'AutoCompleteTextView' has a hint to guide the user and a completion threshold of 1 character.

Using AutoCompleteTextView in Activity Code

To use 'AutoCompleteTextView' in an activity code, reference the 'AutoCompleteTextView' element defined in the XML layout and interact with it in an activity.

Code Java

```
package com.example.autocompletedemo;

import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.AutoCompleteTextView;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private AutoCompleteTextView autoCompleteTextView;

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

        autoCompleteTextView = findViewById(R.id.autoCompleteTextView);

        // Sample data for AutoCompleteTextView
        String[] suggestions = {"Apple", "Banana", "Cherry",
                                "Date", "Elderberry", "Fig", "Grape", "Honeydew"};

        // Creating an ArrayAdapter with sample data
        ArrayAdapter<String> adapter = new ArrayAdapter<>
                    (this, android.R.layout.simple_dropdown_item_1line, suggestions);

        // Setting the adapter to the AutoCompleteTextView
        autoCompleteTextView.setAdapter(adapter);
    }
}
```

Explanation	Output
This code defines an Android activity that demonstrates interaction with an 'AutoCompleteTextView' view. In the 'onCreate' method, the 'AutoCompleteTextView' is initialized by finding it through its ID defined in the layout XML. An 'ArrayAdapter' is created with a sample array of suggestions, and the adapter is set to the 'AutoCompleteTextView' to provide auto-completion functionality.	

5.3 Using Picker Views

Picker views are interactive UI components that allow users to select a value or a set of values from a predefined range. They are often used in forms and settings where a user needs to choose from a list of options. Picker views enhance the user experience by providing a consistent and intuitive way to make selections.

Types of Picker Views

1. **DatePicker:** Allows users to select a specific date from a calendar or spinner view.
2. **TimePicker:** Enables users to choose a specific time, displayed in either 12-hour or 24-hour format.
3. **NumberPicker:** Lets users pick a number from a predefined range using a vertical scrollable list.
4. **Spinner:** Provides a dropdown menu for selecting a single item from a list of options.

Example	Scenarios of Using Picker Views

Picker Views in Android, such as TimePicker and DatePicker, are commonly used in various scenarios to allow users to select specific values or dates. Here are some generic example scenarios where Picker Views can be utilized in Android applications:

1. **Setting Alarm Time:** An alarm clock application can use a TimePicker to allow users to set the time for their alarms by selecting the hour and minute values.

2. **Selecting Birthdate:** In a user profile setup or registration form, a DatePicker can be used to let users choose their birthdate by selecting the day, month, and year.
3. **Booking Appointments:** A scheduling application can utilize a DatePicker to enable users to select a date for booking appointments with available time slots.
4. **Selecting Event Date and Time:** Event management applications can utilize both DatePicker and TimePicker to enable users to choose the date and time for creating or scheduling events.
5. **Selecting Age Range:** A filtering or search feature in an application can use NumberPickers to allow users to select age ranges or numerical values within a specific range.

Key Benefits of Using Picker Views

1. **Enhanced User Experience:** Picker views provide an intuitive and visually appealing way for users to select values.
2. **Improved Data Accuracy:** By constraining user input to predefined options, picker views help minimize errors and ensure that only valid data is entered.
3. **Efficiency in Data Entry:** Picker views allow for quick selection from a list of options to speed up the data entry process and reduce the need for manual typing.
4. **Customization and Flexibility:** Picker views can be extensively customized to fit the application's design and functionality requirements.
5. **Accessibility:** Built-in support for accessibility features ensures that picker views are usable by all users, including those with disabilities to enhance the overall usability of the application.

5.3.1 Date Picker

'DatePicker' is a UI component that allows users to select a specific date. It can display dates in various formats and allows navigation between days, months, and years.

Example Scenario: Scheduling an appointment, setting a birthday, or selecting a due date.

Features of DatePicker

1. **Date Selection:** Allows users to select a specific date from a calendar or spinner view.
2. **Customizable Display:** Can be displayed as a calendar or spinner, depending on the mode set (calendarViewShown and datePickerMode attributes).
3. **Min and Max Date:** Supports setting minimum and maximum selectable dates to restrict user input to a specific range.
4. **OnDateChangedListener:** Allows developers to handle date changes and perform actions when the selected date changes.

Defining DatePicker in XML Layout File

Code XML

```
<DatePicker
    android:id="@+id/datePicker"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:calendarViewShown="true" />
```

Explanation

This 'DatePicker' element is defined within an Android XML layout file. It has a unique identifier'@+id/datePicker', which allows it to be referenced in the corresponding Java or Kotlin code. The 'layout_width' and 'layout_height' attributes are set to 'wrap_content', meaning the 'DatePicker' will resize to fit its content. The 'android:calendarViewShown="true"' attribute specifies that a calendar view should be displayed, providing a user-friendly interface for selecting dates.

Using DatePicker in Activity Code

Code Java

```
DatePicker datePicker = findViewById(R.id.datePicker);
int day = datePicker.getDayOfMonth();
int month = datePicker.getMonth();
int year = datePicker.getYear();
```

Explanation

This code snippet interacts with a DatePicker view in an Android activity. The DatePicker is first located in the layout using findViewByld(R.id.datePicker) and assigned to the variable datePicker. The currently selected day, month, and year are then retrieved from the DatePicker using getDayOfMonth(), getMonth(), and getYear(), respectively, and stored in the corresponding integer variables. This allows the application to access and use the selected date for various purposes.

5.3.2 TimePicker

'TimePicker' is a UI component that allows users to select a specific time. It can be displayed in either 12-hour or 24-hour formats.

Example Scenario: Setting an alarm, scheduling a meeting, or selecting a reminder time.

Features of TimePicker

1. **Time Selection:** Allows users to select a specific time of day.
2. **12-Hour and 24-Hour Formats:** Can be configured to display time in either 12-hour (AM/PM) or 24-hour format using the is24HourView attribute.
3. **OnTimeChangedListener:** Allows developers to handle time changes and perform actions when the selected time changes.

Defining TimePicker in XML Layout File

Code XML

```
<TimePicker
    android:id="@+id/timePicker"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:timePickerMode="clock" />
```

Explanation

This TimePicker element is defined with a unique identifier @+id/timePicker, allowing it to be referenced in the corresponding Java or Kotlin code. The layout_width and layout_height attributes are set to wrap_content, meaning the TimePicker will resize to fit its content. The android:timePickerMode="clock" attribute specifies that the TimePicker should use a clock interface for time selection, providing a visually intuitive way for users to pick a time.

Using TimePicker in Activity Code

Code Java

```
TimePicker timePicker = findViewById(R.id.timePicker);
int hour = timePicker.getHour();
int minute = timePicker.getMinute();
```

Explanation

This code snippet demonstrates how to interact with a 'TimePicker' view in an Android activity. The 'TimePicker' is first located in the layout using 'findViewById(R.id.timePicker)' and assigned to the variable 'timePicker'. The currently selected hour and minute are then retrieved from the 'TimePicker' using the 'getHour()' and 'getMinute()' methods, respectively, and stored in the integer variables 'hour' and 'minute'. This allows the application to access and use the selected time for various purposes.

5.3.3 NumberPicker

'NumberPicker' is a UI component that allows users to select a number from a predefined range. It displays a vertical list of numbers that can be scrolled.

Example Scenario: Selecting a quantity of items to purchase, setting a volume level, or choosing a rating.

Features of NumberPicker

1. **Number Range Selection:** Allows users to select a number from a predefined range using a vertical scrollable list.
2. **Min and Max Values:** Supports setting minimum and maximum values to define the range of selectable numbers.
3. **Customizable Formatter:** Allows customization of the displayed values using a formatter.
4. **Wrap Selector Wheel:** Option to enable or disable the wrapping of the selector wheel, allowing values to wrap around from min to max and vice versa.

Defining NumberPicker in XML Layout File

Code XML

```
<NumberPicker
    android:id="@+id/numberPicker"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

Explanation

This 'NumberPicker' element is defined within an Android XML layout file. It has a unique identifier '@+id/numberPicker', allowing it to be referenced in the corresponding Java or Kotlin code. The 'layout_width' and 'layout_height' attributes are set to 'wrap_content', meaning the 'NumberPicker' will resize to fit its content.

Using NumberPicker in Activity Code

Code Java

```
numberPicker = findViewById(R.id.numberPicker);
numberPicker.setMinValue(1);
numberPicker.setMaxValue(10);
```

Explanation

This code snippet demonstrates how to configure a 'NumberPicker' in an Android activity. The 'NumberPicker' is first located in the layout using 'findViewById(R.id.numberPicker)' and assigned to the variable 'numberPicker'. The 'setMinValue(1)' method sets the minimum selectable value to 1, and the 'setMaxValue(10)' method sets the maximum selectable value to 10. This configuration allows users to pick a number between 1 and 10 from the 'NumberPicker'.

Example : To Demonstrate DatePicker, TimePicker, and NumberPicker

This example will show how to use DatePicker, TimePicker, and NumberPicker in an Android application. We will provide the XML layout and the corresponding Java code to handle interactions with these picker views.

Code XML Layout

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp">
```

```
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Select a Date"
        android:textSize="16sp"
        android:paddingBottom="4dp" />

    <DatePicker
        android:id="@+id/datePicker"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:calendarViewShown="true" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Select a Time"
        android:textSize="16sp"
        android:paddingTop="8dp"
        android:paddingBottom="4dp" />

    <TimePicker
        android:id="@+id/timePicker"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:timePickerMode="clock"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Select a Number"
        android:textSize="16sp"
        android:paddingTop="8dp"
        android:paddingBottom="4dp" />

    <NumberPicker
        android:id="@+id/numberPicker"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

  </LinearLayout>
</ScrollView>
```

Explanation

This XML layout defines a user interface within a 'ScrollView' to ensure all content is scrollable to fit on the screen. Inside the 'ScrollView' is a vertical 'LinearLayout' with padding of 16dp. It contains three sections, each introduced by a 'TextView' label. The first section has a 'DatePicker' for selecting a date with the calendar view enabled. The second section includes a 'TimePicker' for selecting a time, using the clock mode. The third section contains a 'NumberPicker' for selecting a number, which will be configured programmatically. Each section is spaced out with padding to ensure a clean and readable layout.

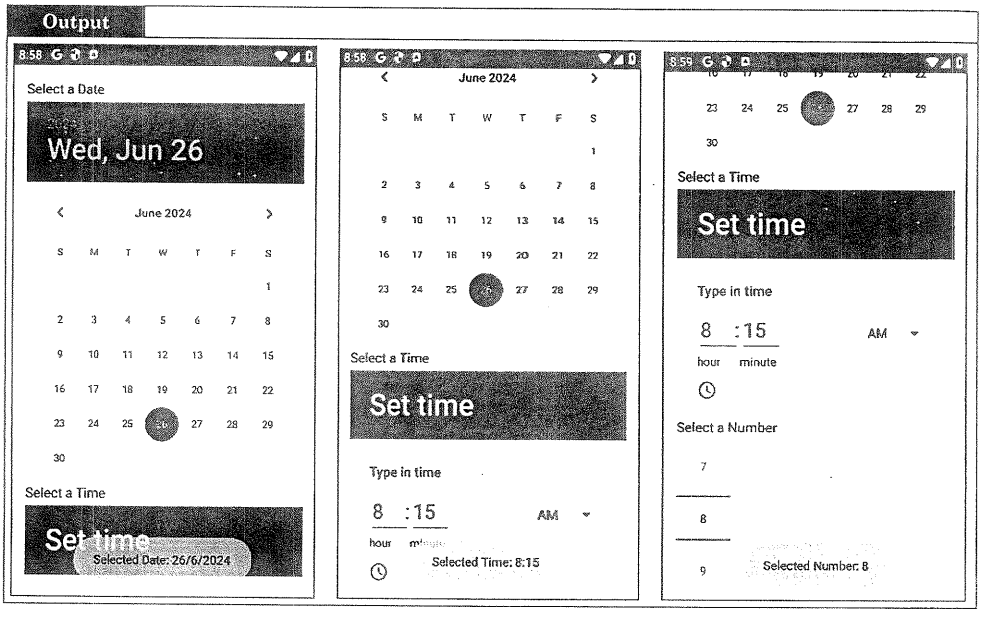
Code Java

```
package com.example.viewdemo;

import android.os.Bundle;
import android.widget.DatePicker;
import android.widget.NumberPicker;
import android.widget.TimePicker;

import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private DatePicker datePicker;
    private TimePicker timePicker;
    private NumberPicker numberPicker;

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

        // Initialize DatePicker
        datePicker = findViewById(R.id.datePicker);
        datePicker.init(datePicker.getYear(), datePicker.getMonth(),
            datePicker.getDayOfMonth(), new DatePicker.OnDateChangedListener() {
            @Override
            public void onDateChanged(DatePicker view, int year,
                                      int monthOfYear, int dayOfMonth) {
                Toast.makeText(MainActivity.this, "Selected Date: " + dayOfMonth +
                    "/" + (monthOfYear + 1) + "/" + year, Toast.LENGTH_SHORT).show();
            }
        });
```

```
        // Initialize TimePicker
        timePicker = findViewById(R.id.timePicker);
        timePicker.setOnTimeChangedListener(new TimePicker.OnTimeChangedListener() {
            @Override
            public void onTimeChanged(TimePicker view, int hourOfDay, int minute) {
                Toast.makeText(MainActivity.this, "Selected Time: " + hourOfDay +
                        ":" + minute, Toast.LENGTH_SHORT).show();
            }
        });

        // Initialize NumberPicker
        numberPicker = findViewById(R.id.numberPicker);
        numberPicker.setMinValue(1);
        numberPicker.setMaxValue(10);
        numberPicker.setOnValueChangedListener
                    (new NumberPicker.OnValueChangeListener() {
           @Override
            public void onValueChange(NumberPicker picker, int oldVal, int newVal) {
                Toast.makeText(MainActivity.this, "Selected Number: " +
                                newVal, Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

Explanation

- This Java code snippet demonstrates how to initialize and handle interactions with a 'DatePicker', 'TimePicker', and 'NumberPicker' in an Android activity. The 'DatePicker' is initialized with 'findViewById' and an 'OnDateChangedListener' is set to display a 'Toast' message showing the selected date whenever the date changes.
- The 'TimePicker' is similarly initialized with 'findViewById' and an 'OnTimeChangedListener' is set to display a 'Toast' message with the selected time in the format "hour:minute" whenever the time changes, allowing users to see their selected time instantly.
- The 'NumberPicker' is initialized by finding it with its ID, setting its minimum and maximum values to 1 and 10, respectively, and setting an 'OnValueChangedListener' to show a 'Toast' message displaying the selected number whenever it changes, providing real-time feedback to the user.



5.4 Using List Views to Display Long Lists

List views in Android are essential components designed to display long lists of items in a scrollable format to provide a structured and user-friendly way to manage extensive datasets. Using ListViews to display long lists is a common practice in Android app development, especially when we need to show a large amount of data that exceeds the screen size.

There are two primary types of list views:

1. **ListView :** ListView is ideal for displaying a vertically scrollable list of items where users can see multiple items at once and select any of them such as contact lists or message inboxes.
2. **RecyclerView:** The RecyclerView is a more advanced and flexible version of the ListView. It is a container for displaying large data sets that can be scrolled efficiently by maintaining a limited number of views. The RecyclerView uses a flexible view holder pattern to improve performance by recycling views that are no longer visible.
3. **SpinnerView :** SpinnerView provides a dropdown menu interface that allows users to select one item from a list for scenarios where space is limited and a single selection is needed, like choosing a country from a list.
4. **GridView:** The GridView displays items in a two-dimensional, scrollable grid.It is useful for displaying items in a grid layout such as a gallery of images or a grid of products.

Example | Generic Scenarios for Using List Views

1. **ListView:** In a contacts application, a ListView can display a list of contacts with names, phone numbers, and profile pictures. Users can scroll through the list to find and select a specific contact for viewing or editing.
2. **RecyclerView:** An e-commerce application can utilize a RecyclerView with a GridLayoutManager to display a grid of product items. Each item in the grid can include product images, names, prices, and possibly ratings. Users can scroll through the grid to explore different products and make selections for purchase. The RecyclerView efficiently handles the loading and recycling of views, providing a smooth scrolling experience for users.
3. **SpinnerView:** In a user registration form, a SpinnerView can be used to present a dropdown list of countries for users to select their country of residence. This saves space on the screen and simplifies the selection process for the user.
4. **GridView:** An e-commerce application can use a GridView to display a grid of product items with images, names, and prices. Users can browse through the grid layout to view and select products they are interested in purchasing.choose their preferred payment method from a predefined list of options.

5.4.1 ListView

ListView is a fundamental component in Android development that allows developers to display a scrollable list of items on the screen. It offers various features and functionalities to enhance the user experience and facilitate the presentation of data in a structured manner. It is ideal for presenting large datasets in a structured format to navigate and interact with the data efficiently and intuitively. The ListView is particularly beneficial when the number of items to be displayed is not known beforehand or when the data is dynamic.

Key Concepts for Using ListView in Android

Understanding the following key concepts is essential for effectively using 'ListView' in Android applications. These concepts helps in efficiently managing data and creating user-friendly interfaces.

1. **Adapters:**

   Adapters act as a bridge between the data source and the 'ListView'. They convert data items from a data source into viewable components that can be displayed in the list. They are responsible for managing the data and creating views to represent the data items on the screen.

   Think of an adapter as a translator between a book (data) written in one language (data source) and a reader (UI component) who understands a different language. The adapter translates the content of the book into a language that the reader can undersatnd and presents it in a readable format.

   Example: Imagine a list of fruits: "Apple," "Banana," "Cherry." An adapter takes each fruit name and creates a visual representation (like a 'TextView' with the fruit name) that can be displayed in the 'ListView'.

Types:

- 'ArrayAdapter': Best for simple lists of text items.
- 'CursorAdapter': Used for data from database queries.
- 'SimpleAdapter': Suitable for more complex data with multiple views per item.

2. **Data Sources:**

Data sources are the origin of the information that needs to be displayed in the UI. They can be arrays, lists, databases, or network sources containing the data to be shown.

Consider a recipe app where the list of recipes (data) is stored in a recipe book (data source). The data source provides the recipes to the app, which then uses an adapter to display them in a user-friendly manner on the screen.

Example: For displaying a list of employee names, the data source could be an array like '["Srikanth", "Nirmala", "Mary Margeret"]' or a list of 'Employee' objects, each containing details about an employee.

Common Sources:

- Arrays or Lists of simple data (like strings or integers).
- Lists of complex objects (like custom 'Employee' or 'Product' classes).
- Database queries that return a cursor with multiple rows of data.

3. **View Recycling:**

View recycling is a technique used to optimize the performance of displaying long lists by reusing view elements that are not currently visible on the screen. This helps in saving memory and improving scrolling efficiency. View recycling is a technique used to optimize memory usage and performance in 'ListView'. Instead of creating a new view for each item, 'ListView' reuses views that have gone off-screen. This minimizes the number of view objects created and reduces memory consumption.

Let us consider a viewing contacts in a mobile. When the ListView is first loaded, it creates views (UI elements) to display the contacts visible on the screen. As the user scrolls down, the top contacts move off the screen, and new contacts appear at the bottom. Instead of creating new views for the new contacts that come into view, the ListView recycles the views of the contacts that moved off-screen at the top. The recycled views are updated with the information of the new contacts that are now visible at the bottom of the list.

Example: Scrolling through a long list of messages without view recycling would create a new 'TextView' for each message, which is inefficient. With view recycling, 'ListView' reuses 'TextView' objects for new messages as scrolling occurs to improve the performance.

4. **Custom Adapters:**

Custom adapters allow developers to create specialized adapters tailored to specific requirements, such as custom layouts or data processing logic. They provide flexibility in how data is presented in the UI. Customization of the data binding process to handle complex data and layouts is possible by extending base adapter classes.

In a social media app, a custom adapter can be designed to display user profiles with unique layouts, including profile pictures, usernames, and bio information. This custom adapter can handle the data binding and view creation based on the specific design requirements of the user profiles.

Example: Displaying a list of products, each with a name, price, and image requires a custom adapter to inflate a custom layout for each item to ensure all these elements are displayed correctly.

5. **Data Binding:**

   Data binding in 'ListView' refers to the process of connecting the data source to the views defined in the item layout. It establishes a link between the data and the visual elements on the screen. The adapter handles this process to convert each data item into a view that can be displayed.

   Example : For a contacts list, data binding involves taking each 'Contact' object (with fields like name and phone number) and binding these fields to the corresponding views ('TextView' for the name, another 'TextView' for the phone number) in the item layout.

Key Features of ListView

1. **Efficient Data Management:**

   'ListView' handles large datasets efficiently by reusing view elements that have gone off-screen, a process known as view recycling. This technique reduces memory usage and improves performance, ensuring a smooth scrolling experience even with extensive data. The 'getView' method of the adapter plays a crucial role in view recycling by either reusing an existing view or creating a new one if necessary.

2. **Customizable Item Layouts:**

   Each item in a 'ListView' can be customized using XML layout files. Developers can define complex item layouts that include various UI elements such as text, images, buttons, and more. This flexibility allows for the creation of visually appealing and informative list items. Custom layouts are defined in separate XML files and are inflated by the adapter to create individual list items.

3. **Adapters:**

   The 'ListView' relies on adapters to bind data to the view. Adapters act as intermediaries between the data source and the 'ListView', converting data items into viewable components. Common adapters include 'ArrayAdapter', 'CursorAdapter', and 'SimpleAdapter', each suited for different types of data sources. The adapter handles creating the views for each list item and binding data to them.

4. **Event Handling:**

   'ListView' provides built-in support for handling user interactions such as item clicks, long presses, and contextual menus. Developers can set listeners to respond to these events for interactive and dynamic user interfaces. For example, 'setOnItemClickListener' and 'setOnItemLongClickListener' are used to handle item click and long press events, respectively.

5. **Data Binding:**

'ListView' can be easily bound to data sources such as arrays, lists, or even database queries. Adapters take care of the data binding process, converting each data item into a view that can be displayed in the list. This data binding capability simplifies the process of managing and displaying dynamic content. 'ListView' is a powerful tool for applications that require frequent data updates.

Step by Step Implemenation of ListView

This example demonstrates how to implement a 'ListView' in an Android application to display a simple contact list.

Step 1: Define the Data Source:

The data source contains the items that will be displayed in the 'ListView'.

Code

```
String[] contacts = {"Bhagya", "Snigdha", "Srikanth", "Srinath", "Rakshith",
"Manjula","Mary","Nirmala","Raghavendra","Eranna","Yamuna","Smayan"};
```

Explanation

This array of strings represents the contact names to be displayed in the 'ListView'. Each string in the array corresponds to a contact name.

Step 2: Create the Layout for List Items:

Define an XML layout file for the individual items that will be displayed in the 'ListView'. This layout file specifies how each item will look.

Code res/layout/list_item.xml

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/textView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="16dp"
    android:textSize="18sp" />
```

Explanation

- **list_item.xml :** Defines the layout for each item in the 'ListView'. It contains a single 'TextView' which will display the contact name. The 'TextView' is given a unique ID ('@+id/textView'), with its width and height set to match the parent and wrap content, respectively. Padding is added for better spacing, and the text size is set to 18sp.

Step 3: Create the Layout for the Activity:

Define the XML layout file for the activity that contains the 'ListView'.

Code res/layout/activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ListView
        android:id="@+id/listView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Explanation

- **activity_main.xml:** Sets up the main layout for the activity. It uses a 'LinearLayout' with vertical orientation and includes a 'ListView' identified by '@+id/listView', which will display the contact names.

Step 4: Set Up the Adapter:

Choose an appropriate adapter and bind it to the data source. The adapter converts data items into viewable components that can be displayed in the 'ListView'.

Code

```
ArrayAdapter<String> adapter = new ArrayAdapter<>(this, R.layout.list_item,
R.id.textView, contacts);
```

Explanation

An 'ArrayAdapter' is created to link the data source ('contacts') with the 'ListView'. It specifies the context, the item layout ('R.layout.list_item'), and the 'TextView' ID ('R.id.textView') where the data will be displayed.

Step 5: Bind the Adapter to the ListView:

Attach the adapter to the 'ListView' to start displaying the data.

Code

```
ListView listView = findViewById(R.id.listView);
listView.setAdapter(adapter);
```

Explanation

The 'ListView' is retrieved using 'findViewById' and the adapter is set to the 'ListView' using 'setAdapter', binding the contact names to the list.

Step 6: Handle User Interactions (Optional):

Set up event listeners to handle user interactions, such as clicks on list items.

Code

```
listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        String contact = (String) parent.getItemAtPosition(position);
        Toast.makeText(MainActivity.this, "Clicked: " + contact, Toast.LENGTH_
SHORT).show();
    }
});
```

Explanation

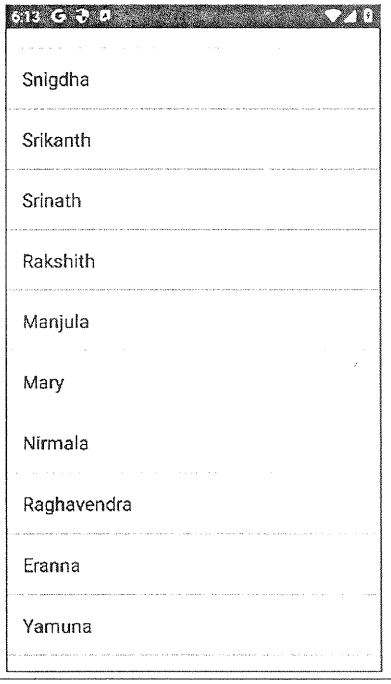
An 'OnItemClickListener' is set on the 'ListView' to respond to item clicks. When an item is clicked, it retrieves the contact name and displays a toast message with the name.

Code **Complete Code for MainActivity.java**

```
package com.example.contactlist;

import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.Toast;
import android.view.View;
import android.widget.AdapterView;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        // Define the data source
        String[] contacts = {"Bhagya", "Snigdha", "Srikanth", "Srinath",
                             "Rakshith", "Manjula","Mary","Nirmala","Raghavendra",
                             "Eranna","Yamuna","Smayan"};

       // Create the adapter
        ArrayAdapter<String> adapter = new ArrayAdapter<>
                            (this, R.layout.list_item, R.id.textView, contacts);

       // Bind the adapter to the ListView
        ListView listView = findViewById(R.id.listView);
        listView.setAdapter(adapter);
```

```
        // Handle item clicks
        listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view,
                                    int position, long id) {
                String contact = (String) parent.getItemAtPosition(position);
                Toast.makeText(MainActivity.this, "Clicked: " + contact,
                               Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

Output



Explanation

The 'MainActivity' class sets up the 'ListView' to display a list of contact names. It starts by defining the data source (an array of contact names). The 'ArrayAdapter' is then created to bind the data to the 'ListView' using the item layout ('list_item.xml') and the 'TextView' within it. The adapter is attached to the 'ListView', and an 'OnItemClickListener' is set up to handle clicks on the list items, displaying a toast message with the clicked contact name.

5.4.2 RecyclerView

'RecyclerView' is a flexible and efficient component introduced in Android as a more advanced and versatile version of the 'ListView' and 'GridView'. It is used to display large sets of data in a scrollable layout. The 'RecyclerView' is designed to handle long lists of data efficiently, with features like view recycling and view holders to improve performance and memory usage.

Key Concepts of RecyclerView

1. **ViewHolder Pattern:**

   The 'ViewHolder' pattern is used to minimize the number of calls to 'findViewById'. A 'ViewHolder' holds references to the UI components of a single item view, which makes it easy to recycle views efficiently.

2. **Adapter:**

   The adapter in 'RecyclerView' binds the data to the item views. It creates new 'ViewHolder' objects as needed and binds data to them.

3. **LayoutManager:**

   The 'LayoutManager' is responsible for measuring and positioning item views within a 'RecyclerView'. It determines how items are laid out, such as in a linear list or a grid.

Key Features of RecyclerView

1. **Efficient View Recycling:**

   RecyclerView efficiently recycles views that are no longer visible, reducing the number of views created and enhancing performance, especially when dealing with large data sets.

2. **ViewHolder Pattern:**

   The ViewHolder pattern is used to minimize the number of calls to findViewById. A ViewHolder holds references to the UI components of a single item view, making it easy to recycle views efficiently.

3. **Flexible Layout Management:**

   RecyclerView uses LayoutManager classes to manage the positioning and layout of items. This allows for flexible laycuts such as linear lists, grids, and staggered grids, enhancing the adaptability of the user interface.

4. **Built-in Animations:**

   RecyclerView supports built-in animations for adding and removing items, providing a smoother and more visually appealing user experience.

Step by Step Implemenation of RecyclerView

Step 1: Create the Layout for RecyclerView:

Define an XML layout file for the activity that contains the 'RecyclerView'.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">
```

```
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Explanation

- **activity_main.xml:** Sets up the main layout for the activity. It uses a 'LinearLayout' with vertical orientation and includes a 'RecyclerView' identified by"@+id/recyclerView'.

Step 2: Create the Layout for Individual Items:

Define an XML layout file for the individual items that will be displayed in the 'RecyclerView'.

Code (res/layout/item_layout.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/textView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="16dp"
    android:textSize="18sp" />
```

Explanation

- item_layout.xml: Defines the layout for each item in the 'RecyclerView'. It contains a single 'TextView' which will display the data.

Step 3: Create the ViewHolder Class:

Create a 'ViewHolder' class that holds references to the UI components of each item view.

Code MyViewHolder.java

```
package com.example.recyclerviewdemo;

import android.view.View;
import android.widget.TextView;

import androidx.recyclerview.widget.RecyclerView;

public class MyViewHolder extends RecyclerView.ViewHolder {
    TextView textView;

    public MyViewHolder(View itemView) {
        super(itemView);
        textView = itemView.findViewById(R.id.textView);
    }
}
```

Explanation
• **MyViewHolder:** Extends 'RecyclerView.ViewHolder' and holds references to the UI components of the item view, which in this case is a 'TextView'.

Step 4: Create the Adapter Class:

Create an adapter class that binds the data to the item views.

Code MyAdapter.java

```
package com.example.recyclerviewdemo;

import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

import androidx.recyclerview.widget.RecyclerView;

public class MyAdapter extends RecyclerView.Adapter<MyViewHolder> {
    private String[] data;

    public MyAdapter(String[] data) {
        this.data = data;
    }

    @Override
    public MyViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext()).
                                    inflate(R.layout.item_layout, parent, false);
        return new MyViewHolder(view);
    }

    @Override
    public void onBindViewHolder(MyViewHolder holder, int position) {
        holder.textView.setText(data[position]);
    }

    @Override
    public int getItemCount() {
        return data.length;
    }
}
```

Explanation
• **MyAdapter:** Extends 'RecyclerView.Adapter' and binds the data to the item views. It overrides the necessary methods: 'onCreateViewHolder', 'onBindViewHolder', and 'getItemCount'.

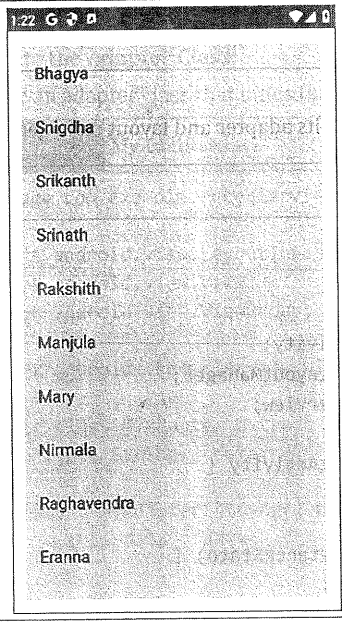
- **onCreateViewHolder:** Inflates the item layout and creates a 'ViewHolder'.
- **onBindViewHolder:** Binds the data to the 'ViewHolder'.
- **getItemCount:** Returns the size of the data set.

Step 5: Set Up the RecyclerView in the Activity:

Initialize the 'RecyclerView' in the activity and set its adapter and layout manager.

Code MainActivity.java

```
package com.example.recyclerviewdemo;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

public class MainActivity extends AppCompatActivity {

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

        // Define the data source
        String[] contacts = {"Bhagya", "Snigdha", "Srikanth", "Srinath",
                             "Rakshith", "Manjula","Mary","Nirmala",
                             "Raghavendra","Eranna","Yamuna","Smayan"};

        // Find the RecyclerView
        RecyclerView recyclerView = findViewById(R.id.recyclerView);

        // Set the LayoutManager
        recyclerView.setLayoutManager(new LinearLayoutManager(this));

        // Create the adapter
        MyAdapter adapter = new MyAdapter(contacts);

        // Bind the adapter to the RecyclerView
        recyclerView.setAdapter(adapter);
    }
}
```

Output	Explanation
	• **MainActivity:** Sets up the 'RecyclerView' to display a list of contact names. It starts by defining the data source (an array of contact names). The 'RecyclerView' is initialized, and a 'LinearLayoutManager' is set to manage the layout of the items. The adapter is created and set on the 'RecyclerView'.

5.4.3 SpinnerView

The ListView displays a long list of items in an activity, but sometimes we may want user interface to display other views, and hence we do not have the additional space for a full-screen view like the ListView. In such cases, we should use the SpinnerView.

The 'SpinnerView' commonly referred to simply as 'Spinner' is a view in Android that provides a quick way to select one value from a list. It displays the currently selected item and, upon user interaction, shows a dropdown menu with all other available values from which the user can choose. This is particularly useful for presenting a list of choices without taking up much screen space.

Key Features of SpinnerView

1. **Compact Display:** The 'Spinner' displays the selected item in a single line. When clicked, it shows the full list of options in a dropdown menu, making it an efficient use of screen space.
2. **Data Binding:** Similar to 'ListView', 'Spinner' uses adapters to bind data from a source to the view. This allows for flexible and dynamic data presentation.
3. **User Interaction:** The 'Spinner' is designed to handle user selection events to provide a simple and intuitive way for users to select from a list of options.
4. **Customizable Layout:** The appearance of both the selected item and the dropdown list can be customized through XML layouts and adapter configurations.

Example : Demonstrating SpinnerView

Step 1: Define the Data Source:

The data source contains the items that will be displayed in the 'Spinner'. This can be a simple array, a list of objects, or data fetched from a database.

Code

```
String[] countries = {"India", "USA", "China", "UK", "Australia"};
```

Explanation

This array of strings represents the items to be displayed in the 'Spinner'. Each string in the array corresponds to a country name.

Step 2: Create the Layout for the Spinner:

Define an XML layout file for the activity that contains the 'Spinner'.

Code (res/layout/activity_main.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <Spinner
        android:id="@+id/spinner"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:spinnerMode="dropdown" />
</LinearLayout>
```

Explanation

- **activity_main.xml:** Sets up the main layout for the activity. It uses a 'LinearLayout' with vertical orientation and includes a 'Spinner' identified by '@+id/spinner'. The 'spinnerMode' attribute is set to 'dropdown', indicating that the spinner should display a dropdown menu when clicked.

Step 3: Set Up the Adapter:

Choose an appropriate adapter and bind it to the data source. The adapter converts data items into viewable components that can be displayed in the 'Spinner'.

Code

```
ArrayAdapter<String> adapter = new ArrayAdapter<>(this, android.R.layout.simple_
spinner_item, countries);
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
```

Explanation

- **ArrayAdapter :** A simple adapter that is used when the data source is an array or list of objects.
- **android.R.layout.simple_spinner_item:** A built-in layout resource for displaying the selected item.
- **setDropDownViewResource:** Specifies the layout resource to use for each item in the dropdown list. In this case, 'android.R.layout.simple_spinner_dropdown_item' is used.

Step 4: Bind the Adapter to the Spinner:

Attach the adapter to the 'Spinner' to start displaying the data.

Code

```
Spinner spinner = findViewById(R.id.spinner);
spinner.setAdapter(adapter);
```

Explanation

The 'Spinner' is retrieved using 'findViewById' and the adapter is set to the 'Spinner' using 'setAdapter', binding the country names to the spinner.

Step 5: Handle User Interactions (Optional):

Set up event listeners to handle user interactions, such as selection changes.

Code

```
spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
    @Override
    public void onItemSelected(AdapterView<?> parent, View view,
                               int position, long id) {
        String selectedCountry = parent.getItemAtPosition(position).toString();
        Toast.makeText(MainActivity.this, "Selected: " + selectedCountry,
                       Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onNothingSelected(AdapterView<?> parent) {
        // Do nothing
    }
});
```

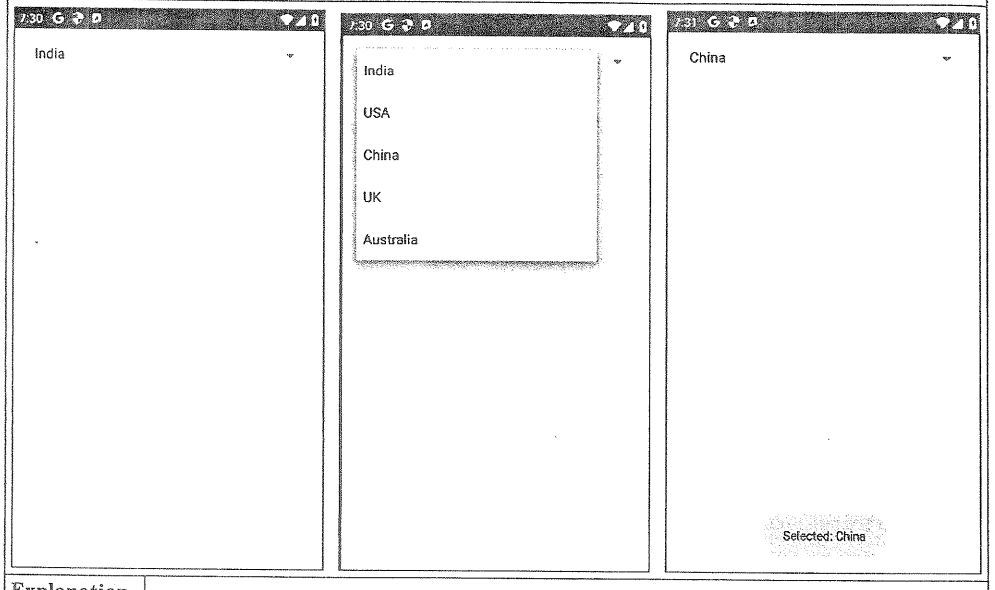
Explanation

- **setOnItemSelectedListener :** Sets a listener to respond to item selection events.
- **onItemSelected :** This method is called when an item is selected in the 'Spinner'.
- **position :** The position of the selected item in the list.
- **parent.getItemAtPosition(position):** Retrieves the data item at the specified position.
- **Toast :** Displays a short message to the user.

Code Complete Code for MainActivity.java

```
package com.example.spinnerdemo;

import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.Toast;
import android.view.View;
import android.widget.AdapterView;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        // Define the data source
        String[] countries = {"India", "USA", "China", "UK", "Australia"};

        // Create the adapter
        ArrayAdapter<String> adapter = new ArrayAdapter<>
                        (this, android.R.layout.simple_spinner_item, countries);
        adapter.setDropDownViewResource(android.R.layout.
                                        simple_spinner_dropdown_item);

        // Bind the adapter to the Spinner
        Spinner spinner = findViewById(R.id.spinner);
        spinner.setAdapter(adapter);

        // Handle item selections
        spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
            @Override
            public void onItemSelected(AdapterView<?> parent, View view,
                                       int position, long id) {
              String selectedCountry = parent.getItemAtPosition(position).toString();
              Toast.makeText(MainActivity.this, "Selected: " +
                             selectedCountry, Toast.LENGTH_SHORT).show();
            }

            @Override
            public void onNothingSelected(AdapterView<?> parent) {
                // Do nothing
            }
        });
    }
}
```

Output



Explanation

The 'MainActivity' class sets up the 'Spinner' to display a list of country names. It starts by defining the data source (an array of country names). The 'ArrayAdapter' is then created to bind the data to the 'Spinner' using a built-in layout for both the selected item and the dropdown items. The adapter is attached to the 'Spinner', and an 'OnItemSelectedListener' is set up to handle selection changes, displaying a toast message with the selected country name.

5.5 Understanding Specialized Fragments

In Chapter 3, the fragment feature i was discussed. Fragments allow for the customization of the user interface in Android applications by dynamically adjusting fragments to fit within an activity. This flexibility is particularly useful for developing applications that can adapt to various screen sizes.

Fragments can be viewed as "mini-activities" with their own life cycles. To create a fragment, a class extending the Fragment base class is required. In addition to the Fragment base class, there are three subclasses - **ListFragment**, **DialogFragment**, and **PreferenceFragment** - that can be extended to create specialized fragments.

5.5.1 Using a ListFragment

ListFragment is a specialized fragment provided by Android that simplifies the implementation of a fragment containing a list view. It is an extension of the Fragment class and provides built-in support for displaying a list of items, handling item clicks, and managing data binding to the list.

It is designed to simplify the process of displaying a list within a Fragment. ListFragment encapsulates the behavior of a ListView within a Fragment to provide a more modular and reusable approach to displaying lists in Android applications. ListFragment has its own lifecycle methods similar to an Activity or Fragment for easier management of list-related operations. ListFragment can be used to create modular and reusable components that display lists of data within different parts of an application.

Key Features of a ListFragment

1. **Encapsulation of ListView:**

   ListFragment encapsulates a ListView as its primary layout element, simplifying the process of displaying lists within a Fragment. This encapsulation promotes modular design and reusability of list-related functionality.

2. **Lifecycle Management:**

   ListFragment has its own lifecycle methods, similar to an Activity or Fragment, allowing for easier management of list-related operations. This includes methods such as onCreate(), onCreateView(), onStart(), onResume(), onPause(), onStop(), onDestroy(), etc.

3. **Built-in Empty View Handling:**

   ListFragment provides built-in support for handling empty lists by displaying a default empty view when the list has no items. This feature simplifies the implementation of empty state handling in list-based UIs.

4. **Adapter Integration:**

   ListFragment seamlessly integrates with adapters (such as ArrayAdapter or CursorAdapter) to bind data to the ListView. This integration simplifies the process of populating the list with data and updating it when the underlying dataset changes.

5. **Modular and Reusable Components:**

   ListFragment allows developers to create modular and reusable components that display lists of data within different parts of an application. By encapsulating list functionality within a Fragment, ListFragment promotes code reusability and maintainability.

Step-by-Step Implementation of a ListFragment

Step 1: Set Up the Project:

Create an Android project using empty views activity and name it as testdemo.

Step 2: Create the ListFragment Class

1. **Add a New Java Class:**
   - In the project structure, right-click on the 'java' package (e.g., 'com.example.testdemo').
   - Select 'New > Java Class'.
   - Name the class 'FruitListFragment'.
   - Click "OK".

2. Add the below code for FruitListFragment.java

Code FruitListFragment.java

```
package com.example.testdemo;

import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.Toast;
import androidx.fragment.app.ListFragment;

public class FruitListFragment extends ListFragment {

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        // Sample data for the list
        String[] fruits = {"Apple", "Banana", "Cherry", "Date", "Elderberry"};

        // Create an ArrayAdapter to bind the data to the ListFragment
        ArrayAdapter<String> adapter = new ArrayAdapter<>(getActivity(),
                android.R.layout.simple_list_item_1, fruits);
        setListAdapter(adapter);
    }

    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        super.onListItemClick(l, v, position, id);

        // Get the selected item
        String selectedFruit = (String) getListAdapter().getItem(position);

        // Display a toast message with the selected item
        Toast.makeText(getActivity(), "Selected: " +
                        selectedFruit, Toast.LENGTH_SHORT).show();
    }
}
```

Explanation

- FruitListFragment: Extends 'ListFragment' to display a list of fruits.
- onActivityCreated: This method is called when the fragment's activity has been created. An 'ArrayAdapter' is created to bind the list of fruits to the fragment.
- onListItemClick: This method handles item click events and displays a toast message with the selected fruit.

Step 3: Update the Activity Layout - activity_main.xml:

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment
        android:id="@+id/fragment_container"
        android:name="com.example.testdemo.FruitListFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</FrameLayout>
```

Explanation

- **FrameLayout:** Acts as a container for the fragment.
- **Fragment Tag:** Includes the 'FruitListFragment' directly in the layout.

Step 4: Update the MainActivity.java:

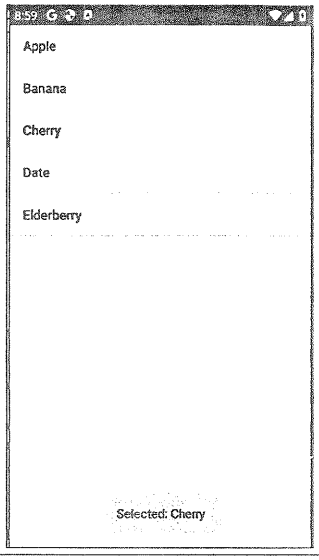
Code MainActivity.java

```
package com.example.testdemo;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

Explanation

The 'MainActivity' sets the content view to the 'activity_main.xml' layout, which includes the 'FruitListFragment'.

Output



5.5.2 Using a DialogFragment

'DialogFragment' is a specialized fragment in Android that simplifies the implementation of dialogs within an application. Extending the 'Fragment' class, 'DialogFragment' provides built-in support for displaying dialogs, handling dialog events, and managing data binding within the dialog. The primary aim of 'DialogFragment' is to encapsulate the behavior of a dialog within a fragment, thereby promoting modular and reusable design for dialog-based interactions in Android applications.

Key Features of DialogFragment

1. **Encapsulation of Dialog:**

   'DialogFragment' encapsulates a dialog as its primary layout element. This encapsulation streamlines the process of displaying dialogs within a Fragment and promotes modular design and reusability of dialog-related functionality.

2. **Lifecycle Management:**

   'DialogFragment' includes its own lifecycle methods, similar to those of an Activity or Fragment. These methods facilitate easier management of dialog-related operations, such as 'onCreateDialog()', 'onStart()', 'onResume()', 'onPause()', 'onStop()', and 'onDestroyView()'.

3. **Built-in Dialog Handling:**

   'DialogFragment' offers built-in support for creating and managing dialogs, including custom dialogs. This feature simplifies the implementation of dialog interactions within Android applications.

4. **Adapter Integration:**

   'DialogFragment' can integrate with adapters to bind data to the dialog. This integration simplifies the process of populating the dialog with data and updating it when the underlying dataset changes.

5. **Modular and Reusable Components:**

   'DialogFragment' facilitates the creation of modular and reusable components that display dialogs within various parts of an application. By encapsulating dialog functionality within a Fragment, 'DialogFragment' enhances code reusability and maintainability.

Step-by-Step Implementation of DialogFragment

Step 1: Set Up the Project:

An Android project can be created using an empty activity and named 'DialogDemo'.

Step 2: Create the DialogFragment Class:

1. **Add a New Java Class:**
   - Within the project structure, right-click on the 'java' package (e.g., 'com.example.dialogdemo').
   - Select 'New > Java Class'.
   - Name the class 'MyDialogFragment'.
   - Click "OK".
2. **Add the below code for 'MyDialogFragment.java'**

Code MyDialogFragment.java

```
package com.example.dialogdemo;

import android.app.Dialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.widget.Toast;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AlertDialog;
import androidx.fragment.app.DialogFragment;

public class MyDialogFragment extends DialogFragment {

   @NonNull
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
```

```
        builder.setTitle("Sample Dialog")
                .setMessage("This is a simple dialog")
               .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int id) {
                        Toast.makeText(getActivity(), "OK clicked",
                                                Toast.LENGTH_SHORT).show();
                    }
                })
                .setNegativeButton("Cancel", new DialogInterface.
                                                    OnClickListener() {
                    public void onClick(DialogInterface dialog, int id) {
                        Toast.makeText(getActivity(), "Cancel clicked",
                                                Toast.LENGTH_SHORT).show();
                    }
                });
        return builder.create();
    }
}
```

Explanation

- **MyDialogFragment:** This class extends 'DialogFragment' to display a simple dialog.
- **onCreateDialog:** This method is invoked to create the dialog. An 'AlertDialog.Builder' is used to construct the dialog, setting its title, message, and two buttons ('OK' and 'Cancel'). Each button, when clicked, triggers a toast message to be displayed.

Step 3: Update the MainActivity.java:

1. **Modify the MainActivity:**

   The 'MainActivity.java' file should be updated to show the dialog when a button is clicked.

2. **Add the below code for 'MainActivity.java':**

Code MainActivity.java

```
package com.example.dialogdemo;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

```
        Button showDialogButton = findViewById(R.id.showDialogButton);
        showDialogButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                MyDialogFragment dialogFragment = new MyDialogFragment();
          dialogFragment.show(getSupportFragmentManager(), "MyDialogFragment");
            }
        });
    }
}
```

Explanation

- **MainActivity**: This class sets the content view to 'activity_main.xml' and initializes a button to show the dialog.
- **Button Click Listener:** The click listener is set to display the 'MyDialogFragment' when the button is clicked.

Step 4: Update the Activity Layout - activity_main.xml:

Code activity_main.xml

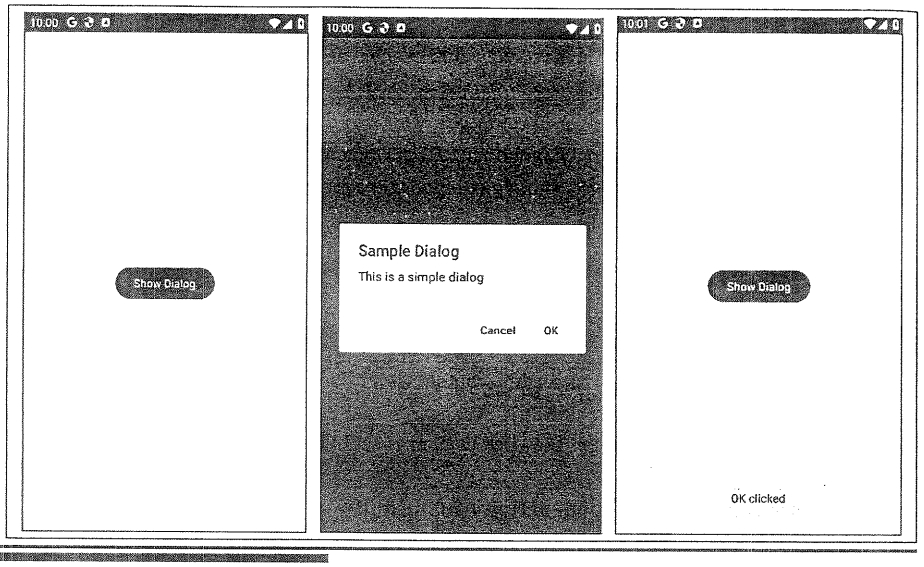
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center">

    <Button
        android:id="@+id/showDialogButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Show Dialog" />
</LinearLayout>
'''
```

Explanation

- **LinearLayout:** This layout acts as a container for the button.
- **Button:** The button, when clicked, triggers the display of 'MyDialogFragment'.

Output

When the application is run and the "Show Dialog" button is clicked, a dialog with the title "Sample Dialog" and the message "This is a simple dialog" appears. Clicking "OK" or "Cancel" displays a toast message indicating which button was clicked.



5.5.3 Using a PreferenceFragment

'PreferenceFragment' is a specialized fragment in Android designed to handle preferences in an application. Extending the 'Fragment' class, 'PreferenceFragment' provides a standard way to display and manage user preferences, The primary goal of 'PreferenceFragment' is to simplify the implementation of a settings screen within an Android application, promoting modular and reusable design for preference management.

By using PreferenceFragment, developers can easily create a settings screen for their application by defining preferences in XML and letting the PreferenceFragment handle the display and management of these preferences. This promotes modular and reusable design for handling user preferences in Android applications.

Key Features of PreferenceFragment

1. **Encapsulation of Preferences:**

   'PreferenceFragment' is used to streamline the process of displaying and managing user preferences within a Fragment. This encapsulation promotes modular design and reusability of preference-related functionality.

2. **Lifecycle Management:**

   'PreferenceFragment' includes its own lifecycle methods, similar to those of an Activity or Fragment. These methods facilitate easier management of preference-related operations, such as 'onCreate()', 'onCreateView()', 'onResume()', 'onPause()', and 'onDestroyView()'.

3. **XML-Based Preference Hierarchy:**

   'PreferenceFragment' utilizes an XML-based preference hierarchy for the definition and organization of user preferences. This approach allows for a clear and structured representation of settings within the application.

4. **Built-in Preference Handling:**
   - 'PreferenceFragment' offers built-in support for standard preference types (e.g., 'CheckBoxPreference', 'EditTextPreference', 'ListPreference', etc.), simplifying the implementation of common preference interactions.
5. **Modular and Reusable Components:**

   'PreferenceFragment' facilitates the creation of modular and reusable components that manage user preferences within various parts of an application. By encapsulating preference functionality within a Fragment, 'PreferenceFragment' enhances code reusability and maintainability.

Step-by-Step Implementation of PreferenceFragment

Step 1: Set Up the Project:

An Android project can be created using an empty activity and named 'PreferenceDemo'.

Step 2: Create the PreferenceFragment Class:

1. **Add a New Java Class:**
   - Within the project structure, right-click on the 'java' package (e.g., 'com.example.preferencedemo').
   - Select 'New > Java Class'.
   - Name the class 'SettingsFragment'.
   - Click "OK".
2. **Add the below code for 'SettingsFragment.java':**

Code SettingsFragment.java

```
package com.example.preferencedemo;

import android.os.Bundle;
import androidx.preference.PreferenceFragmentCompat;

public class SettingsFragment extends PreferenceFragmentCompat {

    @Override
    public void onCreatePreferences(Bundle savedInstanceState, String rootKey) {
        setPreferencesFromResource(R.xml.preferences, rootKey);
    }
}
```

Explanation

- **SettingsFragment:** This class extends 'PreferenceFragmentCompat' to display user preferences.
- **onCreatePreferences:** This method is invoked to create the preference hierarchy. The 'setPreferencesFromResource' method is used to load the preferences from an XML resource file ('R.xml.preferences').

Step 3: Create the XML Preference Resource:

1 .Add a New XML Resource File:

- Within the 'res' directory, create a new 'xml' directory if it doesn't exist.
- Right-click on the 'xml' directory, select 'New > XML > XML Resource File'.
- Name the file 'preferences'.
- Click "OK".

2. **Add the below code for 'preferences.xml':**

Code preferences.xml

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

    <EditTextPreference
        android:key="username"
        android:title="Username"
        android:summary="Enter your username"
        android:inputType="text" />

    <CheckBoxPreference
        android:key="notifications"
        android:title="Enable Notifications"
        android:summary="Receive notifications" />

    <ListPreference
        android:key="ringtone"
        android:title="Ringtone"
        android:summary="Select your ringtone"
        android:entries="@array/ringtone_names"
        android:entryValues="@array/ringtone_values" />

</PreferenceScreen>
```

Explanation

- **PreferenceScreen:** The root element for the preference hierarchy.
- **EditTextPreference:** A preference that allows text input for the username.
- **CheckBoxPreference:** A preference that enables or disables notifications.
- **ListPreference:** A preference that allows selection from a list of ringtones.

Step 4: Update the MainActivity.java:

1. **Modify the MainActivity:**

The 'MainActivity.java' file should be updated to show the 'SettingsFragment'.

2. **Add the below code for 'MainActivity.java':**

Code MainActivity.java

```
package com.example.preferencedemo;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        getSupportFragmentManager()
                .beginTransaction()
                .replace(R.id.settings_container, new SettingsFragment())
                .commit();
    }
}
```

Explanation

- **MainActivity:** This class sets the content view to 'activity_main.xml' and replaces the container with 'SettingsFragment'.

Step 5: Update the Activity Layout - activity_main.xml:

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/settings_container" />
```

Explanation

- **FrameLayout:** This layout acts as a container for the 'SettingsFragment'.

Step 6: Add String Arrays for ListPreference:

1. **Add a New String Array Resource:**

   Within the 'res\values' directory, open the 'strings.xml' file.

2. **Add the below code for 'strings.xml':**

Code strings.xml

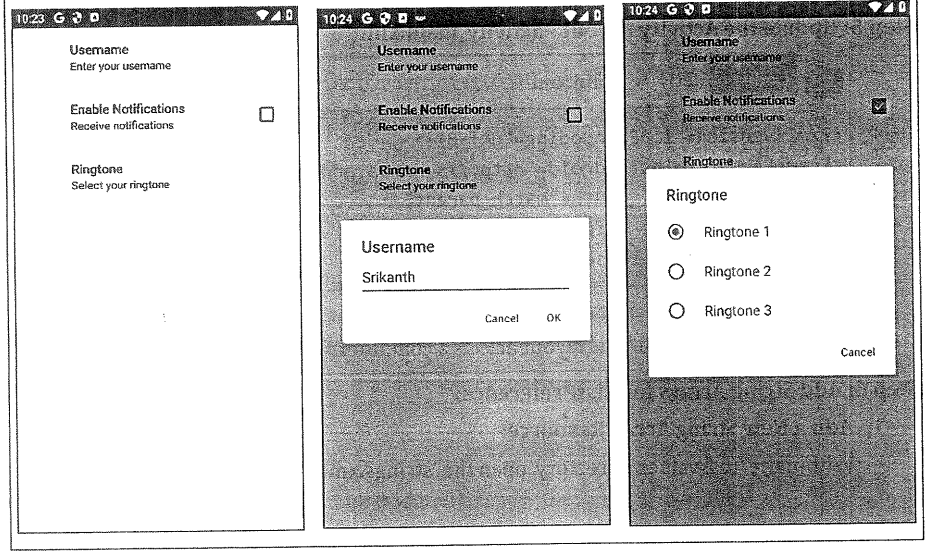
```
<resources>
    <string name="app_name">PreferenceDemo</string>
    <string-array name="ringtone_names">
        <item>Ringtone 1</item>
        <item>Ringtone 2</item>
        <item>Ringtone 3</item>
    </string-array>
    <string-array name="ringtone_values">
        <item>ringtone_1</item>
        <item>ringtone_2</item>
        <item>ringtone_3</item>
    </string-array>
</resources>
```

Explanation

- **string-array:** Defines arrays for ringtone names and values used by the 'ListPreference'.

Output

When the application is run, the settings screen displays three preferences: a text input for the username, a checkbox to enable notifications, and a list to select a ringtone. These preferences are defined in the XML resource and managed by the 'SettingsFragment'.



5.6 Using Image Views to Display Pictures

Displaying pictures is a common requirement in many types of applications, such as e-commerce platforms showcasing product images or book applications presenting cover images. Images play a crucial role in enhancing the visual appeal and user experience of an app. In Android development, several image views are available to cater to different display needs, each offering unique features and functionalities:

1. **ImageView:**

   The fundamental view for displaying images in Android is the 'ImageView'. It allows developers to show a single image within a layout. For example, in an e-commerce app, the product details screen may include an 'ImageView' to showcase the product's main image.

2. **Gallery:**

   The 'Gallery' view enables users to scroll through a collection of images horizontally. In a travel app, the Gallery view can be used to display a series of destination photos to swipe through them for a visual tour. Although Gallery is deprecated in the newer versions of Android SDK, a similar functionality can be achieved using RecyclerView with a horizontal layout manager

3. **ImageSwitcher:**

   The 'ImageSwitcher' view facilitates smooth transitions between images. It is ideal for creating image slideshows or galleries with animated effects. For instance, in a photography app, the ImageSwitcher can be used to showcase a portfolio with elegant image transitions.

4. **GridView:**

   When dealing with multiple images that need to be displayed in a grid layout, the 'GridView' view is a suitable choice. In a social media app, the user's photo album can be organized using a GridView to present a grid of thumbnail images that users can tap to view in full size.

5.6.1 Using ImageView

ImageView is a specialized view provided by Android for displaying images. It extends the View class and offers built-in support for loading, scaling, and displaying images from various sources, such as drawable resources, bitmap objects, or URIs. The primary goal of ImageView is to simplify the implementation of image-based UI elements within Android applications to promote efficient and effective image handling.

Key Features of ImageView

- **Loading Images:**

  ImageView supports loading images from various sources, including drawable resources, bitmap objects, and URIs. This flexibility simplifies the process of displaying images in an application.

- **Scaling and Adjusting:**

  ImageView offers multiple scaling options, such as fitCenter, centerCrop, centerInside, and matrix, allowing precise control over how images are displayed within the view boundaries.

- **Adjusting Image Properties:**

  ImageView allows adjusting various image properties, such as opacity, tint, and color filters. These properties can enhance the visual presentation of images in the application.

- **Handling Click Events:**

  ImageView can handle click events, making it possible to use images as interactive UI elements, such as buttons or links.

- **Built-in Animation Support:**

  ImageView supports built-in animation for transitioning between images to enhance the user experience with smooth visual effects.

Step-by-Step Implementation of ImageView

This implementation will demonstrate how to create a layout featuring three products: a TV, a Laptop, and an iPhone, with their respective images.

Step 1: Set Up the Project:

An Android project can be created using an empty activity and named 'ImageViewDemo'.

Step 2: Add Image Resources:

- **Add Images to the Drawable Folder:**

  Place three image files in the 'res/drawable' directory, naming them 'tv_image.png', 'laptop_image.png', and 'iphone_image.png'.

Step 3: Update the Activity Layout - activity_main.xml:

1. **Modify the XML Layout File:**

   Open the 'activity_main.xml' file and add 'ImageView' and 'TextView' elements for each product.

2. **Add the below code for 'activity_main.xml':**

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <!-- TV Product -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:padding="8dp"
        android:gravity="center_vertical">
```

```
    <ImageView
        android:id="@+id/tvImageView"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:src="@drawable/tv_image"
        android:contentDescription="@string/tv_image_description" />

    <TextView
        android:id="@+id/tvNameTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:text="TV"
        android:textSize="18sp" />
</LinearLayout>

<!-- Laptop Product -->
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="8dp"
    android:gravity="center_vertical">

    <ImageView
        android:id="@+id/laptopImageView"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:src="@drawable/laptop_image"
        android:contentDescription="@string/laptop_image_description" />

    <TextView
        android:id="@+id/laptopNameTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:text="Laptop"
        android:textSize="18sp" />
</LinearLayout>

<!-- iPhone Product -->
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
```

```
        android:padding="8dp"
        android:gravity="center_vertical">

        <ImageView
            android:id="@+id/iphoneImageView"
            android:layout_width="100dp"
            android:layout_height="100dp"
            android:src="@drawable/iphone_image"
            android:contentDescription="@string/iphone_image_description" />

        <TextView
            android:id="@+id/iphoneNameTextView"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="16dp"
            android:text="iPhone"
            android:textSize="18sp" />
    </LinearLayout>
</LinearLayout>
```

Explanation

- **LinearLayout:** The root layout and each product layout use a 'LinearLayout' with vertical orientation for the root and horizontal orientation for the product entries.
- **ImageView:** Displays the product images ('tv_image', 'laptop_image', 'iphone_image') from the drawable resources.
- **TextView:** Displays the product names ('TV', 'Laptop', 'iPhone') next to their respective images.

Step 4: Update the MainActivity.java:

1. **Modify the MainActivity:**

   The 'MainActivity.java' file can remain simple, as no additional logic is required for this static layout.

2. **Add the below code for 'MainActivity.java'**

Code MainActivity.java

```
package com.example.imageviewdemo;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

Explanation
• MainActivity: Sets the content view to 'activity_main.xml', which displays the static layout with product images and names.

Step 5: Add String Resources:

1. **Add Descriptions to the String Resources:**

   Open the 'strings.xml' file to add descriptions for the images.

2. **Add the below code for 'strings.xml':**

Code strings.xml

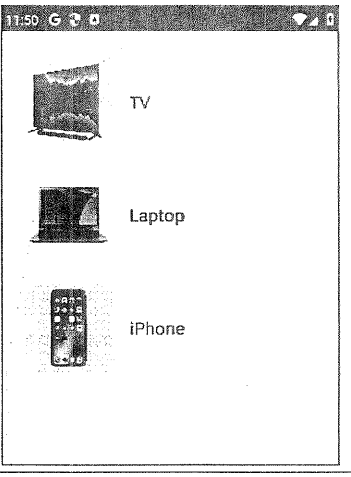
```
<resources>
    <string name="app_name">ProductDemo</string>
    <string name="tv_image_description">Image of a TV</string>
    <string name="laptop_image_description">Image of a Laptop</string>
    <string name="iphone_image_description">Image of an iPhone</string>
</resources>
```

Explanation

- **string:** Defines descriptions for each product image to improve accessibility.

Output

When the application is run, the main screen displays three products (TV, Laptop, and iPhone) with their respective images and names aligned horizontally. Each product entry consists of an 'ImageView' showing the product image and a 'TextView' displaying the product name.



5.6.2 Using GridView

'GridView' is a view in Android that displays items in a two-dimensional, scrollable grid. Extending the 'AdapterView' class, 'GridView' is highly customizable and efficient for displaying a large set of items, such as images and text, in a structured format. The primary goal of 'GridView' is to simplify the implementation of grid-based layouts within Android applications, promoting efficient and effective display of items.

Key Features of GridView

1. **Flexible Layout:**

   'GridView' provides a flexible grid-based layout, making it easy to display a large set of items in a structured and organized manner.

2. **Adapter Integration:**

   'GridView' works with adapters (such as 'ArrayAdapter' or 'BaseAdapter') to bind data to the grid items. This integration simplifies the process of populating the grid with data and updating it when the underlying dataset changes.

3. **Scalable and Scrollable:**

   'GridView' supports both vertical and horizontal scrolling, allowing it to handle a large number of items efficiently.

4. **Handling Click Events:**

   'GridView' can handle click events on individual grid items, making it possible to use the grid for interactive UI elements.

5. **Customizable Appearance:**

   'GridView' allows customization of item layout and spacing, providing flexibility in the visual presentation of grid items.

Step-by-Step Implementation of GridView

This implementation will demonstrate how to create a layout featuring a grid of three products: a TV, a Laptop, and an iPhone, with their respective images and names.

Step 1: Set Up the Project:

An Android project can be created using an empty activity and named 'GridViewDemo'.

Step 2: Add Image Resources:

1. **Add Images to the Drawable Folder:**

   Place three image files in the 'res/drawable' directory, naming them 'tv_image.png', 'laptop_image.png', and 'iphone_image.png'.

Step 3: Create the Item Layout - item_product.xml:

1. **Create a New XML Layout File:**

   Create a new XML layout file named 'item_product.xml' in the 'res/layout' directory.

2. **Add the below code for 'item_product.xml':**

Code item_product.xml

```
<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="8dp"
    android:gravity="center">
```

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

    <ImageView
        android:id="@+id/productImageView"
        android:layout_width="100dp"
        android:layout_height="100dp"
        android:contentDescription="@string/product_image_description" />

    <TextView
        android:id="@+id/productNameTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="18sp"
        android:layout_marginTop="8dp" />
</LinearLayout>
```

Explanation

- **LinearLayout:** A vertical layout containing an 'ImageView' and a 'TextView' to display the product image and name.

Step 4: Create the Adapter Class - ProductAdapter.java:

1. **Create a New Java Class:**

   Create a new Java class named 'ProductAdapter' in the 'java' package.

2. **Add the below code for 'ProductAdapter.java':**

Code ProductAdapter.java

```
package com.example.gridviewdemo;

import android.content.Context;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseAdapter;
import android.widget.ImageView;
import android.widget.TextView;

public class ProductAdapter extends BaseAdapter {

    private final Context context;
    private final String[] productNames;
    private final int[] productImages;

    public ProductAdapter(Context context, String[] productNames,
                                                    int[] productImages) {
        this.context = context;
        this.productNames = productNames;
        this.productImages = productImages;
    }
```

```
    @Override
    public int getCount() {
        return productNames.length;
    }

    @Override
    public Object getItem(int position) {
        return productNames[position];
    }

    @Override
    public long getItemId(int position) {
        return position;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        if (convertView == null) {
            convertView = LayoutInflater.from(context).
                                inflate(R.layout.item_product, parent, false);
        }

    ImageView productImageView = convertView.findViewById(R.id.productImageView);
        TextView productNameTextView =
                            convertView.findViewById(R.id.productNameTextView);

        productImageView.setImageResource(productImages[position]);
        productNameTextView.setText(productNames[position]);

        return convertView;
    }
}
```

Explanation

- **ProductAdapter:** A custom adapter class for the 'GridView', managing the display of product names and images.
- **getView:** Inflates the 'item_product' layout and binds the product data to the 'ImageView' and 'TextView'.

Step 5: Update the MainActivity.java:

1. **Modify the MainActivity:**

   The 'MainActivity.java' file should be updated to set up the 'GridView' with the 'ProductAdapter'.

2. **Add the below code for 'MainActivity.java':**

Code MainActivity.java

```
package com.example.gridviewdemo;

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

```
public class MainActivity extends AppCompatActivity {

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

        GridView gridView = findViewById(R.id.gridView);

        String[] productNames = {"TV", "Laptop", "iPhone"};
        int[] productImages = {R.drawable.tv_image,R.drawable.laptop_image,
                               R.drawable.iphone_image};

        ProductAdapter adapter = new ProductAdapter
                                        (this, productNames, productImages);
        gridView.setAdapter(adapter);
    }
}
```

Explanation

- **MainActivity:** Sets the content view to 'activity_main.xml' and initializes the 'GridView' with the 'ProductAdapter'.

Step 6: Update the Activity Layout - activity_main.xml:

1. Modify the XML Layout File:

   Open the 'activity_main.xml' file and add a 'GridView' element.

2. **Add the below code for 'activity_main.xml':**

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <GridView
        android:id="@+id/gridView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:numColumns="2"
        android:verticalSpacing="16dp"
        android:horizontalSpacing="16dp"
        android:stretchMode="columnWidth" />
</LinearLayout>
```

Explanation

- **GridView:** A view for displaying the product images and names in a grid layout. The 'numColumns' attribute sets the number of columns in the grid.

Step 7: Add String Resources:

1. **Add Descriptions to the String Resources:**

   Open the 'strings.xml' file to add descriptions for the images.

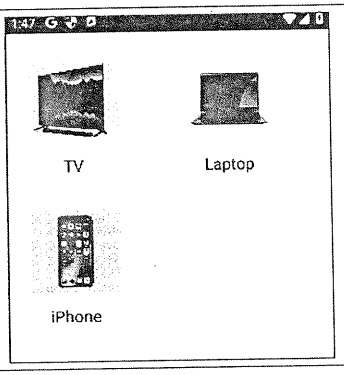
2. **Add the below code for 'strings.xml':**

Code strings.xml

```
<resources>
    <string name="app_name">GridViewDemo</string>
    <string name="product_image_description">Image of a product</string>
</resources>
```

Output

When the application is run, the main screen displays a grid with three products (TV, Laptop, and iPhone) each represented by an image and a name.



Explanation

- **string:** Defines a generic description for the product images to improve accessibility.

5.6.3 Using ImageSwitcher

'ImageSwitcher' is a specialized view in Android designed to animate transitions between images. It extends the 'ViewSwitcher' class and provides built-in support for smooth image transitions. It is ideal for creating image slideshows or gallery-like interfaces. The primary goal of 'ImageSwitcher' is to simplify the implementation of animated image transitions within Android applications to enhance the user experience with visually appealing effects.

Key Features of ImageSwitcher

1. **Smooth Image Transitions:**

   'ImageSwitcher' provides built-in support for animating transitions between images for a smooth and visually appealing user experience.

2. **Flexible Layout:**

   'ImageSwitcher' supports various layouts for displaying images and to customize the appearance and behavior of the image transitions.

3. **Adapter Integration:**

   'ImageSwitcher' can integrate with adapters to bind data to the images. It simplifies the process of populating the switcher with dynamic image content.

4. **Handling Click Events:**

   'ImageSwitcher' can handle click events to use images as interactive UI elements, such as buttons or links.

5. **Customizable Animation:**

   'ImageSwitcher' allows to define custom animations for image transitions. It provides flexibility in the visual presentation of the images.

Step-by-Step Implementation of ImageSwitcher

This implementation will demonstrate how to create a layout featuring an 'ImageSwitcher' that transitions between images of three products: a TV, a Laptop, and an iPhone, with navigation buttons to move to the next or previous image.

Step 1: Set Up the Project:

An Android project can be created using an empty activity and named 'ImageSwitcherDemo'.

Step 2: Add Image Resources:

1. **Add Images to the Drawable Folder:**

   Place three image files in the 'res/drawable' directory, naming them 'tv_image.png', 'laptop_image.png', and 'iphone_image.png'.

Step 3: Update the Activity Layout - activity_main.xml:

1. **Modify the XML Layout File:**

   Open the 'activity_main.xml' file and add an 'ImageSwitcher' element and 'Button' elements to navigate between images.

2. **Add the below code for 'activity_main.xml':**

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <ImageSwitcher
        android:id="@+id/imageSwitcher"
```

```
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:inAnimation="@android:anim/fade_in"
        android:outAnimation="@android:anim/fade_out" />

    <LinearLayout
        android:id="@+id/buttonLayout"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:gravity="center"
        android:layout_marginTop="50dp">

        <Button
            android:id="@+id/prevButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Previous" />

        <Button
            android:id="@+id/nextButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="16dp"
            android:text="Next" />
    </LinearLayout>
</LinearLayout>
```

Explanation

- **LinearLayout:** The root layout uses a 'LinearLayout' with vertical orientation to organize the 'ImageSwitcher' and 'Button' elements.
- **ImageSwitcher:** Displays the images with fade-in and fade-out animations.
- **Button:** Elements are used to navigate between the images.

Step 4: Update the MainActivity.java:

1. **Modify the MainActivity:**

   The 'MainActivity.java' file should be updated to handle the image switching logic.

2. **Add the below code for 'MainActivity.java':**

Code MainActivity.java

```
package com.example.imageswitcherdemo;

import android.os.Bundle;
import android.util.Log;
import android.view.View;
```

```
import android.widget.Button;
import android.widget.ImageSwitcher;
import android.widget.ImageView;
import android.widget.ViewSwitcher;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private ImageSwitcher imageSwitcher;
    private int[] imageIds = {R.drawable.tv_image, R.drawable.laptop_image,
                              R.drawable.iphone_image};
    private int currentIndex = 0;
    private Button prevButton, nextButton;

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

        imageSwitcher = findViewById(R.id.imageSwitcher);
        prevButton = findViewById(R.id.prevButton);
        nextButton = findViewById(R.id.nextButton);

        imageSwitcher.setFactory(new ViewSwitcher.ViewFactory() {
            @Override
            public View makeView() {
                ImageView imageView = new ImageView(MainActivity.this);
                imageView.setScaleType(ImageView.ScaleType.FIT_CENTER);
                imageView.setLayoutParams(new ImageSwitcher.LayoutParams(
                        ImageSwitcher.LayoutParams.WRAP_CONTENT,
                        ImageSwitcher.LayoutParams.WRAP_CONTENT));
                return imageView;
            }
        });

        imageSwitcher.setImageResource(imageIds[currentIndex]);
       prevButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if (currentIndex > 0) {
                    currentIndex--;
                 } else {
                    currentIndex = imageIds.length - 1;
```

```
                }
                imageSwitcher.setImageResource(imageIds[currentIndex]);
            }
        });

        nextButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if (currentIndex < imageIds.length - 1) {
                    currentIndex++;
                } else {
                    currentIndex = 0;
                }
                imageSwitcher.setImageResource(imageIds[currentIndex]);
            }
        });
    }
}
```

Explanation

- **MainActivity:** Sets the content view to 'activity_main.xml' and initializes the 'ImageSwitcher' and 'Button' elements.
- **ImageSwitcher Factory:** Defines how the 'ImageView' for the 'ImageSwitcher' should be created.
- **Button Click Listeners:** Handle the logic for switching images when the "Previous" and "Next" buttons are clicked.

Step 5: Add String Resources:

1. **Add Descriptions to the String Resources:**

   Open the 'strings.xml' file to add descriptions for the images.

2. **Add the below code for 'strings.xml':**

Code strings.xml

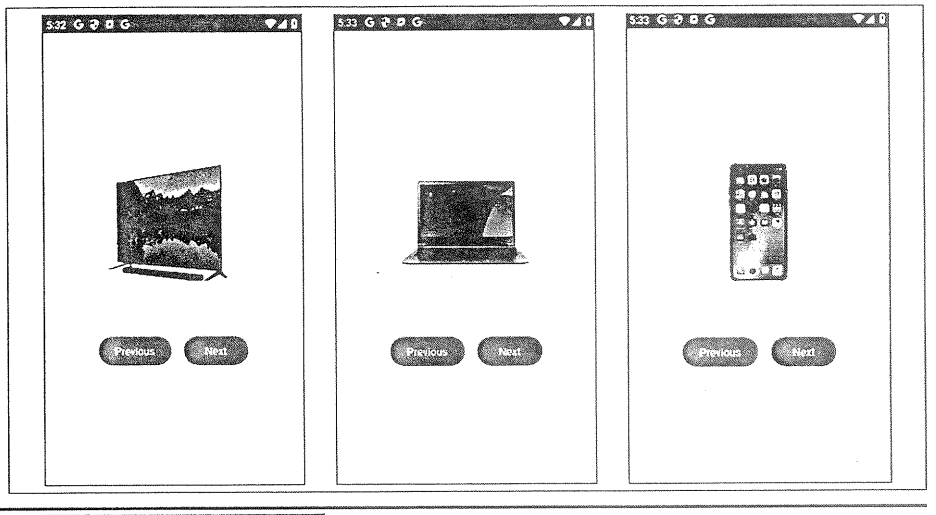
```
<resources>
    <string name="app_name">ImageSwitcherDemo</string>
    <string name="product_image_description">Image of a product</string>
</resources>
```

Explanation

- **string:** Defines a generic description for the product images to improve accessibility.

Output

When the application is run, the main screen displays an 'ImageSwitcher' that animates transition between images of three products (TV, Laptop, and iPhone). The "Previous" and "Next" buttons allow users to navigate through the images.



5.7 Using Menus With Views

In Android, menus provide a way to present user actions and options in a consistent manner. Menus can be used for navigation, contextual actions, or to provide additional options and settings for an app. There are two main types of menus in Android:

1. **Options Menu:** This menu appears when the user presses the menu button on their device or taps the three dots (overflow menu) in the app's action bar.

   Example : In WhatsApp, the three dots in the top-right corner of the chat screen open an options menu with actions such as "New group," "New broadcast," "WhatsApp Web," "Starred messages," and "Settings."

2. **Context Menu:** This menu appears when the user performs a long-click on a view. It provides contextual actions related to the selected view.

   Example : Context menus provide information specific to a selected item within an activity. In a file manager app such as Google Files, long-pressing on a file triggers a context menu with options like renaming the file or sharing it with others.

Features of Options Menu

The Options Menu in Android provides a way to include additional actions or settings relevant to the current activity or screen. Some key features of the Options Menu include:

1. **Visibility:** The Options Menu is typically accessed by tapping the menu icon (three vertical dots) or the physical menu button on older devices. It offers a convenient way to access less frequently used actions without cluttering the main interface.
2. **Contextual Actions:** The Options Menu displays actions that are contextually relevant to the current screen or activity. For example, in a messaging app, options may include actions like composing a new message, searching for a contact, or accessing settings specific to messaging .

3. **Icon Support:** Menu items in the Options Menu can include icons to provide visual cues for users. Icons can help users quickly identify and select the desired action .
4. **Overflow Menu:** In case the Options Menu has more items than can fit on the screen, an overflow menu icon (usually represented by three vertical dots) appears. Tapping on this icon reveals additional menu items .
5. **Customization:** Developers can customize the Options Menu by adding, removing, or modifying menu items based on the requirements of the application. This flexibility allows for tailoring the menu to suit the specific needs of the app .
6. **Handling Menu Selections:** Developers can implement methods like 'onCreateOptionsMenu()' and 'onOptionsItemSelected()' to handle the creation of the Options Menu and the actions taken when a menu item is selected, respectively .

Features of Context Menu

The Context Menu in Android offers specific actions or options relevant to a selected item or view within an activity. Some key features of the Context Menu include:

1. **Contextual Actions:** The Context Menu displays actions that are specific to the item or view that triggered the menu. For example, in a file manager app, long-pressing on a file may bring up a context menu with options to rename, delete, or share that particular file.
2. **Trigger Mechanism:** The Context Menu is typically triggered by a long-press (press and hold) gesture on an item or view. This gesture indicates to the user that additional actions are available for the selected item .
3. **Item-Specific Options:** The menu items in the Context Menu are tailored to the type of item or view selected. This allows for a more focused set of actions that are directly related to the selected item.
4. **Dynamic Content:** The content of the Context Menu can change based on the type of item selected or the context in which the menu is triggered. This dynamic nature ensures that users are presented with relevant options at all times.
5. **Handling Menu Selections:** Developers can implement methods like 'onCreateContextMenu()' and 'onContextItemSelected()' to handle the creation of the Context Menu and the actions taken when a menu item is selected, respectively.
6. **Customization:** Similar to the Options Menu, developers have the flexibility to customize the Context Menu by adding or removing menu items based on the specific requirements of the application .

Note:

In the previous chapter, we explored the creation of menus as part of action bars in Android applications. This involved creating a menu.xml file to define the menu structure, adding action items to the menu, and handling overflow menu items for cases where the screen space is limited. In this section, we will understand how to implement context menu.

Step-by-Step Implementation of Context Menu

This implementation will demonstrate how to create an Android application featuring a context menu. A context menu appears when a user performs a long-click on a view.

Step 1: Create a Project:

Create a new Android project named 'MenuDemo'.

Step 2: Create Context Menu:

1. **Create the XML for the Context Menu:**

   Create a file named 'res/menu/context_menu.xml' with the following content:

Code context_menu.xml

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/context_reply"
        android:title="Reply" />
    <item
        android:id="@+id/context_forward"
        android:title="Forward" />
    <item
        android:id="@+id/context_star"
        android:title="Star" />
    <item
        android:id="@+id/context_delete"
        android:title="Delete" />
    <item
        android:id="@+id/context_copy"
        android:title="Copy" />
</menu>
```

Explanation

- **menu:** Root element defining the menu.
- **item:** Defines each menu item with an ID and a title.

Step 3: Update the Activity Layout - activity_main.xml:

1. **Modify the XML Layout File:**

   Open the 'activity_main.xml' file and add a 'TextView' element for demonstrating the context menu.

2. **Add the below code for 'activity_main.xml':**

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <TextView
        android:id="@+id/sample_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="10pt"
        android:text="Long Press Me for Context Menu" />
</LinearLayout>
```

Explanation

- **LinearLayout:** The root layout uses a 'LinearLayout' with vertical orientation to organize the 'TextView' element.
- **TextView:** Displays a text that will be used to demonstrate the context menu.

Step 4: Update MainActivity.java:

1. **Modify the MainActivity:**

   The 'MainActivity.java' file should be updated to handle the context menu.

2. **Add the below code for 'MainActivity.java':**

Code MainActivity.java

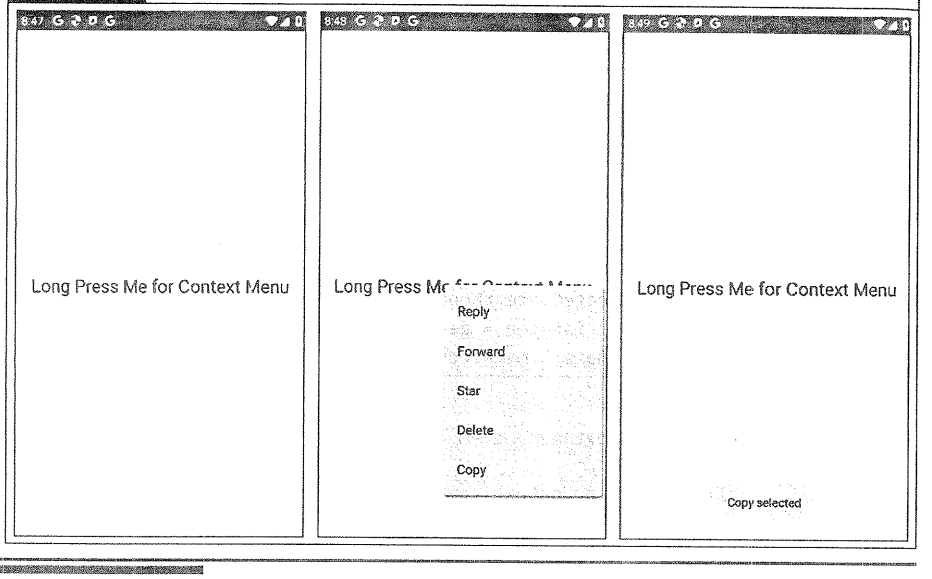
```
package com.example.menudemo;

import android.os.Bundle;
import android.view.ContextMenu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

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

        textView = findViewById(R.id.sample_text);
        registerForContextMenu(textView);
    }

    // Create context menu
    @Override
    public void onCreateContextMenu(ContextMenu menu, View v,
                                    ContextMenu.ContextMenuInfo menuInfo) {
        super.onCreateContextMenu(menu, v, menuInfo);
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.context_menu, menu);
    }

    // Handle context menu item clicks
    @Override
    public boolean onContextItemSelected(MenuItem item) {
        String title = item.getTitle().toString();
        if (title.equals("Reply")) {
          Toast.makeText(this, "Reply selected", Toast.LENGTH_SHORT).show();
            return true;
        } else if (title.equals("Forward")) {
         Toast.makeText(this, "Forward selected", Toast.LENGTH_SHORT).show();
            return true;
        } else if (title.equals("Star")) {
           Toast.makeText(this, "Star selected", Toast.LENGTH_SHORT).show();
            return true;
        } else if (title.equals("Delete")) {
          Toast.makeText(this, "Delete selected", Toast.LENGTH_SHORT).show();
            return true;
        } else if (title.equals("Copy")) {
           Toast.makeText(this, "Copy selected", Toast.LENGTH_SHORT).show();
            return true;
        } else {
            return super.onContextItemSelected(item);
        }
    }
}
```

Explanation

- **MainActivity:** Sets the content view to 'activity_main.xml' and initializes the 'TextView' element.
- **registerForContextMenu:** Registers the 'TextView' to show a context menu when long-pressed.
- **onCreateContextMenu:** Inflates the context menu from 'res/menu/context_menu.xml'.
- **onContextItemSelected:** Handles clicks on the context menu items, showing a 'Toast' message based on the selected item.

Output



5.8 Using WebView

'WebView' is a view that displays web pages within an Android application. It is based on the WebKit engine and allows integration of web content seamlessly into the app. 'WebView' can be used to show static HTML content or to load web pages from the internet. It's particularly useful for displaying content hosted online or for creating hybrid applications that combine web and native views.

Features of WebView

1. **Loading Web Content:**
   - 'WebView' can load and display web pages using the 'loadUrl' method.
   - It can load both local HTML files and web pages from the internet.
2. **JavaScript Support:**
   - 'WebView' can execute JavaScript to run web applications that require JavaScript.
3. **Navigation Controls:**
   - It provides navigation methods such as 'goBack', 'goForward', and 'reload' to control the browsing history within the 'WebView'.
4. **Customizable User Interface:**
   - The look and feel of the 'WebView' can be customized to match the rest of the application's design.
5. **Handling Links and Forms:**
   - 'WebView' can handle links and forms to enable full web page interaction within the application.

6. **Zoom Controls:**
   - It supports pinch-to-zoom and can display zoom controls for a better user experience.
7. **WebViewClient:**
   - By using a 'WebViewClient', the behavior of the 'WebView' when navigating to new URLs, handling page loading, and managing errors can be controlled.

Step-by-Step Implementation of WebView

Step-by-Step Implementation of WebView:

Step 1: Create a Project:

Create a new Android project named 'WebViewDemo'.

Step 2: Add Internet Permission:

1. **Modify the AndroidManifest.xml:**

   Open the 'AndroidManifest.xml' file and add the following line before the '<application>' tag:

Code AndroidManifest.xml

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Explanation

- **uses-permission:** Declares the permission to use the internet, which is required for loading web content.

Step 3: Update the Activity Layout - activity_main.xml:

1. **Modify the XML Layout File:**

   Open the 'activity_main.xml' file and add a 'WebView' element.

2. **Add the below code for 'activity_main.xml':**

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <WebView
        android:id="@+id/webView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</RelativeLayout>
```

Explanation

- **RelativeLayout:** The root layout uses a 'RelativeLayout' to position the 'WebView' element.
- **WebView:** Displays web content and is set to fill the parent layout.

Step 4: Update MainActivity.java:

1. **Modify the MainActivity:**

   The 'MainActivity.java' file should be updated to handle the 'WebView' setup and load a web page.

2. **Add the below code for 'MainActivity.java':**

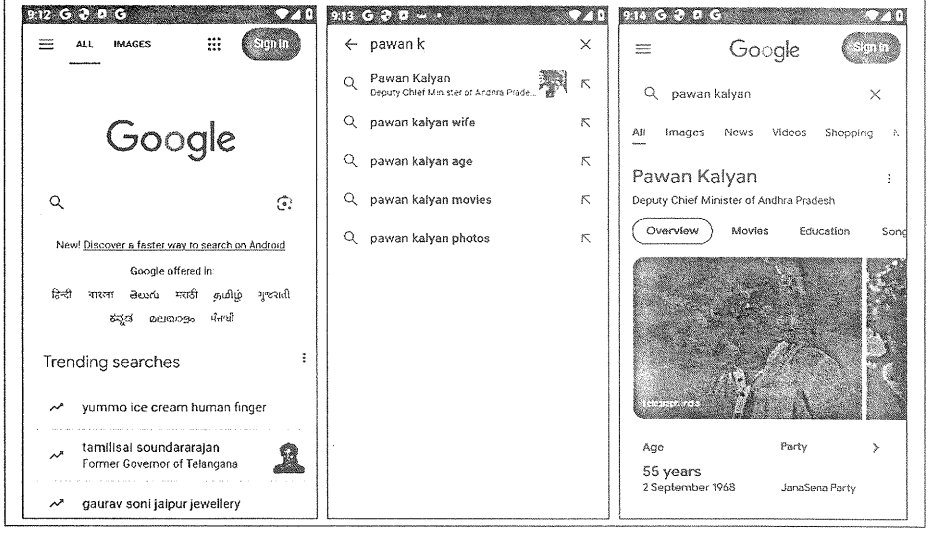
Code MainActivity.java

```
package com.example.webviewdemo;

import android.os.Bundle;
import android.webkit.WebSettings;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private WebView webView;

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

        webView = findViewById(R.id.webView);

        // Enable JavaScript
        WebSettings webSettings = webView.getSettings();
        webSettings.setJavaScriptEnabled(true);

        // Ensure links open within the WebView instead of a browser
        webView.setWebViewClient(new WebViewClient());

        // Load a web page
        webView.loadUrl("https://www.google.co.in");
    }

    // Handle back button press to navigate within WebView

    @Override
    public void onBackPressed() {
        if (webView.canGoBack()) {
            webView.goBack();
        } else {
            super.onBackPressed();
        }
    }
}
```

Explanation

- **MainActivity:** Sets the content view to 'activity_main.xml' and initializes the 'WebView' element.
- **WebSettings:** Enables JavaScript for the 'WebView'.
- **WebViewClient:** Ensures that links clicked within the 'WebView' open within the same 'WebView' instead of an external browser.
- **loadUrl:** Loads a specified URL (www.google.co.in) into the 'WebView'.
- **onBackPressed:** Overrides the back button behavior to navigate back within the 'WebView' history if possible.

Output



5.9 Data Persistence

Data persistence in Android refers to the capability of an application to save data and maintain it across different sessions. This ensures that user data is not lost when the app is closed or the device is restarted. Effective data persistence is crucial for providing a seamless and consistent user experience. Android offers several mechanisms for data persistence, including Shared Preferences for simple key-value pairs, file systems for storing larger files, and using databases for handling complex relational data.

Different Ways of Persisting Data

Android offers three fundamental methods for persisting data:

1. **Shared Preferences:** This lightweight mechanism is used to store small pieces of data, such as user preferences or settings. Shared preferences are key-value pairs that can be easily accessed and modified. This method is suitable for storing simple data that does not require complex querying .

Storing user preferences and settings using shared preferences include theme preferences, language settings, or notification preferences so that the app can personalize the user experience across sessions.

Examples	Shared Preferences

1. **Theme Selection:**

   Users can personalize their app experience by selecting a preferred theme from options like light or dark mode. When a user chooses a theme, the app saves this preference using shared preferences as a key-value pair. Upon each app launch, the stored theme preference is retrieved from shared preferences, allowing the app to apply the selected theme consistently across different screens and interactions. This customization enhances user satisfaction and visual comfort within the app environment.

2. **Notification Preferences:**

   To cater to individual preferences regarding notifications, users are provided with the flexibility to customize settings such as sound and vibration for alerts. When a user adjusts these notification preferences within the app, the choices are stored in shared preferences as key-value pairs. Subsequently, the app references these stored preferences to determine how notifications should be presented to the user to ensure that alerts align with the user's desired experience. This level of personalization enhances user control and engagement with the app's notification system.

2. **Traditional File Systems:** Android applications can also save data using traditional file systems. This method involves writing data to files in internal or external storage. It is useful for storing larger amounts of data or files that need to be accessed directly by the application.

Examples	Traditional File Systems

1. **Media Files:** Saving media files, such as audio or video recordings, to the device's external file system. By storing these files in the external storage (e.g., SD card), users can access and manage their media content separately from the app's internal storage. This approach is beneficial for apps that deal with large media files or require users to share media across different applications.
2. **Offline Data Storage :** Saving user-generated content, such as notes, documents, or images, to the device's internal storage as files. This allows users to access and modify their data even when offline, ensuring a seamless user experience.

3. **Using Databases:** Android supports SQLite databases for managing structured data. SQLite is a relational database management system that allows for efficient storage, retrieval, and querying of data. This method is ideal for handling complex data structures and relationships within an application.

Examples	Using Databases

1. **Login Credentials :** Persisting user login credentials securely using encrypted storage or SQLite databases. This enables users to stay logged in between app sessions without needing to re-enter their credentials each time.

2. **Personalized Content Recommendations:**

   To offer personalized content recommendations, user preferences and interaction data are stored in a SQLite database. By utilizing the database to track user behavior and preferences, the app can deliver tailored content suggestions based on individual interests, improving user engagement and satisfaction.

Choosing the right data persistence mechanism depends on the type and complexity of the data to store. Shared Preferences are ideal for small and simple data, file systems are suitable for larger and unstructured data, and SQLite databases are perfect for complex and relational data structures.

5.9.1 Saving and Loading User Preferences

'**Shared Preferences**' is a lightweight mechanism provided by Android for storing small pieces of data as key-value pairs. This is ideal for saving simple data such as user settings and preferences that need to persist across application sessions.

Features of Shared Preferences

1. **Simple API:**

   Provides an easy-to-use interface for storing and retrieving data.

2. **Persistent Storage:**

   Data is stored persistently across app sessions and device reboots.

3. **Lightweight:**

   Suitable for small pieces of data (e.g., user settings, configuration).

4. **Private Storage:**

   Data is stored in the app's private directory, ensuring security and privacy.

5. **Flexibility:**

   Supports various data types including strings, integers, booleans, floats, and longs.

How It Works?

1. **Initial Setup:**

   - **Create or Open Shared Preferences:**

     To store user preferences, you need to create or open a SharedPreferences file. This file will hold the key-value pairs of your data.

Code

```
SharedPreferences sharedPreferences=getSharedPreferences("UserPreferences",
Context.MODE_PRIVATE);
```

Explanation

- **getSharedPreferences("UserPreferences", Context.MODE_PRIVATE);:** This method creates or opens a SharedPreferences file named "UserPreferences" in private mode, which means the file is accessible only by your application.

2. **Saving User Preferences:**
   - **Retrieve an Editor:**

     To modify the SharedPreferences file, you need to get an instance of SharedPreferences.Editor.

     Code

     ```
     SharedPreferences.Editor editor = sharedPreferences.edit();
     ```

   - **Put Data into Preferences:**

     Use the editor to put data into the SharedPreferences file. The 'putString()', 'putInt()', 'putBoolean()', etc., methods are used for saving different data types.

     Code

     ```
     editor.putString("language", "English");
     editor.putBoolean("notificationsEnabled", true);
     ```

     Explanation

     - **putString("language", "English");:** This method saves the string "English" with the key "language".
     - **putBoolean("notificationsEnabled", true);:** This method saves the boolean value 'true' with the key "notificationsEnabled".

   - **Commit Changes:**

     After putting the data into preferences, you need to commit the changes.

     Code

     ```
     editor.apply(); // or editor.commit();
     ```

     Explanation

     - **apply();:** This method saves the changes asynchronously. It is faster as it does not block the main thread.
     - **commit();:** This method saves the changes synchronously and returns a boolean value indicating success or failure. It blocks the main thread until the changes are saved.

3. **Loading User Preferences:**
   - **Retrieve Data:**

     To load the user preferences, you can retrieve the saved data from the SharedPreferences file using the 'getString()', 'getInt()', 'getBoolean()', etc., methods.

     Code

     ```
     String language = sharedPreferences.getString("language", "defaultLanguage");
     boolean notificationsEnabled = sharedPreferences.getBoolean("notificationsEnabled",
     false);
     ```

Explanation
• **getString("language", "defaultLanguage");:** This method retrieves the string value associated with the key "language". If the key does not exist, it returns "defaultLanguage". • **getBoolean("notificationsEnabled", false);:** This method retrieves the boolean value associated with the key "notificationsEnabled". If the key does not exist, it returns 'false'.

Example Implementation: Saving and Loading a User's Language Preference

Step 1: Create a Project:

Create a new Android project named 'PreferencesDemo'.

Step 2: Update the Activity Layout - activity_main.xml:

1. **Modify the XML Layout File:**

   Open the 'activity_main.xml" file and add UI elements for setting and displaying preferences.

2. **Add the below code for 'activity_main.xml':**

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:id="@+id/languageLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Select Language" />

    <Spinner
        android:id="@+id/languageSpinner"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:entries="@array/languages" />
    <TextView
        android:id="@+id/notificationsLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enable Notifications" />

    <Switch
        android:id="@+id/notificationsSwitch"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

```
        <Button
            android:id="@+id/saveButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Save Preferences" />

        <TextView
            android:id="@+id/preferencesTextView"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingTop="16dp"
            android:text="Preferences will be displayed here." />
    </LinearLayout>
```

Explanation

- **LinearLayout:** The root layout uses a 'LinearLayout' to arrange the UI elements vertically.
- **Spinner:** Allows the user to select a language.
- **Switch:** Allows the user to enable or disable notifications.
- **Button:** Saves the user preferences.
- **TextView:** Displays the saved preferences.

Step 3: Update Strings XML (res/values/strings.xml):

Add an array for the languages:

Code strings.xml

```
<resources>
    <string name="app_name">PreferencesDemo</string>
    <string-array name="languages">
        <item>English</item>
        <item>Spanish</item>
        <item>French</item>
        <item>German</item>
        <item>Chinese</item>
    </string-array>
</resources>
```

Step 4: Update MainActivity.java:

1. **Modify the MainActivity:**

   The 'MainActivity.java' file should be updated to handle the Shared Preferences setup and load/save user preferences.

2. **Add the below code for 'MainActivity.java':**

Code MainActivity.java

```
package com.example.preferencesdemo;

import android.content.Context;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.Spinner;
import android.widget.Switch;
import android.widget.TextView;

    public class MainActivity extends AppCompatActivity {

        private Spinner languageSpinner;
        private Switch notificationsSwitch;
        private TextView preferencesTextView;

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

            languageSpinner = findViewById(R.id.languageSpinner);
            notificationsSwitch = findViewById(R.id.notificationsSwitch);
            preferencesTextView = findViewById(R.id.preferencesTextView);
            Button saveButton = findViewById(R.id.saveButton);

            // Load saved preferences when the app starts
            loadUserPreferences();

            // Save preferences when the button is clicked
            saveButton.setOnClickListener(v -> saveUserPreferences());
        }

        private void saveUserPreferences() {
            // Get selected language
            String language = languageSpinner.getSelectedItem().toString();
            // Get notification preference
            boolean notificationsEnabled = notificationsSwitch.isChecked();
            // Create or open Shared Preferences
            SharedPreferences sharedPreferences =
```

```
            getSharedPreferences("UserPreferences", Context.MODE_PRIVATE);
        SharedPreferences.Editor editor = sharedPreferences.edit();

        // Put data into preferences
        editor.putString("language", language);
        editor.putBoolean("notificationsEnabled", notificationsEnabled);

        // Commit changes
        editor.apply();

        // Update the preferences TextView
        preferencesTextView.setText(String.format
    ("Language:%s\nNotificationsEnabled:%b",language,notificationsEnabled));
    }

    private void loadUserPreferences() {
        // Create or open Shared Preferences
        SharedPreferences sharedPreferences =
            getSharedPreferences("UserPreferences", Context.MODE_PRIVATE);

        // Retrieve data from Shared Preferences
      String language = sharedPreferences.getString("language", "English");
        boolean notificationsEnabled = sharedPreferences.
            getBoolean("notificationsEnabled", false);
        // Set the selected language in the spinner
       ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource
            (this,R.array.languages, android.R.layout.simple_spinner_item);
        adapter.setDropDownViewResource(android.R.layout.
                                        simple_spinner_dropdown_item);
        languageSpinner.setAdapter(adapter);

        int spinnerPosition = adapter.getPosition(language);
        languageSpinner.setSelection(spinnerPosition);

        // Set the notification switch state
        notificationsSwitch.setChecked(notificationsEnabled);

        // Update the preferences TextView
      preferencesTextView.setText(String.format("Language: %s\nNotification
                        Enabled: %b", language, notificationsEnabled));
    }
}
```

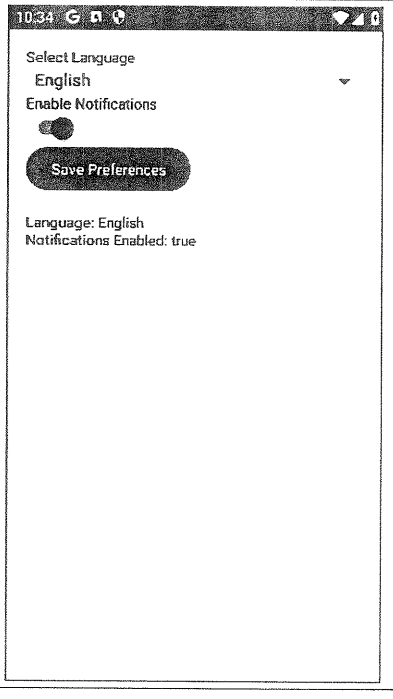
Explanation

1. **onCreate Method:**
   - **setContentView:** This method sets the layout for the activity using 'R.layout.activity_main', which refers to the XML layout file.
   - **Initialize UI Components:** The 'findViewById' method is used to link the UI components in the layout file to their corresponding fields in the activity code ('languageSpinner', 'notificationsSwitch', 'preferencesTextView', 'saveButton').
   - **loadUserPreferences:** This method is called to load any saved preferences when the activity starts. It ensures that the UI is updated with the saved preferences.
   - **setOnClickListener:** An 'OnClickListener' is set on the save button, which calls 'saveUserPreferences' when the button is clicked. This saves the current settings to Shared Preferences.
2. **saveUserPreferences Method:**
   - **Retrieve Selected Values:** The method gets the selected language from the 'Spinner' and the notification preference from the 'Switch'.
   - **Open Shared Preferences:** 'getSharedPreferences("UserPreferences", Context.MODE_PRIVATE)' is used to create or open a SharedPreferences file named "UserPreferences". The 'MODE_PRIVATE' mode means the file is accessible only by the application.
   - **Save Data:** The 'SharedPreferences.Editor' object, obtained by calling 'edit()', is used to put data into the preferences. The 'putString' and 'putBoolean' methods store the language and notification preferences.
   - **Commit Changes:** The 'apply' method commits the changes asynchronously, ensuring the save operation does not block the main thread.
   - **Update UI:** The 'TextView' is updated to display the saved preferences, confirming the operation to the user.
3. **loadUserPreferences Method:**
   - **Open Shared Preferences:** The method opens the same "UserPreferences" file in private mode.
   - **Retrieve Data:** The 'getString' and 'getBoolean' methods are used to retrieve the saved preferences. Default values ("English" and 'false') are provided in case no preferences are found.
   - **Set UI Components:**
     - **Spinner:** An 'ArrayAdapter' is created to populate the 'Spinner' with language options. The saved language is set as the selected item in the 'Spinner' by finding its position in the adapter and calling 'setSelection'.
     - **Switch:** The state of the 'Switch' is set to reflect the saved notification preference.
     - **Update UI:** The 'TextView' is updated to display the loaded preferences, providing visual feedback to the user about the current settings.

Explanation

When the application is run, the user can select their language preference and toggle notifications. These preferences are saved when the "Save Preferences" button is clicked and loaded the next time the app is opened.

Output



5.9.2 Persisting Data to Files

File Systems in Android provide a mechanism for storing data as files on the device. This is suitable for larger and more complex data that doesn't fit well into key-value pairs. We can store data in internal storage (private to the app) or external storage (shared with other apps).

Features of File Systems

1. **Large Storage Capacity:**

   Can handle large files, including media files.

2. **Different Storage Locations:**

   Supports both internal storage (private to the app) and external storage (accessible by other apps with proper permissions).

3. **Flexible File Formats:**

   Allows storage of text files, binary files, images, etc.

4. **Private Storage:**

   Internal storage ensures that files are private to the app and cannot be accessed by other apps.

How It Works?

1. **Writing Data to Files:**

   To store data in a file, we need to open a file output stream and write data to it.

   Example: Saving user-generated notes.

Code

```
String filename = "user_notes.txt";
String content = "This is a user note.";

FileOutputStream fos = openFileOutput(filename, Context.MODE_PRIVATE);
fos.write(content.getBytes());
fos.close();
```

Explanation

- **openFileOutput:** This method opens a file output stream to write data to the specified file. The 'MODE_PRIVATE' mode means the file is accessible only by the application.
- **write:** This method writes the data (as bytes) to the file.
- **close:** This method closes the file output stream, ensuring all data is written to the file.

2. **Reading Data from Files:**

   To read data from a file, you need to open a file input stream and read the data.

Code

```
String filename = "user_notes.txt";

FileInputStream fis = openFileInput(filename);
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);
StringBuilder sb = new StringBuilder();
String line;

while ((line = br.readLine()) != null) {
    sb.append(line);
}

String fileContent = sb.toString();
fis.close();
```

Explanation

- **openFileInput:** This method opens a file input stream to read data from the specified file.
- **InputStreamReader:** This class reads bytes and decodes them into characters using a specified charset.
- **BufferedReader:** This class reads text from an input stream, buffering characters for efficient reading.
- **readLine:** This method reads a line of text from the input stream.
- **close:** This method closes the file input stream, ensuring all data is read from the file.

Example Implementation: Saving and Loading User Notes

Step 1: Create a Project:

Create a new Android project named 'FilePersistenceDemo'.

Step 2: Declare the Permission in the AndroidManifest.xml:

Add the 'WRITE_EXTERNAL_STORAGE' permission in the 'AndroidManifest.xml' (before application tag)

Code

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Step 3: Update the Activity Layout - activity_main.xml:

1. **Modify the XML Layout File:**

   Open the 'activity_main.xml' file and add UI elements for entering and displaying notes.

2. **Add the below code for 'activity_main.xml':**

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:id="@+id/notesLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enter your notes" />

    <EditText
        android:id="@+id/notesEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter your notes here"
        android:inputType="textMultiLine"
        android:lines="4"
        android:gravity="top|start" />

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

```
        android:layout_height="wrap_content"
        android:text="Save Notes Internally" />

    <Button
        android:id="@+id/downloadButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Download Notes to Device Storage" />

    <TextView
        android:id="@+id/notesTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="16dp"
        android:text="Notes will be displayed here." />
</LinearLayout>
```

Explanation

- **LinearLayout:** The root layout uses a 'LinearLayout' to arrange the UI elements vertically.
- **EditText:** Allows the user to enter notes.
- **inputType="textMultiLine":** Enables multi-line text input.
- **lines="4":** Sets the EditText to display four lines by default.
- **gravity="top|start":** Aligns the text to the top and start (left) of the EditText.
- **Button:** Two buttons for saving notes internally and downloading notes to device storage.
- **TextView:** Displays the saved notes.

Step 4: Update MainActivity.java:

1. **Modify the MainActivity:**

   The 'MainActivity.java' file should be updated to handle file operations for saving and loading user notes and for downloading notes to external storage.

2. **Add the below code for 'MainActivity.java':**

Code MainActivity.java

```
package com.example.filepersistencedemo;

import android.Manifest;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.os.Environment;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
```

```
import android.widget.Toast;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.InputStreamReader;

public class MainActivity extends AppCompatActivity {

    private EditText notesEditText;
    private TextView notesTextView;
    private static final int REQUEST_WRITE_STORAGE = 112;

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

        notesEditText = findViewById(R.id.notesEditText);
        notesTextView = findViewById(R.id.notesTextView);
        Button saveButton = findViewById(R.id.saveButton);
        Button downloadButton = findViewById(R.id.downloadButton);

        // Request permissions for writing to external storage
        requestWriteStoragePermission();

        // Load saved notes when the app starts
        loadNotes();

        // Save notes when the button is clicked
        saveButton.setOnClickListener(v -> saveNotes());

        // Download notes to external storage when the button is clicked
        downloadButton.setOnClickListener(v -> downloadNotes());
    }

    private void requestWriteStoragePermission() {
        boolean hasPermission = (ContextCompat.checkSelfPermission(this,
```

```
                Manifest.permission.WRITE_EXTERNAL_STORAGE) == PackageManager.
PERMISSION_GRANTED);
        if (!hasPermission) {
            ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE},
                    REQUEST_WRITE_STORAGE);
        }
    }

      @Override
      public void onRequestPermissionsResult(int requestCode,
                                        @NonNull String[] permissions,
                                        @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);

    if (requestCode == REQUEST_WRITE_STORAGE) {
        if (grantResults.length > 0 &&
            grantResults[0] == PackageManager.PERMISSION_GRANTED) {

            Toast.makeText(this,
                           "Write permission granted",
                           Toast.LENGTH_SHORT).show();
        } else {
            Toast.makeText(this,
                           "Write permission denied",
                           Toast.LENGTH_SHORT).show();
        }
    }
}

    private void saveNotes() {
        String notes = notesEditText.getText().toString();

        try {
          FileOutputStream fos = openFileOutput("user_notes.txt", MODE_PRIVATE);
            fos.write(notes.getBytes());
            fos.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

        // Update the TextView to display the saved notes
        notesTextView.setText(notes);
    }

    private void loadNotes() {
        StringBuilder sb = new StringBuilder();
```

```
        try {
            FileInputStream fis = openFileInput("user_notes.txt");
            InputStreamReader isr = new InputStreamReader(fis);
            BufferedReader br = new BufferedReader(isr);
            String line;

            while ((line = br.readLine()) != null) {
                sb.append(line).append("\n");
            }

            fis.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

        String notes = sb.toString();
        // Update the EditText to display the loaded notes
        notesEditText.setText(notes);
        // Update the TextView to display the loaded notes
        notesTextView.setText(notes);
    }

    private void downloadNotes() {
          if (ContextCompat.checkSelfPermission(this,Manifest.permission.WRITE_
EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED) {
           Toast.makeText(this, "Write permission is required to download notes",
Toast.LENGTH_SHORT).show();
            return;
        }

        String notes = notesEditText.getText().toString();
        File externalStorageDir = Environment.getExternalStorageDirectory();
        File file = new File(externalStorageDir, "downloaded_notes.txt");

        try {
            FileWriter writer = new FileWriter(file);
            writer.write(notes);
            writer.close();

            // Show a toast message with the file location
        Toast.makeText(this, "File successfully downloaded to: "
                        + file.getAbsolutePath(), Toast.LENGTH_LONG).show();
```

```
        } catch (Exception e) {
            e.printStackTrace();
            Toast.makeText(this, "Failed to download file",
                            Toast.LENGTH_SHORT).show();
        }
    }
}
```

Explanation

1. **onCreate Method:**
   - **setContentView:** This method sets the layout for the activity using 'R.layout.activity_main', which refers to the XML layout file.
   - **Initialize UI Components:** The 'findViewById' method is used to link the UI components in the layout file to their corresponding fields in the activity code ('notesEditText', 'notesTextView', 'saveButton', 'downloadButton').
   - **requestWriteStoragePermission:** This method requests permission to write to external storage if not already granted. This is necessary to ensure the app can save files to the external storage.
   - **loadNotes:** This method is called to load any saved notes when the activity starts. It reads the saved notes from the file and updates the 'EditText' and 'TextView' to display them.
   - **setOnClickListener:** An 'OnClickListener' is set on the save button, which calls 'saveNotes' when the button is clicked. This saves the current notes entered by the user to a file. Another 'OnClickListener' is set on the download button, which calls 'downloadNotes' when the button is clicked. This saves the notes to external storage.
2. **saveNotes Method:**
   - **Retrieve Notes:** The method gets the text from the 'EditText' where the user has entered their notes.
   - **Open File Output Stream:** 'openFileOutput("user_notes.txt", Context.MODE_PRIVATE)' is used to create or open a file named "user_notes.txt". The 'MODE_PRIVATE' mode means the file is accessible only by the application.
   - **Write Data:** The 'FileOutputStream' object is used to write the notes to the file. The notes are converted to bytes using 'getBytes()' before being written to the file.
   - **Close Stream:** The 'close' method ensures all data is written to the file and the file output stream is properly closed.
   - **Update UI:** The 'TextView' is updated to display the saved notes, confirming to the user that their notes have been saved.
3. **loadNotes Method:**
   - **Open File Input Stream:** The method opens the "user_notes.txt" file in private mode for reading.
   - **Read Data:** The 'BufferedReader' and 'InputStreamReader' classes are used to read the notes from the file line by line. Each line read from the file is appended to a 'StringBuilder' with a newline character to preserve the formatting of the notes.

- **Close Stream:** The 'close' method ensures all data is read from the file and the file input stream is properly closed.
- **Update UI:** The notes read from the file are converted to a string and displayed in both the 'EditText' and 'TextView'. This ensures that the notes are visible and editable when the activity starts.

4. **downloadNotes Method:**
   - **Permission Check:** Checks if the WRITE_EXTERNAL_STORAGE permission is granted before attempting to write to external storage. If the permission is not granted, a toast message is displayed to inform the user.
   - **Retrieve Notes:** Gets the text from the 'EditText'.
   - **Create File in External Storage:** Uses 'Environment.getExternalStorageDirectory()' to get the external storage directory and creates a new file named "downloaded_notes.txt" in this directory.
   - **Write Data:** Uses 'FileWriter' to write the notes to the newly created file in external storage.
   - **Close Writer:** Ensures all data is written to the file and the writer is properly closed.
   - **Show Toast Message:** Displays a toast message to inform the user that the file has been successfully downloaded to the specified location. If the operation fails, a different toast message indicates the failure.

Output

When the application is run, the user can enter notes in the 'EditText'. These notes are saved when the "Save Notes Internally" button is clicked and loaded the next time the app is opened, providing a consistent user experience. Additionally, when the "Download Notes to Device Storage" button is clicked, the notes are saved to the external storage as 'downloaded_notes.txt', which can be accessed from the external storage directory. A toast message will notify the user of the successful download and the file location.

- **Figure 1: Initial App Launch:**

  When the app is opened for the first time, the 'EditText' for entering notes is empty, and the 'TextView' displays a message indicating that notes will be displayed here.

- **Figure 2: Entering and Saving Notes:**

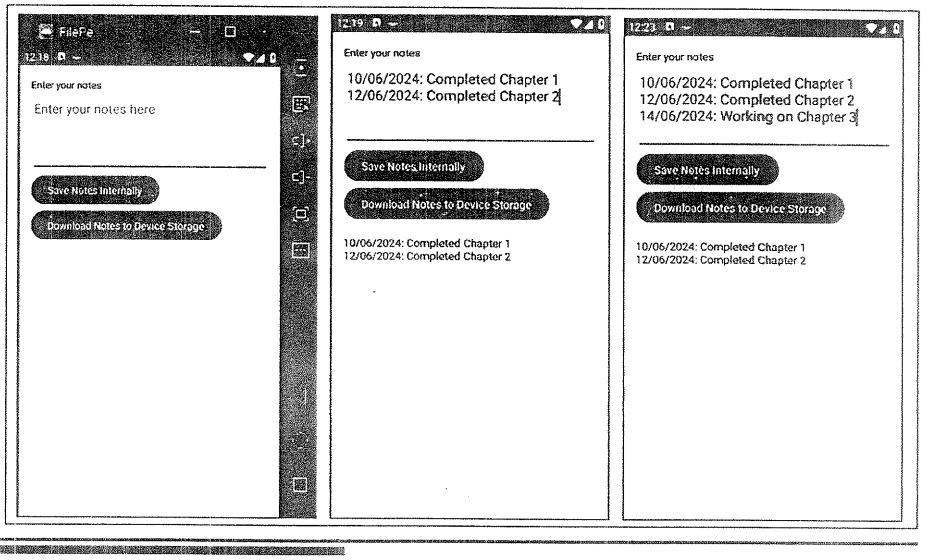
  After entering text into the 'EditText' and clicking on the "Save Notes Internally" button, the entered notes are saved to a file. The 'TextView' updates to display the saved notes, confirming the save action to the user.

- **Figure 3: Subsequent App Launch:**

  When the app is opened the next time, the previously saved notes are loaded from the file and displayed in both the 'EditText' and the 'TextView'. This allows the user to see their notes and continue editing if necessary.

- **Figure 4: Downloading Notes to External Storage:**

  After entering text into the 'EditText' and clicking on the "Download Notes to Device Storage" button, the notes are saved to the external storage as 'downloaded_notes.txt'. A toast message will notify the user of the successful download and the file location.



5.9.3 Creating and Using Databases

Using databases is essential for applications that need to store large amounts of structured data, perform complex queries, and ensure data integrity. Scenarios where databases are important include storing user information, maintaining app-specific data like notes or contacts, and keeping track of large datasets like inventory or student records.

SQLite in Android : Android provides built-in support for SQLite, a lightweight, disk-based database that doesn't require a separate server process. SQLite databases are ideal for mobile applications due to their simplicity and efficiency. The database created for an application is only accessible to that application; other applications will not be able to access it.

In this section, the process of programmatically creating a SQLite database in an Android application will be demonstrated. The SQLite database created programmatically in an application is always stored in the '/data/data/<package_name>/databases' folder.

How It Works

Generic Process:

1. **Define Database Schema:** Define the tables and columns for the database. This involves specifying the data types and constraints for each column.
2. **Create Helper Class:** Create a helper class that extends 'SQLiteOpenHelper'. This class will manage database creation and version management.
3. **Implement onCreate() Method:** Implement the 'onCreate()' method to execute SQL statements for creating tables when the database is first created.
4. **Implement onUpgrade() Method:** Implement the 'onUpgrade()' method to handle database schema changes when the database version is incremented.

5. **Open and Close Database:** Use methods to open and close the database connection.
6. **CRUD Operations:** Implement methods for Create, Read, Update, and Delete (CRUD) operations to interact with the database.

The DBHelper Class:

The 'DBHelper' class extends 'SQLiteOpenHelper' and is responsible for creating, opening, upgrading, and managing the database. This class encapsulates the complexities of database interactions, providing a simple interface for the rest of the application.

- **onCreate():** This method is called when the database is created for the first time. It is used to create tables and initialize the database.
- **onUpgrade():** This method is called when the database needs to be upgraded, typically by incrementing the database version. It handles the migration of database schemas.

Step-by-Step Implementation

Step 1: Create a Project:

Create a new Android project named 'DatabaseDemo'.

Step 2: Declare the Permission in the AndroidManifest.xml:

Add the 'WRITE_EXTERNAL_STORAGE' permission in the 'AndroidManifest.xml' (before <application> tag)

Code

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Step 3: Update the Activity Layout - activity_main.xml:

1. **Modify the XML Layout File:**

   Open the 'activity_main.xml' file and add UI elements for entering and displaying contacts.

2. **Add the below code for 'activity_main.xml':**

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:id="@+ic/nameLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Name" />
```

```
    <EditText
        android:id="@+id/nameEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter name" />

    <TextView
        android:id="@+id/emailLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Email" />

    <EditText
        android:id="@+id/emailEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter email" />

    <Button
        android:id="@+id/addButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Add Contact" />

    <Button
        android:id="@+id/viewAllButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="View All Contacts" />

    <TextView
        android:id="@+id/contactsTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="16dp"
        android:text="Contacts will be displayed here." />
</LinearLayout>
```

Explanation

- **LinearLayout:** The root layout uses a 'LinearLayout' to arrange the UI elements vertically.
- **TextView:** Displays labels for the name and email input fields.
- **EditText:** Allows the user to enter the name and email.
- **Button:** Two buttons for adding a contact to the database and viewing all contacts.
- **TextView:** Displays the saved contacts.

Step 4: Create the DBAdapter.java File:

Add a new Java Class file to the package and name it 'DBAdapter.java':

Code DBAdapter.java

```
package com.example.databasedemo;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class DBAdapter {
    static final String KEY_ROWID = "_id";
    static final String KEY_NAME = "name";
    static final String KEY_EMAIL = "email";
    static final String TAG = "DBAdapter";
    static final String DATABASE_NAME = "MyDB";
    static final String DATABASE_TABLE = "contacts";
    static final int DATABASE_VERSION = 1;
    static final String DATABASE_CREATE =
            "create table contacts (_id integer primary key autoincrement, " +
                    "name text not null, email text not null);";
    final Context context;
    DatabaseHelper DBHelper;
    SQLiteDatabase db;

    public DBAdapter(Context ctx) {
        this.context = ctx;
        DBHelper = new DatabaseHelper(context);
    }

    private static class DatabaseHelper extends SQLiteOpenHelper {
        DatabaseHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
            try {
                db.execSQL(DATABASE_CREATE);
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
```

```
        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
            // handle database upgrade
        }
    }

    //---opens the database---
    public DBAdapter open() throws SQLException {
        db = DBHelper.getWritableDatabase();
        return this;
    }

    //---closes the database---
    public void close() {
        DBHelper.close();
    }

    //---insert a contact into the database---
    public long insertContact(String name, String email) {
        ContentValues initialValues = new ContentValues();
        initialValues.put(KEY_NAME, name);
        initialValues.put(KEY_EMAIL, email);
        return db.insert(DATABASE_TABLE, null, initialValues);
    }

    //---retrieves all the contacts---
    public Cursor getAllContacts() {
        return db.query(DATABASE_TABLE, new String[]{KEY_ROWID, KEY_NAME,
                KEY_EMAIL}, null, null, null, null, null);
    }
}
```

Explanation

- **Constants:** Define constants for database fields, table name, and SQL create statement.
- DatabaseHelper: Inner class extending 'SQLiteOpenHelper' to manage database creation and version management.
- **onCreate:** Creates the database table.
- **onUpgrade:** Upgrades the database by dropping and recreating the table.
- **Database Operations:**
  - **open:** Opens the database for reading and writing.
  - **close:** Closes the database.
  - **insertContact:** Inserts a new contact into the database.
  - **getAllContacts:** Retrieves all contacts from the database.

Step 5: Update MainActivity.java:

Modify the 'MainActivity.java' file to interact with the database:

Code MainActivity.java

```
package com.example.databasedemo;

import android.database.Cursor;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private DBAdapter dbAdapter;
    private EditText nameEditText;
    private EditText emailEditText;
    private TextView contactsTextView;

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

        dbAdapter = new DBAdapter(this);

        nameEditText = findViewById(R.id.nameEditText);
        emailEditText = findViewById(R.id.emailEditText);
        contactsTextView = findViewById(R.id.contactsTextView);
        Button addButton = findViewById(R.id.addButton);
        Button viewAllButton = findViewById(R.id.viewAllButton);

        addButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                dbAdapter.open();
                dbAdapter.insertContact(nameEditText.getText().toString(),
                                        emailEditText.getText().toString());
                dbAdapter.close();
                nameEditText.setText("");
                emailEditText.setText("");
            }
        });
```

```
        viewAllButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                dbAdapter.open();
                Cursor cursor = dbAdapter.getAllContacts();
                StringBuilder result = new StringBuilder();
                if (cursor.moveToFirst()) {
                    do {
                        result.append("Name: ")
                              .append(cursor.getString(1)).append("\n")
                              .append("Email: ").append(cursor.getString(2))
                              .append("\n");
                    } while (cursor.moveToNext());
                }
                cursor.close();
                dbAdapter.close();
                contactsTextView.setText(result.toString());
            }
        });
    }
}
```

Explanation

Initialization: Initialize the 'DBAdapter', UI elements, and buttons.

- **addButton OnClickListener:**
  - Opens the database.
  - Inserts a new contact with the name and email entered in the EditText fields.
  - Closes the database.
  - Clears the EditText fields.
- **viewAllButton OnClickListener:**
  - Opens the database.
  - Retrieves all contacts from the database.
  - Displays the contacts in the TextView.
  - Closes the database.

Output

When the application is run, the user can enter a name and email in the EditText fields and click the "Add Contact" button to insert the contact into the database. Clicking the "View All Contacts" button displays all contacts in the TextView.

- **Figure 1: Initial App Launch:**

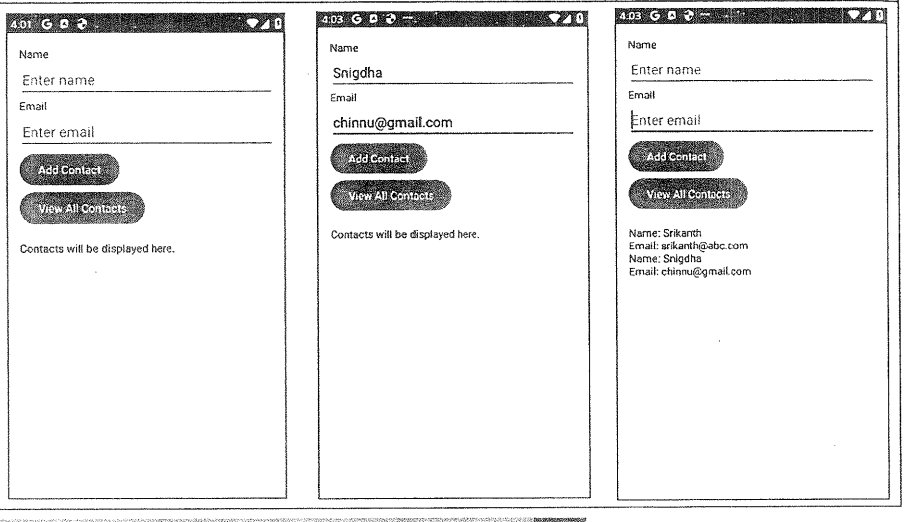
  When the app is opened for the first time, the EditText fields for entering name and email are empty, and the TextView displays a message indicating that contacts will be displayed here.

- **Figure 2: Entering and Adding Contacts:**

  After entering text into the EditText fields and clicking on the "Add Contact" button, the entered contact is saved to the database. The EditText fields are cleared, and the contact is stored.

- **Figure 3: Viewing All Contacts:**

  After clicking on the "View All Contacts" button, all previously saved contacts are retrieved from the database and displayed in the TextView. This allows the user to see the list of stored contacts.



How to Access and View the SQLite Database in Android Studio

- **Open Device File Explorer:**

  Navigate to 'View > Tool Windows > Device Explorer' in Android Studio.

- **Navigate to the Database Folder:**

  In Device Explorer, go to 'data > data > com.example.databasedemo > databases'.

- **Download the Database File:**

  Right-click on the database file (e.g., 'MyDB') and select 'Save As' to download it to the local machine.

- **Use a SQLite Viewer:**

  Use a SQLite viewer tool (such as DB Browser for SQLite) to open and inspect the downloaded database file on the local machine.

5.10 Sample Applications

5.10.1 Developing a Login Window Using UI Controls

Step 1: Create a New Project:

Create an Android project by selecting "Empty Views Activity", naming the project "LoginApp", and choosing Java as the language.

Step 2: Set Up the Login Activity Layout:

Edit the 'activity_main.xml' file to create a user interface for the login window.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <TextView
        android:id="@+id/label_username"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Username"
        android:textSize="18sp"
        android:layout_marginTop="50dp"
        android:layout_centerHorizontal="true"/>

    <EditText
        android:id="@+id/input_username"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter your username"
        android:layout_below="@id/label_username"
        android:layout_marginTop="10dp"
        android:padding="10dp"/>

    <TextView
        android:id="@+id/label_password"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Password"
        android:textSize="18sp"
        android:layout_below="@id/input_username"
        android:layout_marginTop="20dp"
        android:layout_centerHorizontal="true"/>

    <EditText
        android:id="@+id/input_password"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter your password"
```

```
        android:inputType="textPassword"
        android:layout_below="@id/label_password"
        android:layout_marginTop="10dp"
        android:padding="10dp"/>

    <Button
        android:id="@+id/button_login"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Login"
        android:layout_below="@id/input_password"
        android:layout_marginTop="30dp"
        android:padding="10dp"/>
</RelativeLayout>
'''
```

Explanation

- **RelativeLayout:** Used as the root layout, allowing child views to be positioned relative to each other.
- **TextView (Username Label):** Displays the label "Username" centered horizontally with a margin from the top.
- **EditText (Username Input):** Allows users to enter their username. It is positioned below the username label with a margin and padding for better UI experience.
- **TextView (Password Label):** Displays the label "Password" centered horizontally and below the username input field.
- **EditText (Password Input):** Allows users to enter their password. It is positioned below the password label, with an 'inputType' set to 'textPassword' for masking the input.
- **Button (Login Button):** A button that users can press to attempt a login. It is positioned below the password input field and spans the full width of the parent layout, with padding to make it visually appealing.

Step 3: Modify the MainActivity Java File:

Edit the 'MainActivity.java' file to reference the UI components. Since we're only focusing on the UI in this example, the code will simply set up the view without any additional functionality.

Code MainActivity.java

```
package com.example.loginapp;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

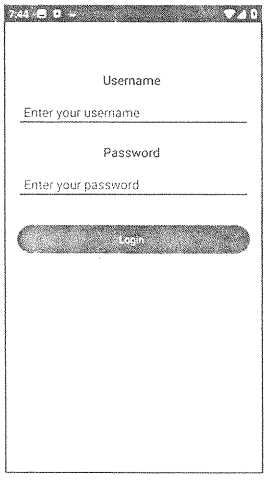
```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

      // UI elements are defined in the XML layout and no additional functionality
    }
}
```

Explanation

- The 'MainActivity' class extends 'AppCompatActivity'.
- The 'onCreate' method sets the content view to 'activity_main.xml', which contains the UI for the login screen.
- No additional functionality is added in this example since we're focusing on the UI.

Output Explanation

When the application runs, it displays a login window with a user interface containing two labels, two input fields (one for the username and one for the password), and a login button. This UI is designed to provide a clean and user-friendly login experience.



5.10.2 Developing Login Window and Validating User Credentials.

In this section, we will develop an Android application named "LoginAppValidation" that includes a login screen with validation. The app will check if the username and password fields are empty and validate the input against predefined credentials. If the login is successful, a toast message "Login Successful" will be displayed; otherwise, a "Login Fail" message will be shown.

Step 1: Create a New Project:

Create an Android project by selecting "Empty Views Activity", naming the project "LoginAppValidation", and choosing Java as the language.

Step 2: Set Up the Login Activity Layout:

Use the same 'activity_main.xml' file from the previous section to create a user interface for the login window.

- **EditText (Password Input):** Allows users to enter their password. It is positioned below the password label, with an 'inputType' set to 'textPassword' for masking the input.
- **Button (Login Button):** A button that users can press to attempt a login. It is positioned below the password input field and spans the full width of the parent layout, with padding to make it visually appealing.

Step 3: Modify the MainActivity Java File:

Edit the 'MainActivity.java' file to add functionality for validating the username and password fields, and displaying toast messages based on the validation.

Code MainActivity.java

```
package com.example.loginappvalidation;

import android.os.Bundle;
import android.text.TextUtils;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private EditText inputUsername;
    private EditText inputPassword;
    private Button buttonLogin;

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

        inputUsername = findViewById(R.id.input_username);
        inputPassword = findViewById(R.id.input_password);
        buttonLogin = findViewById(R.id.button_login);

        buttonLogin.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                validateLogin();
            }
        });
    }

    private void validateLogin() {
        String username = inputUsername.getText().toString().trim();
        String password = inputPassword.getText().toString().trim();
```

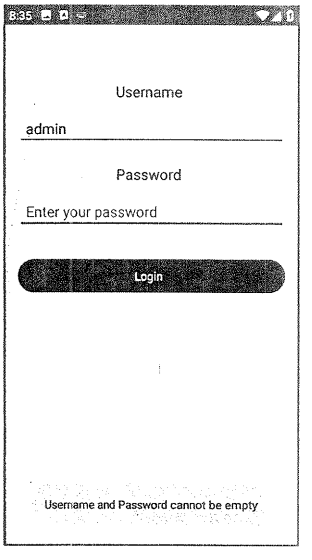
```
        if (TextUtils.isEmpty(username) || TextUtils.isEmpty(password)) {
            Toast.makeText(this, "Username and Password cannot be empty",
                           Toast.LENGTH_SHORT).show();
        } else {
            if (username.equals("admin") && password.equals("1234")) {
              Toast.makeText(this, "Login Successful", Toast.LENGTH_SHORT).show();
            } else {
                Toast.makeText(this, "Login Fail", Toast.LENGTH_SHORT).show();
            }
        }
    }
}
```

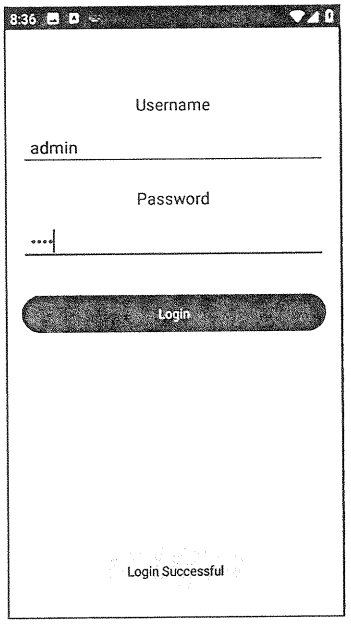
Explanation

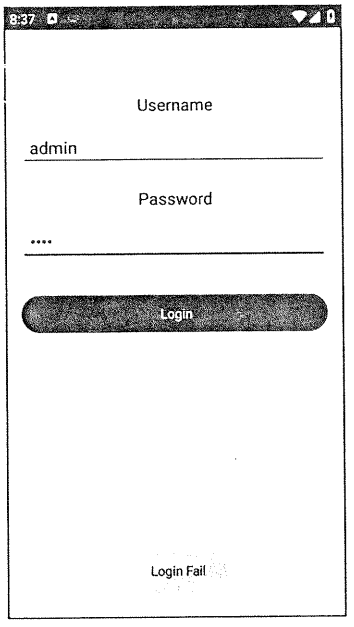
- **onCreate Method:** Sets the content view to 'activity_main.xml' and initializes the UI elements. It also sets an 'OnClickListener' on the login button.
- **validateLogin Method:** This method is called when the login button is clicked. It:
  - Retrieves the text from the username and password input fields.
  - Checks if either field is empty and shows a toast message if so.
  - Validates the credentials against predefined values ("admin" and "1234").
  - Displays a toast message "Login Successful" if the credentials are correct; otherwise, it displays "Login Fail".

Output

When the application runs, it displays a login window with a user interface containing two labels, two input fields (one for the username and one for the password), and a login button. If the username or password fields are empty, a toast message "Username and Password cannot be empty" is displayed. If the user enters the correct credentials ("admin" and "1234"), a toast message "Login Successful" is displayed. Otherwise, a "Login Fail" message is shown.







5.10.3 Creating an UI with All Views

In this section, we will create an Android application that includes a user interface with various UI components and link these components in the activity code. This project will demonstrate how to use common Android views such as TextView, EditText, Button, CheckBox, RadioButton, DatePicker, TimePicker, NumberPicker, and others.

Step 1: Create a New Project:

Create an Android project by selecting "Empty Views Activity", naming the project "AllViewsApp", and choosing Java as the language.

Step 2: Set Up the Main Activity Layout:

Edit the 'activity_main.xml' file to create a user interface with various UI components.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is a TextView"
        android:textSize="18sp"
        android:layout_marginBottom="16dp"/>

    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Enter your text here"
        android:layout_marginBottom="16dp"/>

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me"
        android:layout_marginBottom="16dp"/>

    <CheckBox
        android:id="@+id/checkBox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Check Me"
        android:layout_marginBottom="16dp"/>

    <RadioGroup
        android:id="@+id/radioGroup"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:layout_marginBottom="16dp">

        <RadioButton
            android:id="@+id/radioButton1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Option 1"/>
```

```
            <RadioButton
                android:id="@+id/radioButton2"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Option 2"/>
        </RadioGroup>

        <Switch
            android:id="@+id/switch1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Switch"
            android:layout_marginBottom="16dp"/>

        <Spinner
            android:id="@+id/spinner"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginBottom="16dp"/>

        <ProgressBar
            android:id="@+id/progressBar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="16dp"/>

        <DatePicker
            android:id="@+id/datePicker"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="16dp"/>

        <TimePicker
            android:id="@+id/timePicker"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="16dp"/>

        <NumberPicker
            android:id="@+id/numberPicker"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="16dp"/>
    </LinearLayout>

</ScrollView>
```

Explanation

- **ScrollView:** Allows vertical scrolling if the content is too long to fit on the screen.
- **LinearLayout:** Used as the root layout inside the ScrollView, with vertical orientation.
- **TextView:** Displays text.
- **EditText:** Allows user input.
- **Button:** A clickable button.
- **CheckBox:** A checkable box.
- **RadioGroup and RadioButton:** A group of radio buttons for selecting one option.
- **Switch:** A toggle switch.
- **Spinner:** A dropdown menu.
- **ProgressBar:** A progress indicator.
- **DatePicker:** Allows the user to select a date.
- **TimePicker:** Allows the user to select a time.
- **NumberPicker:** Allows the user to select a number within a specified range.

Step 3: Link the UI Components in MainActivity:

Edit the 'MainActivity.java' file to link these UI components using 'findViewById'.

Code MainActivity.java

```
package com.example.allviewsapp;

import android.os.Bundle;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.DatePicker;
import android.widget.EditText;
import android.widget.NumberPicker;
import android.widget.ProgressBar;
import android.widget.RadioButton;
import android.widget.RadioGroup;
import android.widget.Spinner;
import android.widget.Switch;
import android.widget.TextView;
import android.widget.TimePicker;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private EditText editText;
    private Button button;
```

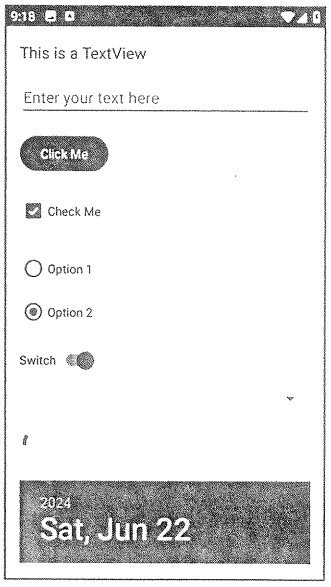
```
    private CheckBox checkBox;
    private RadioGroup radioGroup;
    private RadioButton radioButton1;
    private RadioButton radioButton2;
    private Switch switch1;
    private Spinner spinner;
    private ProgressBar progressBar;
    private DatePicker datePicker;
    private TimePicker timePicker;
    private NumberPicker numberPicker;

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

        // Link UI components
        textView = findViewById(R.id.textView);
        editText = findViewById(R.id.editText);
        button = findViewById(R.id.button);
        checkBox = findViewById(R.id.checkBox);
        radioGroup = findViewById(R.id.radioGroup);
        radioButton1 = findViewById(R.id.radioButton1);
        radioButton2 = findViewById(R.id.radioButton2);
        switch1 = findViewById(R.id.switch1);
        spinner = findViewById(R.id.spinner);
        progressBar = findViewById(R.id.progressBar);
        datePicker = findViewById(R.id.datePicker);
        timePicker = findViewById(R.id.timePicker);
        numberPicker = findViewById(R.id.numberPicker);

        // Additional configurations if needed
        numberPicker.setMinValue(0);
        numberPicker.setMaxValue(100);
    }
}
```

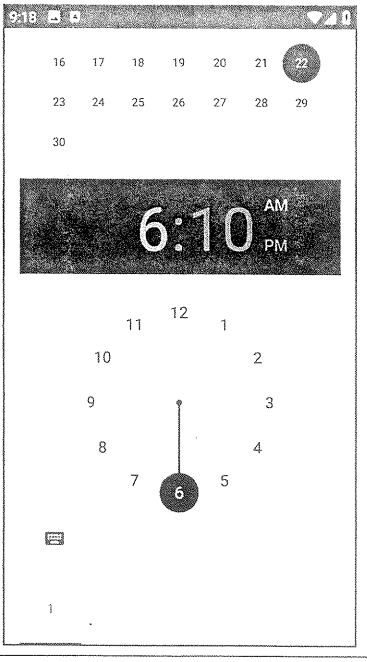
Explanation

- **onCreate Method:** Sets the content view to 'activity_main.xml' and initializes the UI elements using 'findViewById'.
- **Configurations:** Configures the 'NumberPicker' with a minimum and maximum value.

Output Explanation

When the application runs, it displays a UI containing various Android views like TextView, EditText, Button, CheckBox, RadioGroup with RadioButtons, Switch, Spinner, ProgressBar, DatePicker, TimePicker, and NumberPicker. The 'MainActivity' links these views so they can be accessed and manipulated in the activity code. This setup provides a comprehensive example of using different UI components in an Android application.





5.10.4 Creating an Application with a Custom Designed Opening Screen

In this section, we will create an Android application that includes a custom designed opening screen (splash screen). This splash screen will be displayed for a few seconds when the app is launched before transitioning to the main activity.

Step 1: Create a New Project:

Create an Android project by selecting "Empty Views Activity", naming the project "SplashScreenApp", and choosing Java as the language.

Step 2: Set Up the Splash Screen Layout:

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!-- Splash Screen Layout -->
    <RelativeLayout
        android:id="@+id/splash_screen"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="#FFA500"
        android:padding="16dp"
        android:gravity="center">

        <ProgressBar
            android:id="@+id/progress_bar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerHorizontal="true"
            android:indeterminate="true" />

        <TextView
            android:id="@+id/app_name"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Welcome to Skyward App"
            android:textSize="24sp"
            android:textColor="#FF000000"
            android:layout_below="@id/progress_bar"
            android:layout_centerHorizontal="true"
            android:layout_marginTop="16dp"/>
    </RelativeLayout>

    <!-- Main Content Layout -->
    <LinearLayout
        android:id="@+id/main_content"
```

```
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp"
        android:gravity="center"
        android:visibility="gone">

        <TextView
            android:id="@+id/welcome_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Welcome to the Main Activity Screen"
            android:textSize="18sp"
            android:textColor="#000000" />
    </LinearLayout>
</FrameLayout>
```

Explanation

- **FrameLayout:** Used as the root layout to overlap the splash screen and main content layouts.
- **RelativeLayout (Splash Screen Layout):** Contains a 'ProgressBar' and a 'TextView', both centered within the layout.
- **ProgressBar:** Displays an indeterminate progress bar centered horizontally.
- **TextView (App Name):** Displays the app name below the progress bar with custom styling.
- **LinearLayout (Main Content Layout):** Contains a 'TextView' for the main content, centered in the layout. This layout is initially hidden ('android:visibility="gone"').

Step 3: Modify the MainActivity Java File:

Edit the 'MainActivity.java' file to handle the splash screen logic and transition to the main content.

Code MainActivity.java

```
package com.example.splashscreenapp;

import android.os.Bundle;
import android.os.Handler;
import android.view.View;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private static final int SPLASH_DISPLAY_LENGTH = 10000; // Duration of splash
screen in milliseconds

    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

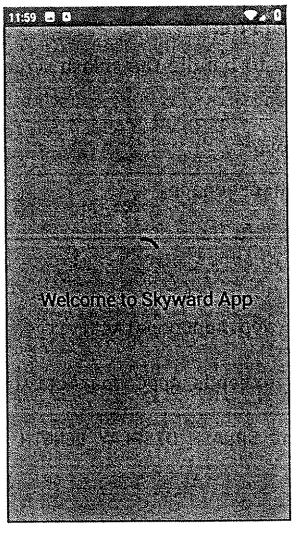
```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        final View splashScreen = findViewById(R.id.splash_screen);
        final View mainContent = findViewById(R.id.main_content);

        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
    // Hide the splash screen and show the main content after the splash duration
                splashScreen.setVisibility(View.GONE);
                mainContent.setVisibility(View.VISIBLE);
            }
        }, SPLASH_DISPLAY_LENGTH);
    }
}
```

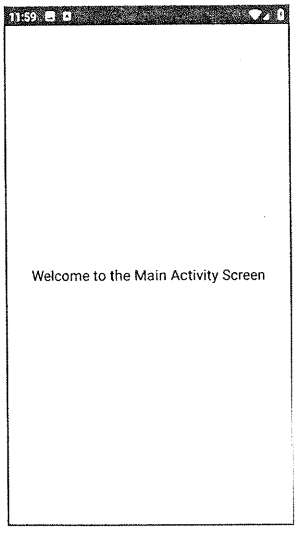
Explanation

- **SPLASH_DISPLAY_LENGTH:** Duration of the splash screen in milliseconds (10000ms = 10 seconds).
- **onCreate Method:** Sets the content view to 'activity_main.xml', finds the splash screen and main content views by their IDs, and uses a 'Handler' to delay the transition from the splash screen to the main content. After the delay, the splash screen is hidden and the main content is made visible.

Output

When the application runs, it displays a custom designed splash screen with a centered progress bar and app name for 10 seconds. After the splash screen duration, it transitions to the main content within the same activity, which displays a welcome message. This setup provides a smooth and visually appealing opening screen for your Android application using a single activity.





5.11 Review Questions

Section - A

1. What is a TextView in Android?
2. List two features of the TextView component.
3. What is an EditText view in Android?
4. What is a Button in Android?
5. How do you handle a click event for a Button in XML?
6. How do you enable or disable a Button programmatically?
7. What is an ImageButton in Android?
8. What is a CheckBox in Android?
9. What is a ToggleButton in Android?
10. How do you set the text for the "on" and "off" states of a ToggleButton?
11. What is a RadioButton in Android?
12. How do you group multiple RadioButtons together?
13. What is a ProgressBar in Android?
14. What is the difference between indeterminate and determinate modes in a ProgressBar?
15. What is an AutoCompleteTextView in Android?
16. What is a DatePicker in Android?
17. List two key features of DatePicker.
18. What is a TimePicker in Android?
19. Name two formats in which TimePicker can display time.
20. What is a NumberPicker in Android?
21. Which methods are used to set the minimum and maximum values of a NumberPicker?
22. What is a ListView in Android?
23. List two key features of ListView.
24. How does a ListView handle large datasets efficiently?
25. What is the purpose of an Adapter in a ListView?
26. What is a RecyclerView in Android?
27. Name two key features of RecyclerView.
28. What is the role of a ViewHolder in RecyclerView?
29. What is the main advantage of RecyclerView over ListView?
30. What is a SpinnerView in Android?
31. List two key features of SpinnerView.
32. What is a GridView in Android?

33. List two key features of GridView.
34. What is a ListFragment in Android?
35. List two key features of ListFragment.
36. How does ListFragment encapsulate ListView functionality?
37. What is a DialogFragment in Android?
38. List two key features of DialogFragment.
39. How does DialogFragment handle dialog lifecycle management?
40. What is a PreferenceFragment in Android?
41. List two key features of PreferenceFragment.
42. How are preferences defined in a PreferenceFragment?
43. What is an ImageView in Android?
44. List two key features of ImageView.
45. What is a GridView in Android?
46. List two key features of GridView.
47. What is an ImageSwitcher in Android?
48. List two key features of ImageSwitcher.
49. What is a WebView in Android?
50. List two key features of WebView.
51. What are Shared Preferences in Android?
52. List two key features of Shared Preferences.
53. What is the purpose of using File Systems in Android for data persistence?
54. List two key features of using File Systems for data storage.

Section -B

1. How to define a TextView in XML Layout File? Explain with an example.
2. What are the key features and attributes of a TextView?
3. Explain how to use TextView in Activity Code with an example.
4. Explain how to set up an EditText in XML with an example.
5. What are the key features of an EditText view?
6. How to define a Button in XML Layout File? Explain with an example.
7. Explain how to handle click events for a Button programmatically.
8. Explain how to define an ImageButton in XML with an example.
9. Describe the key features and use cases of an ImageButton.
10. How to define a CheckBox in XML Layout File? Explain with an example.
11. Describe the key features of a CheckBox.

12. Explain how to define a ToggleButton in XML Layout File with an example.
13. Explain the key features of a ToggleButton.
14. Explain how to define RadioButtons and group them in a RadioGroup in XML Layout File with an example.
15. Explain the key features of a RadioButton.
16. How to define a ProgressBar in XML Layout File? Explain with an example.
17. Explain how to define an AutoCompleteTextView in XML Layout File with an example.
18. Describe how to set up and use an adapter with AutoCompleteTextView.
19. Explain the key features of an AutoCompleteTextView.
20. How to define a DatePicker in XML Layout File? Explain with an example.
21. Describe the key features of a DatePicker.
22. Explain how to use DatePicker in Activity Code with an example.
23. How to define a TimePicker in XML Layout File? Explain with an example.
24. Describe the key features of a TimePicker.
25. Explain how to handle time changes in a TimePicker programmatically with an example.
26. How to define a NumberPicker in XML Layout File? Explain with an example.
27. Describe the key features of a NumberPicker.
28. Explain how to handle value changes in a NumberPicker programmatically with an example.
29. How to define a ListView in XML Layout File? Explain with an example.
30. Describe the key features of a ListView.
31. Explain how to use a ListView in Activity Code with an example.
32. How to define a RecyclerView in XML Layout File? Explain with an example.
33. Describe the key features of a RecyclerView.
34. Explain how to set up a RecyclerView with an Adapter and LayoutManager in Activity Code with an example.
35. How to define a SpinnerView in XML Layout File? Explain with an example.
36. Describe the key features of a SpinnerView.
37. Explain how to handle item selections in a SpinnerView programmatically with an example.
38. How to define a GridView in XML Layout File? Explain with an example.
39. Describe the key features of a GridView.
40. Explain how to set up a GridView with an Adapter in Activity Code with an example.
41. How to define a ListFragment in Android? Explain with an example.
42. Describe the key features of a ListFragment.
43. Explain how to handle item click events in a ListFragment programmatically with an example.
44. How to define a DialogFragment in Android? Explain with an example.

45. Describe the key features of a DialogFragment.
46. Explain how to create and display a custom DialogFragment with an example.
47. How to define a PreferenceFragment in Android? Explain with an example.
48. Describe the key features of a PreferenceFragment.
49. Explain how to manage user preferences using PreferenceFragment with an example.
50. How to define an ImageView in XML Layout File? Explain with an example.
51. Describe the key features of an ImageView.
52. Explain how to load and display an image from a URL in an ImageView programmatically with an example.
53. How to define a GridView in XML Layout File? Explain with an example.
54. Describe the key features of a GridView.
55. Explain how to bind data to a GridView using an adapter with an example.
56. How to define an ImageSwitcher in XML Layout File? Explain with an example.
57. Describe the key features of an ImageSwitcher.
58. Explain how to implement smooth transitions between images in an ImageSwitcher with an example.
59. How to define a WebView in XML Layout File? Explain with an example.
60. Describe the key features of a WebView.
61. Explain how to handle navigation and JavaScript in a WebView with an example.
62. How to save and load user preferences using Shared Preferences in Android? Explain with an example.
63. Describe the key features of Shared Preferences.
64. Explain how to store and retrieve different data types in Shared Preferences with an example.
65. How to write and read data to and from internal storage in Android? Explain with an example.
66. Describe the key features of using File Systems for data persistence.
67. Explain how to handle permissions and write data to external storage in Android with an example.
68. How to create and use a SQLite database in Android? Explain with an example.
69. Describe the key features of SQLite databases.
70. Explain how to perform CRUD operations in a SQLite database with an example.

Section - C

1. Explain TextView along with its attributes and features. Give an example.
2. Explain any eight key attributes for EditText and describe their uses.
3. Explain Button along with its attributes and features. Give an example.
4. Describe how to create custom buttons with advanced styling and click handling. Provide a detailed code example.

5. Explain ImageButton along with its attributes and features. Give an example.
6. Explain CheckBox along with its attributes and features. Give an example.
7. Explain ToggleButton along with its attributes and features. Give an example.
8. Explain RadioButton along with its attributes and features. Give an example.
9. Explain ProgressBar along with its attributes and features. Give an example.
10. Explain AutoCompleteTextView along with its attributes and features. Give an example.
11. Explain DatePicker along with its attributes and features. Give an example.
12. Discuss how to implement a DatePicker dialog and handle date selection in an Android application. Provide a detailed code example.
13. Explain TimePicker along with its attributes and features. Give an example.
14. Discuss how to implement a TimePicker dialog and handle time selection in an Android application. Provide a detailed code example.
15. Explain NumberPicker along with its attributes and features. Give an example.
16. Discuss how to create a custom NumberPicker with specific formatting and handle value changes in an Android application. Provide a detailed code example
17. Explain ListView along with its attributes and features. Give an example.
18. Discuss the importance of adapters in ListView and how to implement a custom adapter. Provide a detailed code example.
19. Explain RecyclerView along with its attributes and features. Give an example.
20. Discuss how to implement a custom RecyclerView Adapter and ViewHolder. Provide a detailed code example.
21. Explain SpinnerView along with its attributes and features. Give an example.
22. Discuss how to create a custom SpinnerView Adapter and handle item selections. Provide a detailed code example.
23. Explain GridView along with its attributes and features. Give an example.
24. Discuss how to create a custom GridView Adapter and handle item clicks. Provide a detailed code example.
25. Explain ListFragment along with its attributes and features. Give an example.
26. Discuss how to create a custom ListFragment and handle item selection. Provide a detailed code example.
27. Explain DialogFragment along with its attributes and features. Give an example.
28. Discuss how to implement a custom DialogFragment and handle user interactions. Provide a detailed code example.
29. Explain PreferenceFragment along with its attributes and features. Give an example.
30. Discuss how to create and manage a settings screen using PreferenceFragment. Provide a detailed code example.
31. Explain ImageView along with its attributes and features. Give an example.

32. Discuss how to implement image loading and caching in an ImageView using a third-party library. Provide a detailed code example.
33. Explain GridView along with its attributes and features. Give an example.
34. Discuss how to create a custom GridView adapter and handle item clicks. Provide a detailed code example.
35. Explain ImageSwitcher along with its attributes and features. Give an example.
36. Discuss how to create an image slideshow using ImageSwitcher. Provide a detailed code example.
37. Explain WebView along with its attributes and features. Give an example.
38. Discuss how to create a hybrid application using WebView and handle web page interactions. Provide a detailed code example.
39. Explain Shared Preferences along with its attributes and features. Give an example.
40. Discuss how to create a preference management system using Shared Preferences. Provide a detailed code example.
41. Explain the use of File Systems for data persistence in Android along with its features. Give an example.
42. Discuss how to implement file reading and writing operations in both internal and external storage. Provide a detailed code example.
43. Explain SQLite databases along with their attributes and features. Give an example.
44. Discuss how to create and manage a SQLite database with multiple tables and relationships. Provide a detailed code example.

UNIT - III

CHAPTER 6

DESIGNING USER INTERFACE (SUDOKU GAME APP)

CONTENTS

- Introduction
- Understanding Sudoku Game
- Designing by Declaration
- Creating the Opening Screen
- Using Alternate Resources
  - Using ScrollView
  - Using Different Layout for Landscape Mode
- Implementing an About Box
- Applying a Theme
- Adding a Menu
- Adding Settings
- Debugging
  - Debugging with Log Messages
  - Debugging with Debugger
- Review Questions

6.1 Introduction

Chapter 1 introduced us to Android development with a simple "Hello, Android" program. This set the stage for our journey. In Chapter 2, we explored the core structure and layout of Android apps. This helped build a solid foundation. Chapter 3 covered essential components like activities, intents, and fragments. These concepts gave us a deeper understanding of Android's building blocks. Chapters 4 and 5 expanded our knowledge by exploring user interfaces and various views in the Android ecosystem.

Now, in Chapter 6, we dive into a practical project by designing a user interface for a Sudoku game. This chapter bridges theory and application. It allows us to use what we've learned to create a visually appealing and user-friendly interface for the game. Through this hands-on project, we gain practical experience. This helps deepen our understanding of key elements in Android development.

The Sudoku game serves as a practical example. It helps us learn how to design interfaces that are intuitive, interactive, and visually pleasing. This exercise lets us grasp the intricacies of UI design within the context of a real-world application. It makes the learning process both engaging and effective. Let's design the UI for the Sudoku game.

6.2 Understanding Sudoku Game

The Sudoku game is a popular logic-based puzzle that involves filling a 9x9 grid with numbers so that each row, each column, and each of the nine 3x3 subgrids contain all the numbers from 1 to 9 without repetition. Let us understand the rules and gameplay of Sudoku:

	7		5	8	3		2	
	5	9	2			3		
3	4				6	5		7
7	9	5				6	3	2
		3	6	9	7	1		
6	8				2	7		
9	1	4	8	3	5		7	6
	3		7		1	4	9	5
5	6	7	4	2	9		1	3

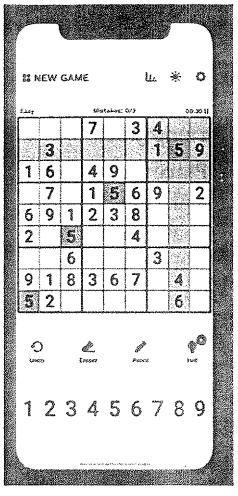
1. **Grid Structure:** The Sudoku grid consists of 81 cells, organized into a 9x9 layout. The grid is further divided into nine 3x3 subgrids.

2. **Objective:** The goal of Sudoku is to fill in the grid with numbers from 1 to 9 in such a way that each row, each column, and each 3x3 subgrid contains all the numbers exactly once.
3. **Given Numbers**: At the beginning of the game, some numbers are already provided in the grid. These are known as "givens" or "clues." The player's task is to complete the grid by filling in the remaining empty cells.
4. **Number Placement:** To solve the puzzle, players must use logic and deduction to determine the correct placement of numbers in each cell. The numbers must adhere to the rules of Sudoku to ensure that no row, column, or subgrid contains repeated numbers.
5. **Rules :**
   - Each row must contain the numbers 1 to 9, without repetition.
   - Each column must contain the numbers 1 to 9, without repetition.
   - Each 3x3 box must contain the numbers 1 to 9, without repetition.
6. **Logic and Strategy:** Solving Sudoku requires logical thinking and strategic planning. Players often use techniques such as scanning, cross-hatching, and elimination to deduce the correct numbers for each cell.
7. **Unique Solution:** A valid Sudoku puzzle has only one unique solution. This means that there is only one way to correctly fill in the grid while following the rules of the game.
8. **Difficulty Levels :** Sudoku puzzles come in varying levels of difficulty, ranging from easy to hard. The complexity of the puzzle is determined by the number and placement of initial givens.
9. **Error Avoidance:** In Sudoku, making a mistake can lead to an unsolvable puzzle. Therefore, players must be careful and methodical in their number placements to avoid errors.

The solution to the above sudoku is given below

1	7	6	5	8	3	9	2	4
8	5	9	2	7	4	3	6	1
3	4	2	9	1	6	5	8	7
7	9	5	1	4	8	6	3	2
4	2	3	6	9	7	1	5	8
6	8	1	3	5	2	7	4	9
9	1	4	8	3	5	2	7	6
2	3	8	7	6	1	4	9	5
5	6	7	4	2	9	8	1	3

Traditionally, Sudoku is played using pencil and paper, but digital versions have gained popularity. In the paper format, errors can lead to significant rework, requiring the erasure of previous entries. In contrast, the mobile app version offers the advantage of easily modifying entries without the hassle of erasing mistakes manually. The Sudoku example program for Android is shown in the figure.



6.3 Designing by Declaration

In the context of designing user interfaces, two main approaches are commonly used: **procedural** and **declarative**.

1. **Procedural Design:** Procedural design involves creating user interfaces using code. For example, in a Swing application, developers write Java code to define and manipulate UI components like JFrame and JButton. This approach is known for its programmatic nature, where the logic and structure of the UI are defined through code.
2. **Declarative Design:** Declarative design, on the other hand, focuses on describing the UI in a markup language without the need for extensive code. For example, when designing a web page, HTML is used to specify the layout and content presentation without detailing the implementation logic. XML is another markup language commonly used for declarative UI design.

Android's Approach

- Android provides flexibility by supporting both procedural (Java code) and declarative (XML) approaches for creating user interfaces. Developers can choose to work predominantly in Java or XML to define the UI components and their attributes.
- Android documentation typically includes information on both the Java APIs and the corresponding XML attributes for UI components, allowing developers to select their preferred method.

Google Recommendation :

Google suggests utilizing declarative XML for designing Android user interfaces whenever possible. XML code is often more concise, easier to comprehend, and less prone to changes in future Android versions compared to Java code.

What is Designing by Declaration?

Designing by declaration refers to the approach of creating user interfaces by defining their structure and appearance using declarative markup languages such as XML, rather than writing procedural code. In this method, developers specify the layout, properties, and behavior of UI components through descriptive markup to focus on what should be displayed rather than how it should be implemented.

In the context of Android app development, designing by declaration involves using XML files to define the visual elements and attributes of the user interface. By separating the UI design from the application logic, developers can achieve a more modular and maintainable codebase. This approach allows for a clear separation of concerns, making it easier to manage and update the UI independently of the underlying functionality.

Advantages of Declarative Design

1. **Clarity and Readability:** XML markup is often more readable and intuitive than complex procedural code for developers to understand and modify the UI layout in easy manner.
2. **Ease of Maintenance:** Changes to the UI can be made more efficiently by editing the declarative markup, without the need to modify the application logic.
3. **Consistency:** Declarative design promotes consistency in UI elements and layouts across the application to enhance the overall user experience.
4. **Collaboration:** Designers and developers can collaborate effectively by working on separate aspects of the UI, with designers focusing on the visual presentation using XML files.

6.4 Creating the Opening Screen

An opening screen. also known as a splash screen is the starting page of an app. The opening screen serves as the entry point to the application and often provides essential information, options, or actions for users to interact with. This screen is crucial as it provides the first impression and navigational options for the user.

Key Characteristics and Components of an Opening Screen

The opening screen sets the tone for the user experience and serves as a gateway to the application's features and functionalities. It is designed to capture users' attention, provide a seamless entry point into the application, and guide users towards their intended actions or destinations within the app. The key characteristics and components of an openning screen are listed below.

- **Introduction:** The opening screen may introduce the application, provide a brief overview of its purpose, or display the application's logo or branding elements.
- **Navigation:** It typically offers navigation options for users to move to different sections of the application such as starting a new task, accessing settings, viewing tutorials, or exploring features.
- **Call to Action:** The opening screen may include prominent buttons or prompts that encourage users to take specific actions such as signing in, creating an account, or starting a new activity.

- **Visual Design:** The design of the opening screen plays a crucial role in creating a positive first impression on users. It often incorporates visually appealing elements, colors, and graphics to engage users.
- **Functionality:** Depending on the application's purpose, the opening screen may include interactive elements, animations, or multimedia content to enhance user engagement.
- **Loading Indicators:** In cases where the application requires time to load or initialize, the opening screen may display loading indicators or progress bars to inform users about the process.

The opening screen should be visually appealing, user-friendly, and offer clear navigation to ensure users can easily access the main features of the app.

UI Requirements for the Sudoku Opening Screen

For a Sudoku game, the opening screen serves as the gateway to the various functionalities of the game. The specific requirements include:

1. **Starting a New Game :** A prominent button that allows users to start a new Sudoku game session.
2. **Continuing a Previous Game :** A button that enables users to resume a Sudoku game they had previously saved.
3. **Viewing Information About Sudoku :** A button to display instructions or information about how to play Sudoku, including rules and tips.
4. **Exiting the App :** A button to close the Sudoku application.

The Sudoku opening screen should be designed with the following considerations:

- **Layout :** Use a simple and intuitive layout that guides the user to their desired action without confusion.
- **Design :** Incorporate a visually appealing design that matches the theme of Sudoku, possibly using calm and neutral colors to reflect the puzzle nature of the game.
- **User Experience :** Ensure the screen is responsive and accessible, with buttons that are easy to read and interact with.

By focusing on these aspects, the opening screen will not only serve its functional purposes but also enhance the overall user experience, making the app inviting and easy to use.

Step-by-Step Process for Creating the Sudoku Opening Screen

Step 1. Project Setup:

The first step in creating any Android application is to set up the project. This involves defining the project name, package name, language and minimum SDK version, among other settings. This ensures that the application is correctly configured and compatible with the desired range of Android devices.

1. Open Android Studio.
2. Start a New Android Studio Project.
3. Select "Empty View Activity" Template.

4. Name Your Project: 'Sudoku'.
5. Set Language: Java.
6. Set the minimum SDK to "API 21: Android 5.0 (Lollipop)".
7. Finish Project Setup

Step 2. Define the Layout in XML:

Layouts in Android define the structure of the user interface. XML files are used to create a declarative layout, which is preferred over procedural layouts due to its simplicity and readability. The layout file specifies how UI elements are arranged on the screen. The below code create and define the layout for the opening screen of the Sudoku game using XML.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/background"
    android:padding="30dp">

    <TextView
        android:id="@+id/main_title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/main_title"
        android:textSize="24sp"
        android:layout_gravity="center"
        android:layout_marginTop="100dp"
        android:layout_marginBottom="25dp"
        android:textColor="@color/text_color" />

    <Button
        android:id="@+id/continue_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/continue_label" />

    <Button
        android:id="@+id/new_game_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/new_game_label" />
```

```
    <Button
        android:id="@+id/about_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/about_label" />

    <Button
        android:id="@+id/exit_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/exit_label" />
</LinearLayout>
```

Explanation

This layout XML defines a simple and clean opening screen for the Sudoku game, with a centered title and four buttons arranged vertically for different actions.

1. **LinearLayout Container:**
   - The root element of the layout is a '<LinearLayout>'. This container arranges its child views in a single column (vertical orientation).
   - Attributes such as 'android:layout_width="match_parent"' and 'android:layout_height="match_parent"' ensure that the layout fills the entire screen. The background color is set using a reference to a color resource ('@color/background'), and padding is applied ('android:padding="30dp"') to provide space around the edges.
2. **TextView for Title:**
   - Inside the 'LinearLayout', a '<TextView>' is used to display the main title of the app. The 'android:id="@+id/main_title"' attribute assigns a unique ID to this 'TextView' so it can be referenced in the code.
   - The text to be displayed is referenced from a string resource ('@string/main_title'). Other attributes include 'android:textSize="24sp"' for text size, 'android:layout_gravity="center"' to center the title horizontally, 'android:layout_marginBottom="25dp"' for spacing below the title, and 'android:textColor="@android:color/white"' to set the text color to white.
3. **Continue Button:**
   - A '<Button>' is defined for continuing a previous game. The 'android:id="@+id/continue_button"' attribute assigns a unique ID to this button.
   - The button text is set using a reference to a string resource ('@string/continue_label'). The 'android:layout_width="match_parent"' attribute makes the button span the full width of the parent container, while 'android:layout_height="wrap_content"' allows the button height to adjust based on its content.
4. New Game Button:
   - Another '<Button>' is defined for starting a new game. It has the ID '@+id/new_game_button' and the text is set from a string resource ('@string/new_game_label').
   - Similar to the continue button, the layout width and height are set to 'match_parent' and 'wrap_content', respectively, ensuring consistent button sizing.

5. **About and Exit Buttons:**
   - Two more '<Button>' elements are defined for displaying information about the game ('@+id/about_button' with text '@string/about_label') and for exiting the app ('@+id/exit_button' with text '@string/exit_label').
   - Both buttons follow the same layout attributes as the previous buttons, ensuring they fill the parent container's width and adjust their height based on content.

Step 3. Define Strings in strings.xml:

String resources allow text strings to be defined separately from the code and hence it is easier to manage and localize text in the application. The below code define the text strings used in the layout in the 'strings.xml' file.

Code res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Sudoku</string>
    <string name="main_title">Android Sudoku</string>
    <string name="continue_label">Continue</string>
    <string name="new_game_label">New Game</string>
    <string name="about_label">About</string>
    <string name="exit_label">Exit</string>
</resources>
```

Explanation

- **<resources>:** The root element for defining resources.
- **<string name="...">:** Defines a string resource with a specific name and value.

For example, '<string name="app_name">Sudoku</string>' defines a string resource named 'app_name' with the value "Sudoku".

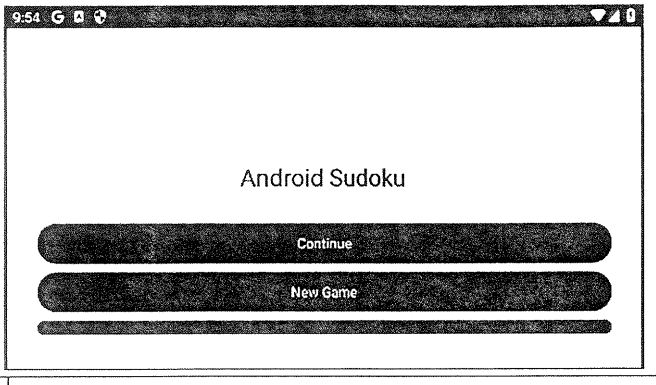
These string resources are referenced in the layout XML file using the '@string/' syntax, allowing for easy updates and localization.

Step 4. Define Colors in colors.xml:

Color resources allow colors to be defined and reused throughout the application to ensure a consistent color scheme. The below code define the background color used in the layout in the 'colors.xml' file.

Code res/values/colors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="background">#FFFFFF</color>
    <color name="text_color">#000000</color>
</resources>
```



Explanation

- **package org.example.sudoku;:** Declares the package for the class.
- **import statements:** Import necessary classes from the Android and AndroidX libraries.
- **MainActivity extends AppCompatActivity:** Defines the main activity class, inheriting from 'AppCompatActivity' to ensure compatibility with older Android versions.
- **onCreate(Bundle savedInstanceState):** This method is called when the activity is first created. It is used to perform the initial setup for the activity.
- **super.onCreate(savedInstanceState):** Calls the superclass's 'onCreate' method to perform any setup required by the parent class.
- **setContentView(R.layout.activity_main):** Sets the layout for the activity using the layout resource defined in 'activity_main.xml'.
- **findViewById(R.id....):** Finds views by their ID and assigns them to variables for further manipulation.
- **setOnClickListener(new View.OnClickListener()):** Sets click listeners for the buttons to handle user interactions.
- **continueButton.setOnClickListener(...):** Sets a click listener for the "Continue" button. The code inside the 'onClick' method will be executed when the button is clicked.
- **newGameButton.setOnClickListener(...):** Sets a click listener for the "New Game" button.
- **aboutButton.setOnClickListener(...):** Sets a click listener for the "About" button.
- **exitButton.setOnClickListener(...):** Sets a click listener for the "Exit" button. The 'finish()' method is called to close the activity when the button is clicked.

6.5 Using Alternate Resources

Switching the emulator or physical device to landscape mode may result in a few buttons running off the bottom of the screen. This issue can be observed in the landscape mode output shown in the previous section. Adjusting the layout to work with all orientations can be challenging and often leads to odd-looking screens. When this occurs, it is necessary to create a different layout specifically for landscape mode or use a 'ScrollView'. These approaches ensure that the application maintains a user-friendly and visually appealing interface in both orientations.

6.5.1 Using ScrollView

A ScrollView is a layout container that enables scrolling for a single child view. It is useful when the content is too large to fit on the screen. By wrapping the existing layout in a ScrollView, the user can scroll through the content to ensure all buttons are accessible regardless of the screen orientation.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/background"
    android:padding="30dp">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@color/background"
        android:padding="30dp">

        <TextView
            android:id="@+id/main_title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/main_title"
            android:textSize="24sp"
            android:layout_gravity="center"
            android:layout_marginTop="100dp"
            android:layout_marginBottom="25dp"
            android:textColor="@color/text_color" />

        <Button
            android:id="@+id/continue_button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/continue_label" />

        <Button
            android:id="@+id/new_game_button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/new_game_label" />
```

```
        <Button
            android:id="@+id/about_button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/about_label" />

        <Button
            android:id="@+id/exit_button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/exit_label" />
    </LinearLayout>
</ScrollView>
```

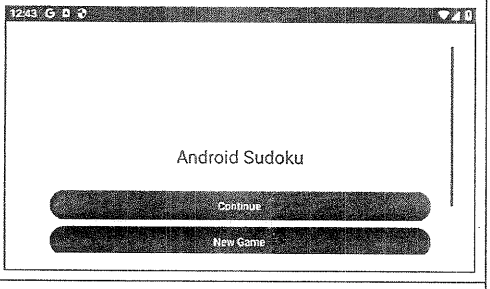
Explanation

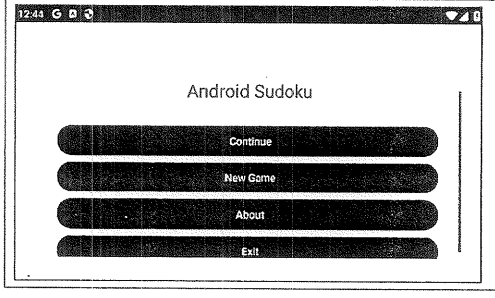
- **ScrollView:** The root element is a 'ScrollView', which allows the entire content to be scrollable vertically. This ensures that all buttons are accessible, even if they don't fit within the available vertical space on the screen.
- **LinearLayout:** The 'LinearLayout' is placed inside the 'ScrollView'. Its height is set to 'match_parent', allowing it to expand only as much as needed to fit its content.
- **TextView and Buttons:** The 'TextView' and 'Button' elements are nested within the 'LinearLayout', to ensure that they are displayed in a vertical stack.

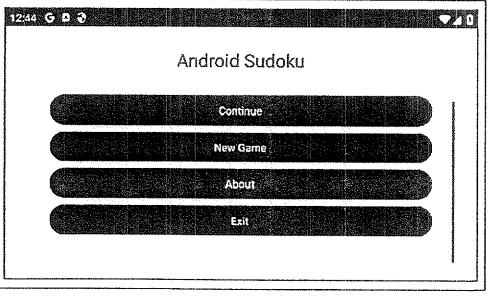
By using a 'ScrollView', the layout can adapt to different screen sizes and orientations, ensuring all UI elements are accessible.

Output

The below output demonstrate the use of a ScrollView in both portrait and landscape modes for the Android Sudoku application's opening screen. This ensures all buttons—Continue, New Game, About, and Exit—are accessible to the user. The scroll bar remains always visible for users to navigate vertically through the options seamlessly, regardless of screen orientation. This design choice enhances the user experience by maintaining a consistent and clean interface while accommodating varying screen sizes and orientations.







6.5.2 Using Different Layout for Landscape Mode

Creating a separate layout file for landscape mode allows for better optimization of the UI for different orientations. This approach involves defining a distinct layout that rearranges or resizes elements to fit appropriately within the horizontal space available in landscape mode.

Code res/layout-land/activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/background"
    android:padding="30dp"
    android:gravity="center">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:gravity="center">

        <TextView
            android:id="@+id/main_title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/main_title"
            android:textSize="24sp"
            android:layout_gravity="center"
            android:layout_marginBottom="25dp"
            android:textColor="@color/text_color" />

        <Button
            android:id="@+id/continue_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/continue_label"/>

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

```
            android:layout_height="wrap_content"
            android:text="@string/new_game_label"/>

        <Button
            android:id="@+id/about_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/about_label"/>

        <Button
            android:id="@+id/exit_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/exit_label"/>
    </LinearLayout>

</LinearLayout>
```

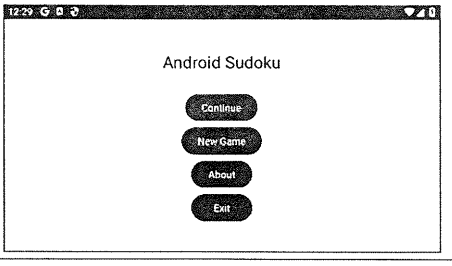
Explanation

- **Separate Layout File:** The layout file is placed in the 'res/layout-land' directory, which is automatically used by the Android system when the device is in landscape mode.
- **Horizontal LinearLayout:** The root element is a 'LinearLayout' with horizontal orientation, allowing the buttons and title to be arranged side by side, optimizing the use of horizontal space.
- **Vertical Inner LinearLayout:** Inside the horizontal 'LinearLayout', a vertical 'LinearLayout' is used to stack the title and buttons vertically. The 'android:layout_weight="1"' attribute ensures that the inner layout expands to fill the available vertical space, centering the elements.

By using a different layout for landscape mode, the UI can be tailored to better fit the available screen space, ensuring all elements are visible and accessible.

Output

The output image shows the application's opening screen in landscape mode with vertically centered buttons and a title. This layout ensures all UI components are visible and accessible.



6.6 Implementing an About Box

The About box is an essential feature that provides users with information about the application. An About box provides users with information about the application to enhance the overall user experience by giving insights into what the app does, its purpose, or any other relevant information. In the context of the Sudoku app, the About box will display a brief description of Sudoku, how it's

played, and its objectives. When the user selects the About button, the app will display a new screen (activity) containing the information about Sudoku. The user can read through the information and then press the Back button to return to the main screen.

The Steps to Implement an About Box

To create an About box that provides information about the Sudoku game when the user selects the About button, follow these steps. This process involves adding an activity and setting it up to display information about the game.

1. **Define a New Activity:** This activity will display the About information.
2. **Create a Layout File for the Activity:** The layout will include a ScrollView containing a TextView for displaying the content.
3. **Add String Resources:** These resources will contain the title and content for the About box.
4. **Create the Activity Class:** This class will set up the layout and handle the display of the content.
5. **Wire Up the About Button:** In the main activity, set up a click listener for the About button to start the new activity.
6. **Update AndroidManifest.xml:** Declare the new activity in the manifest file so the system recognizes it.

Step-by-Step Implementation of an About Box in Sudoku App

Step 1: Define the Layout for the About Activity:

Create a new XML layout file 'about.xml' in the 'res/layout' directory.

Code 'res/layout/about.xml'

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="10dp">
    <TextView
        android:id="@+id/about_content"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
            android:textSize="10pt"
        android:text="@string/about_text" />
</ScrollView>
```

Explanation

- **ScrollView:** Makes the content scrollable if it exceeds the screen size.
- **TextView:** Displays the About text, referenced from a string resource.

Step 2: Define Strings in 'strings.xml':

Add the following strings to the 'res/values/strings.xml' file.

Code 'res/values/strings.xml'

```
<string name="about_title">About Android Sudoku</string>
<string name="about_text">
    Sudoku is a logic-based number placement puzzle.
    Starting with a partially completed 9x9 grid, the
    objective is to fill the grid so that each
    row, each column, and each of the 3x3 boxes
    (also called <i>blocks</i>) contains the digits
    1 to 9 exactly once.
</string>
```

Explanation

- **about_title:** Title for the About activity.
- **about_text:** Text content for the About activity, providing information about Sudoku.

Step 3: Create the About Activity:

Create a new Java class 'About.java' in the 'src/main/java/com/example/sudoku' directory.

Code 'About.java'

```
package com.example.sudoku;

import android.app.Activity;
import android.os.Bundle;

public class About extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.about);
    }
}
```

Explanation

- **About Class:** Defines a new activity that extends 'Activity'.
- **onCreate Method:** Sets the content view to 'about.xml' when the activity is created.

Step 4: Modify MainActivity to Wire Up the About Button:

Update 'MainActivity.java' to handle the About button click and start the About activity.

- **Changes to 'MainActivity.java':**
  1. **Add imports:**

Code

```
import android.content.Intent;
```

2. **Set click listener for the About button in the 'onCreate' method:**

Code

```
Button aboutButton = findViewById(R.id.about_button);
aboutButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(MainActivity.this, About.class);
        startActivity(intent);
    }
});
```

Explanation

- **Intent to Start About Activity:** In the 'aboutButton.setOnClickListener' method, an 'Intent' is created to start the 'About' activity.
- **startActivity Method:** This method is called with the 'Intent' to launch the About activity.

Step 5: Declare the New Activity in AndroidManifest.xml:

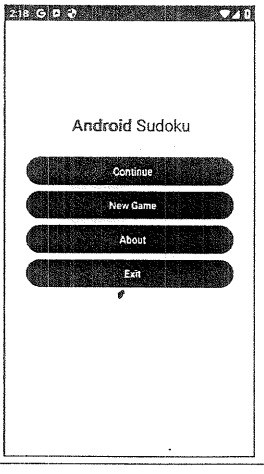
Add the following activity declaration to the 'AndroidManifest.xml' file.

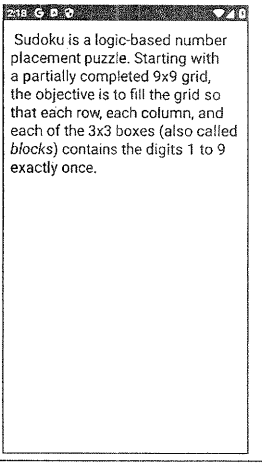
Code 'AndroidManifest.xml'

```
<activity android:name=".About"
    android:label="@string/about_title">
</activity>
```

Output

When users click the About button in the Sudoku application's main screen, the About activity is launched, displaying information about Sudoku in a scrollable view. The Back button can be used to dismiss the window and return to the main screen. This implementation provides users with detailed information about the game.





Explanation
• **Activity Declaration:** Declares the About activity so it can be launched by the application.

6.7 Applying a Theme

A theme in Android is a collection of style attributes that define the look and feel of an application or activity. Themes are similar to Cascading Style Sheets (CSS) used in web development, as they separate the content from the presentation. This separation allows developers to apply a consistent look and feel across the entire app or specific components without modifying the underlying functionality.

Importance and Benefits of Applying a Theme

1. **Consistency:** Themes ensure a consistent user interface throughout the application. By defining a set of styles and applying them globally, all UI components adhere to the same visual standards.
2. **Reusability:** Themes allow for the reuse of style attributes. Instead of setting styles individually for each component, a theme can apply a uniform style across multiple activities or views.
3. **Maintainability:** With themes, updating the look and feel of the app is straightforward. Changes to the theme are reflected throughout the app, making it easier to maintain and update the design.
4. **Customization:** Themes provide the flexibility to customize and extend default styles. Developers can create their own themes or extend existing ones to tailor the UI to their specific needs.
5. **User Experience:** A well-designed theme enhances the user experience by providing a visually appealing and coherent interface. It makes the app look professional and polished.

Step-by-Step Implementation of Applying a Theme

Step 1: Applying a Default Theme:

The first step is to modify the 'AndroidManifest.xml' file to apply a predefined theme to the About activity.

- **Modify the About Activity Declaration**

  Update the About activity declaration to include the 'android:theme' property.

Code 'AndroidManifest.xml'

```
<activity android:name=".About"
    android:label="@string/about_title"
    android:theme="@android:style/Theme.Dialog">
</activity>
```

Explanation
• **android:theme:** This attribute applies the specified theme to the activity. The '@android:style/Theme.Dialog' reference points to a predefined theme that styles the activity as a dialog.
• **@android:** prefix: Indicates that the referenced resource is defined by Android, not by the application.

Step 2: Applying a Custom Theme:

If a predefined theme doesn't meet the requirements, a custom theme can be created by defining styles in 'res/values/styles.xml'.

- **Define a Custom Theme:**

Code 'res/values/styles.xml'

```
<resources>
    <style name="CustomDialogTheme" parent="Theme.AppCompat.Dialog">
        <item name="android:windowBackground">@color/background</item>
        <item name="android:textColor">@color/text_color</item>
        <!-- Add more style attributes as needed -->
    </style>
</resources>
```

Explanation

- **CustomDialogTheme:** A custom theme that extends an existing theme ('Theme.AppCompat.Dialog'). Additional style attributes can be defined to customize the look and feel of the dialog.

- **Apply the Custom Theme:**

Update the About activity declaration to include the custom theme in the 'AndroidManifest.xml'.

Code 'AndroidManifest.xml'

```
<activity android:name=".About"
    android:label="@string/about_title"
    android:theme="@style/CustomDialogTheme">
</activity>
```

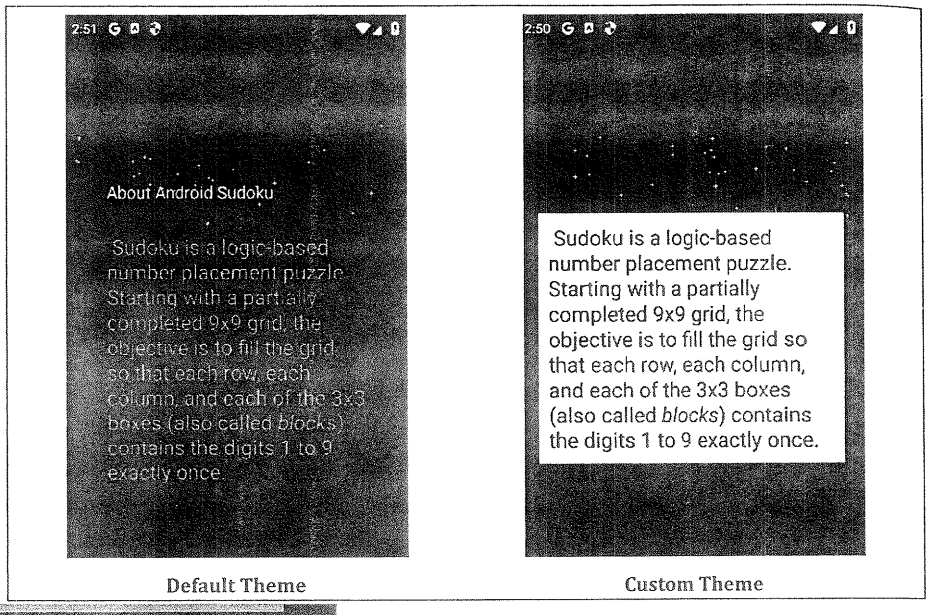
Explanation

- **android:theme:** Applies the custom theme to the activity. The '@style/CustomDialogTheme' reference points to the custom theme defined in 'styles.xml'.

Output

The About box will now appear as a dialog, making it visually distinct from the rest of the application. This style change helps in emphasizing the informational nature of the About box and makes it look more integrated with the overall app design.

When users click the About button, the About activity is displayed as a dialog. This provides a clear and focused presentation of the information about Sudoku, with the ability to scroll through the text if needed. The dialog can be dismissed by pressing the Back button, returning the user to the main screen.



Default Theme Custom Theme

Different between Styles vs Themes

Feature	Style	Theme
Definition	A set of attributes that can be applied to a specific view or view group.	A set of attributes that can be applied to an entire activity or application.
Application	Applied directly to views in layout XML files or programmatically.	Applied to activities or the entire application in the AndroidManifest.xml or programmatically.
Inheritance	Can inherit styles from other styles using the parent attribute.	Can inherit themes from other themes using the parent attribute.
Scope	Applied only to the view or view group to which it is assigned.	Applied globally to all views or activities that use the theme.
Attribute Override	Can override specific attributes of a view, such as text color or background.	Can override theme attributes for specific views or view groups using theme overlays.
Reusability	Can be reused across different views or view groups within the same app.	Can be reused across different activities or the entire app.
Use Cases	Used for defining view-specific appearance, such as text color or background.	Used for defining app-wide appearance, such as color scheme or font family.

6.8 Adding a Menu

Android supports two kinds of menus: the **options menu**, which appears when the user presses the physical Menu button or taps the overflow menu icon (three vertical dots) on modern devices, and

the **context menu**, which pops up when the user presses and holds on the screen. Let us discuss on how to add an options menu that opens when the user interacts with the the overflow menu icon to access to settings and other options.

Step-by-Step Implementation

Step 1: Change the Theme to Include the ActionBar:

This step is to ensure the application theme includes an ActionBar to display the menu. The ActionBar provides a familiar navigation and menu interaction for users.

We need to modify the 'styles.xml' file to extend a theme that includes the ActionBar. This involves setting the base theme to 'Theme.Material3.DayNight', which includes ActionBar support.

Code 'res/values/styles.xml'

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.Sudoku" parent="Theme.Material3.DayNight" />
    <style name="CustomDialogTheme" parent="Theme.AppCompat.Dialog">
            <item name="android:windowBackground">@color/background</item>
            <item name="android:textColor">@color/text_color</item>
    </style>
</resources>
```

Explanation

- **parent="Theme.Material3.DayNight":** Sets the base theme to one that includes ActionBar support.
- **Theme.Sudoku:** Defines the primary theme for the Sudoku application.
- **CustomDialogTheme:** Defines a custom theme for dialog windows with specified background and text colors.

Step 2: Define Strings for Menu Items:

We need to define the text labels for the menu items to ensure they are easily configurable and maintainable. Add the necessary string resources in 'strings.xml' to define labels for the menu items.

Code 'res/values/strings.xml'

```
<string name="settings_label">Settings...</string>
<string name="hints_title">Show Hints</string>
<string name="hints_summary">Show Past Results</string>
```

Explanation

- **settings_label:** Defines the label for the Settings menu item.
- **show_hints:** Defines the label for the Show Hints option.
- **show_past_results:** Defines the label for the Show Past Results option

Step 3: Define the Menu in XML:

We need to create a menu resource file that defines the structure and content of the options menu. Create a new XML file 'menu.xml' in the 'res/menu' directory to define the menu items.

Code 'res/menu/menu.xml'

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/settings"
        android:title="@string/settings_label" />
    <item android:id="@+id/show_hints"
        android:title="@string/show_hints" />
    <item android:id="@+id/show_past_results"
        android:title="@string/show_past_results" />
</menu>
```

Explanation

- **menu:** The root element for defining a menu.
- **item:** Defines a single menu item. The android:id attribute uniquely identifies the item, while android:title sets the label.

Step 4: Modify MainActivity to Handle the Menu:

We need to inflate the menu defined in XML and handle menu item selections within the MainActivity. Update 'MainActivity.java' to include the necessary methods to inflate the menu and respond to menu item selections.

1. **Add the necessary imports to 'MainActivity.java':**

Code

```
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
```

Explanation

- **Menu:** Represents the options menu.
- **MenuInflater:** Used to instantiate menu XML files into Menu objects.
- **MenuItem:** Represents a single item in a menu.

2. **Override the 'onCreateOptionsMenu' method to inflate the menu:**

Code

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu, menu);
    return true;
}
```

Explanation

- **onCreateOptionsMenu:** This method is called when the options menu is first created.
- **super.onCreateOptionsMenu(menu):** Calls the superclass implementation.
- **MenuInflater inflater = getMenuInflater():** Gets a MenuInflater to instantiate menu XML files.
- **inflater.inflate(R.menu.menu, menu):** Inflates the menu from the menu.xml resource file into the provided Menu object.

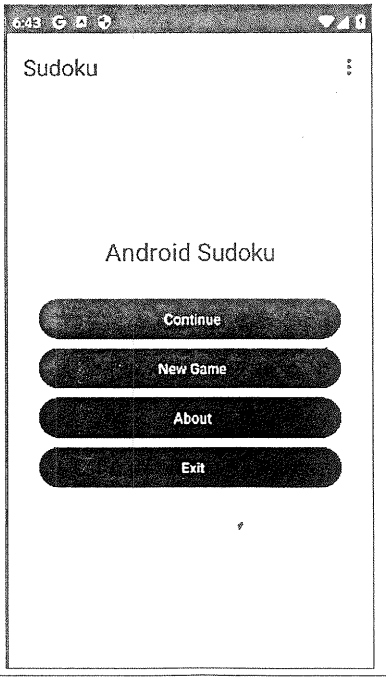
3. **Override the 'onOptionsItemSelected' method to handle menu item selections:**

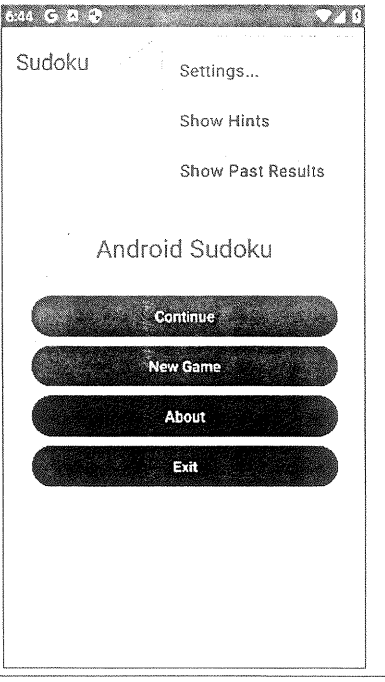
Code

```
@Override
  public boolean onOptionsItemSelected(MenuItem item) {
        //  We will implement the actions in next section
        return false;
    }
```

Output

When the Menu button is pressed or the overflow menu icon is tapped, a menu with options for Settings, Show Hints, and Show Past Results appears. The selection of these menu items can be further implemented to perform their respective actions. This enhances the user experience by providing easy access to configurable options.





Explanation

- **onOptionsItemSelected:** This method is called whenever an item in the options menu is selected.
- **MenuItem item:** Represents the selected menu item.
- **return false:** Indicates that the menu item selection has not been handled yet. Actions will be implemented in the next section.

6.9 Adding Settings

We have already added the menu with menu items Settings, Show Hints, and Show Past Results. Now, by clicking on Settings, let's create preferences. Preferences allow users to configure settings that control the behavior and appearance of the application.

Understanding Preferences

Preferences are used to save and retrieve key-value pairs of primitive data types. They provide a simple way for apps to store user preferences such as settings and retrieve them when needed. Preferences are essential for maintaining user-specific configurations to improve user experience by allowing personalization and to ensure that settings persist across app sessions.

How Preferences Work

- Preferences are stored in a SharedPreferences file, which is essentially a key-value pair storage system.
- They can be accessed and modified using methods provided by the SharedPreferences class.
- Preferences can be defined in an XML file, which Android uses to generate the necessary UI components to display and manage the settings.

Step-by-Step Implementation of Preferences

Step 1: Define the Preferences Activity:

We need to create a new activity to handle user preferences for users to configure settings for the app. Let us create a new Java class 'Prefs.java' to manage the preferences.

Code 'src/main/java/com/example/sudoku/Prefs.java'

```
package com.example.sudoku;

import android.os.Bundle;
import androidx.preference.PreferenceFragmentCompat;
import androidx.appcompat.app.AppCompatActivity;

public class Prefs extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getSupportFragmentManager().beginTransaction()
```

```
                .replace(android.R.id.content, new PrefsFragment())
                .commit();
    }

    public static class PrefsFragment extends PreferenceFragmentCompat {
        @Override
        public void onCreatePreferences(Bundle savedInstanceState,
                                        String rootKey) {
            setPreferencesFromResource(R.xml.preferences, rootKey);
        }
    }
}
```

Explanation

- **Prefs Class:** Extends 'AppCompatActivity' to provide a container for the preferences fragment.
- **onCreate:** Sets up the fragment transaction to display the 'PrefsFragment'.
- **PrefsFragment Class:** Extends 'PreferenceFragmentCompat' to display the preferences defined in the 'preferences.xml' file.
- **onCreatePreferences:** Loads the preferences from the XML resource.

Step 2: Define the Preferences in XML:

We need to define the preferences options available for users to configure. Create a new XML file 'preferences.xml' to define the preferences.

Code

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <ListPreference
        android:key="difficulty_level"
        android:title="Difficulty Level"
        android:summary="Select the difficulty level of the game"
        android:entries="@array/difficulty_levels"
        android:entryValues="@array/difficulty_values"
        android:defaultValue="medium" />
    <SwitchPreferenceCompat
        android:key="download_data"
        android:title="Download Data"
        android:summary="Enable or disable downloading additional data"
        android:defaultValue="false" />
</PreferenceScreen>
```

Explanation

- **PreferenceScreen:** The root element for defining a screen of preferences.
- **ListPreference:** Represents a preference with a list of entries to choose from.
- **android:key:** Unique identifier for the preference.

- **android:title:** Title displayed for the preference.
- **android:summary:** Summary text providing additional information about the preference.
- **android:entries:** The displayed names for the list items.
- **android:entryValues:** The values corresponding to each entry.
- **android:defaultValue:** Default value for the preference.
- **SwitchPreferenceCompat:** Represents a preference with a switch that can be toggled on and off.
- **android:key:** Unique identifier for the preference.
- **android:title:** Title displayed for the preference.
- **android:summary:** Summary text providing additional information about the preference.
- **android:defaultValue:** Default value for the preference.

Step 3: Define Preference Entries in Arrays XML:

We need to provide the entries and values for the ListPreference. Create a new XML file 'arrays.xml' to define the entries and values for the difficulty level preference.

Code 'res/values/arrays.xml'

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="difficulty_levels">
        <item>Easy</item>
        <item>Medium</item>
        <item>Hard</item>
    </string-array>
    <string-array name="difficulty_values">
        <item>easy</item>
        <item>medium</item>
        <item>hard</item>
    </string-array>
</resources>
```

Explanation

- **string-array:** Defines an array of strings.
- **name:** Unique identifier for the array.
- **item:** Represents each item in the array.

Step 4: Update MainActivity:

To connect the settings menu item to the preferences activity, the 'onOptionsItemSelected' method in 'MainActivity' needs to be updated.

Code

```
@Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.settings) {
```

```
            startActivity(new Intent(this, Prefs.class));
            return true;
        }
        // More items can be added here if needed
        return false;
    }
```

Step 5: Register the Preferences Activity in the Manifest:

We need to ensure that the preferences activity is registered in the AndroidManifest.xml so it can be launched. Add the preferences activity to the 'AndroidManifest.xml' file.

Code 'AndroidManifest.xml'

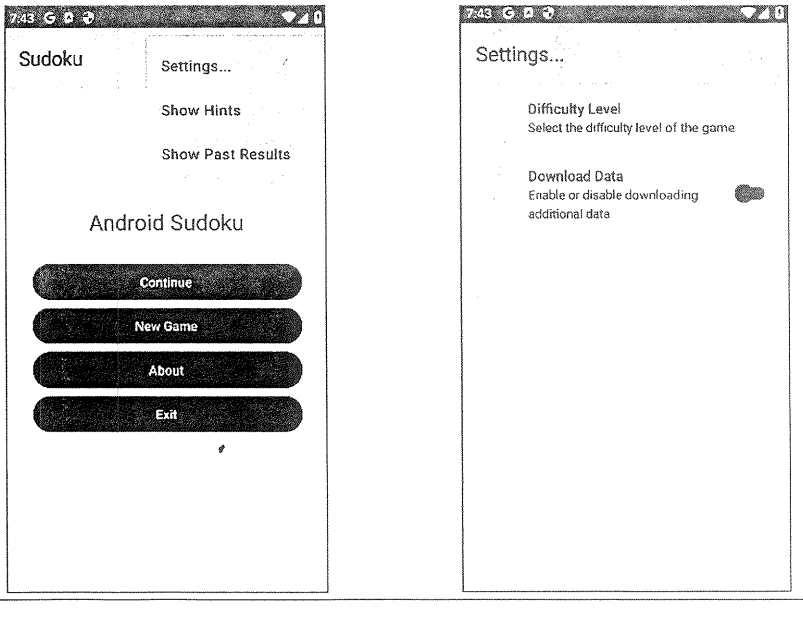
```
<activity android:name=".Prefs"
    android:label="@string/settings_title" />
```

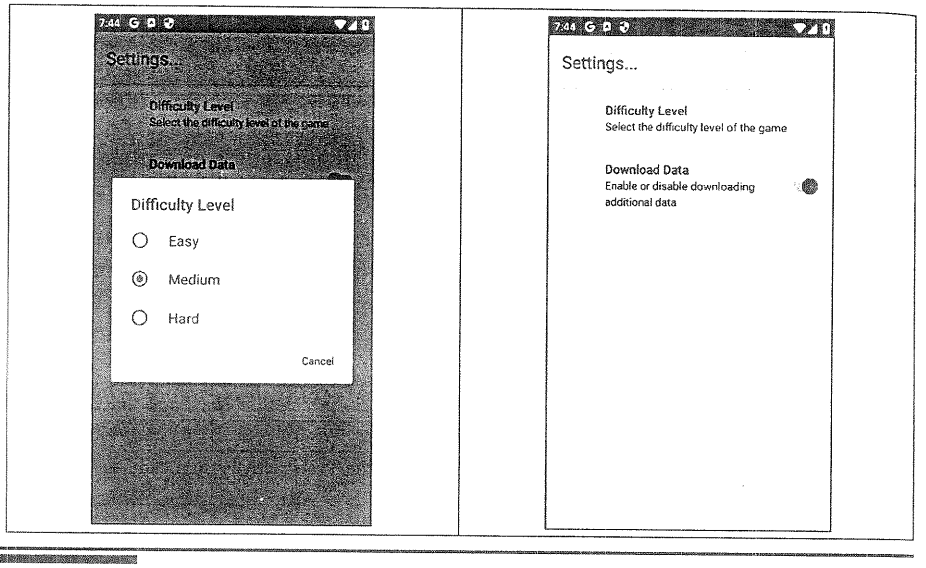
Explanation

- **<activity android:name=".Prefs":** Registers the 'Prefs' activity so it can be launched from the app.
- **android:label:** Sets the label for the activity.
- **android:theme:** Specifies the theme for the activity (if needed).

Output

When the Menu button is pressed, a menu with a Settings option appears. Selecting the Settings option opens a preferences screen where the user can toggle settings such as "Download Data" and select the difficulty level. This enhances the user experience by providing easy access to configurable options.





6.10 Debugging

Debugging is the process of identifying, analyzing, and fixing bugs or issues in software. It is an essential part of the software development lifecycle to ensure that applications run smoothly and meet the desired functionality and performance standards. Android provides various tools and techniques to facilitate the debugging process and ensure the quality and performance of their apps.

Importance of Debugging

1. **Error Identification:** Debugging helps in identifying errors or bugs in the code, which can then be fixed to ensure the application functions as intended.
2. **Performance Optimization:** The inefficiencies in the code can be identified and optimized to improve the performance of the application.
3. **Code Quality Improvement:** Regular debugging ensures that the codebase remains clean, maintainable, and free of bugs, leading to higher quality software.
4. **User Experience Enhancement:** Debugging ensures that the application runs smoothly to provide a better and more reliable experience for users.
5. **Preventing Future Issues:** By thoroughly debugging the application, potential issues can be identified and resolved before they cause problems. It reduces the likelihood of bugs in future updates.

Approaches to Debugging in Android

In Android development, there are two primary approaches to debugging applications:

1. Debugging with Log Messages
2. Debugging with the Debugger

6.10.1 Debugging with Log Messages

Log messages play a crucial role in tracking the flow of execution and diagnosing issues in Android applications. The Android Log class offers several static methods that developers can utilize to print messages of different severity levels to the system log. By incorporating log messages strategically throughout the codebase, developers can gain insights into the application's behavior, identify potential issues, and facilitate the debugging process effectively.

In addition to printing messages of various severity levels, the Android Log class provides developers with the flexibility to customize log messages based on the context and importance of the information being logged. By using different log levels such as Error, Warning, Information, Debugging, and Verbose, developers can categorize and prioritize log messages to focus on critical areas of the codebase or specific functionalities that require attention.

The log messages can also include relevant contextual information, variable values, method calls, and timestamps to provide a comprehensive view of the application's runtime behavior. Developers can leverage log messages not only for debugging purposes but also for performance monitoring, tracking user interactions, and analyzing system events to enhance the overall quality and reliability of the Android application.

Log Levels and Methods

The following table summarizes the different log levels, their methods, and detailed descriptions with example usage:

Method	Description	Example Usage
Log.e	Used to log error messages, indicating serious issues that may cause the application to malfunction.	Log.e(TAG, "Error occurred while processing the request")
Log.w	Used to log warning messages, indicating potential issues that should be looked into.	Log.w(TAG, "Potential issue detected in the configuration")
Log.i	Used to log informational messages, providing useful information about the application's normal operations.	Log.i(TAG, "Application started successfully")
Log.d	Used to log debugging messages, which are useful during development to understand the flow and state of the application.	Log.d(TAG, "Variable x value is: " + x)
Log.v	Used to log verbose messages, providing detailed information for diagnostic purposes.	Log.v(TAG, "Entering method calculateResults()")
Log.wtf	Used to log critical failure messages, indicating serious errors that should never happen.	Log.wtf(TAG, "Unexpected condition occurred!")

Example Code with Log Messages

Code MainActivity.java

```
package com.example.sudoku;

import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import androidx.appcompat.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "SudokuApp";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.i(TAG, "MainActivity created");  // Informational message

        Button continueButton = findViewById(R.id.continue_button);
        Button newGameButton = findViewById(R.id.new_game_button);
        Button aboutButton = findViewById(R.id.about_button);
        Button exitButton = findViewById(R.id.exit_button);

        continueButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d(TAG, "Continue button clicked");
                // Handle continue button click
            }
        });

        newGameButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d(TAG, "New Game button clicked");
                // Handle new game button click
            }
        });
```

```
        aboutButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d(TAG, "About button clicked");
                Intent intent = new Intent(MainActivity.this, About.class);
                startActivity(intent);
            }
        });

        exitButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d(TAG, "Exit button clicked");
                finish();
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.menu, menu);
        Log.i(TAG, "Options menu created");  // Informational message
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.settings) {
            Log.d(TAG, "Settings menu item selected");
            startActivity(new Intent(this, Prefs.class));
            return true;
        }
        Log.w(TAG, "Unhandled menu item selected: " + id);  // Warning message
        // More items can be added here if needed
        return false;
    }
}
```

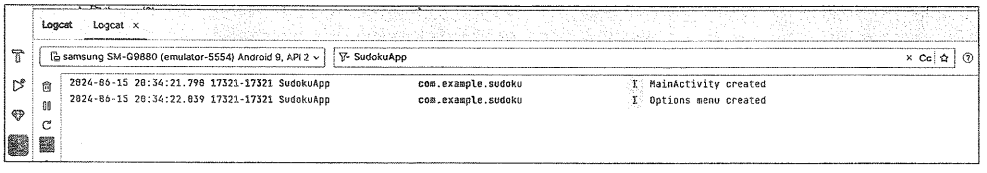
Explanation

- The log messages added in the 'MainActivity.java' code help track key events and actions. When the 'MainActivity' is created, an informational message is logged with 'Log.i(TAG, "MainActivity created")'.
- For button clicks, debugging messages such as 'Log.d(TAG, "Continue button clicked")', 'Log.d(TAG, "New Game button clicked")', 'Log.d(TAG, "About button clicked")', and 'Log.d(TAG, "Exit button clicked")' are logged to track user interactions with the continue, new game, about, and exit buttons, respectively.
- When the options menu is created, another informational message is logged with 'Log.i(TAG, "Options menu created")'.
- Finally, menu item selections are tracked with 'Log.d(TAG, "Settings menu item selected")' for the settings item, and a warning message 'Log.w(TAG, "Unhandled menu item selected: " + id)' is logged for any unhandled menu items.
- The 'TAG' is a constant string used in log messages to identify the source of the log message. It is particularly useful in filtering the logs based on the tag name. In the example code, 'TAG' is set to '"SudokuApp"'. This allows developers to filter and view only the log messages related to the Sudoku app in Logcat or when using 'adb logcat'. This filtering capability makes it easier to track and diagnose issues specific to the application without being overwhelmed by logs from other parts of the system.

Viewing Log Messages

Using Logcat in Android Studio:

1. **Open Logcat:** In Android Studio, open the Logcat window by selecting 'View > Tool Windows > Logcat'.
2. **Filter by Tag:** Use the filter option to search for specific tags or severity levels. We can filter by the TAG used in log messages, such as "SudokuApp".
3. **Run the App:** Run the application to see the log messages in real-time.



6.10.2 Debugging with Debugger

Debugging with a debugger is an essential approach for identifying and resolving issues in Android applications. A debugger allows developers to pause program execution, inspect variables, step through code line-by-line, and monitor the application's state to understand the root cause of any issues. This method is particularly useful for diagnosing complex problems that are not easily traceable with log messages.

Importance of Debugging with a Debugger

+ **Real-time Analysis:** Provides the ability to inspect the current state of an application, including variable values, memory usage, and thread states.

+ **Step-by-Step Execution:** Allows developers to execute code line-by-line, making it easier to identify the exact point where an error occurs.
+ **Conditional Breakpoints:** Enables setting conditions for breakpoints to pause execution only when certain conditions are met to reduce the noise of unnecessary breaks.
+ **Watch Variables:** Helps in tracking the value changes of specific variables during execution.
+ **Call Stack Inspection:** Provides insight into the sequence of method calls leading to a particular point in the code. It helps in understanding the flow and dependencies.

Setting Up the Debugger in Android Studio

1. **Enable Debuggable Mode:**

   Ensure the application is in debuggable mode. This can be done by setting the 'android:debuggable' attribute to 'true' in the 'AndroidManifest.xml' file during development.

Code AndroidManifest.xml

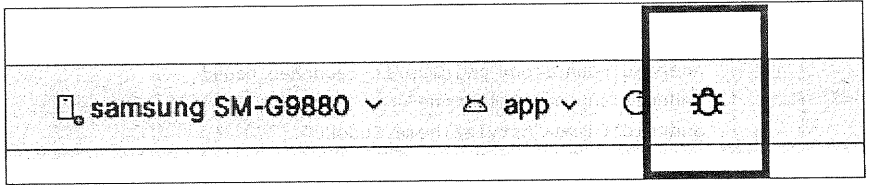
```
   <?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:debuggable="true"
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.Sudoku"
        tools:targetApi="31"
        tools:ignore="HardcodedDebugMode">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".About"
            android:label="@string/about_title"
            android:theme="@android:style/Theme.Dialog">
```

```
        </activity>
        <activity androic:name=".Prefs"
            android:label="@string/settings_label" />

    </application>

</manifest>
```

2. **Set Breakpoints:**
   - Open the Java file in Android Studio.
   - Click in the left margin next to the line number where we want to set a breakpoint. A red dot will appear, indicating the breakpoint.

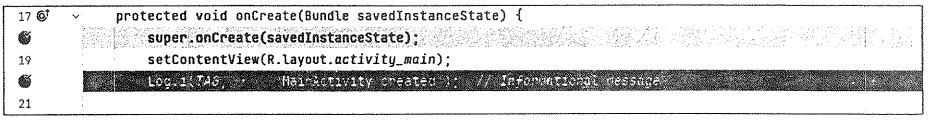
```
16      @Override
17      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
19          setContentView(R.layout.activity_main);
            Log.i(TAG, msg: "MainActivity created"); // Informational message
21
```

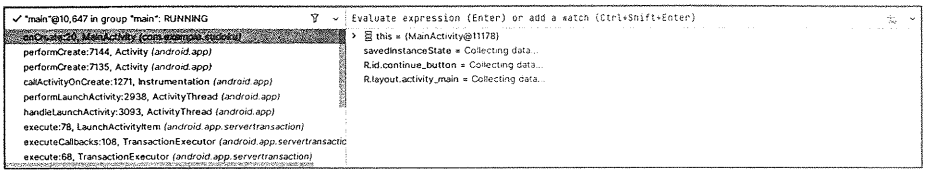
3. **Attach the Debugger:**
   - Click on the 'Debug' button (green bug icon) in the toolbar or right-click the project and select 'Debug 'app''.
   - Android Studio will build the project and deploy the app to the connected device or emulator in debug mode.



4. **Debugging Controls:**
   - **Resume Program (F9):** Continues execution until the next breakpoint or the end of the program.
   - **Pause Program:** Pauses the execution at the current line.
   - **Step Over (F8):** Executes the current line and moves to the next line.
   - **Step Into (F7):** If the current line contains a method call, steps into that method to debug its code.
   - **Step Out (Shift+F8):** Steps out of the current method and returns to the caller method.
   - **Evaluate Expression (Alt+F8):** Allows evaluation of expressions and variables at the current execution point.

5. **Inspect Variables:**
   - When the program is paused at a breakpoint, the 'Variables' pane in Android Studio shows the current values of all variables in the current scope.
   - Hovering over a variable in the editor window also shows its current value.





6. **Call Stack:**

   The 'Call Stack' pane shows the sequence of method calls that led to the current execution point. This helps in understanding the flow of execution and dependencies.

7. **Watch Expressions:**

   Add specific variables or expressions to the 'Watches' pane to monitor their values throughout the execution of the program.

6.11 Review Questions

Section - A

1. What is procedural design in the context of UI design?
2. What is declarative design, and how is it typically implemented in Android development?
3. Why does Google recommend using declarative XML for designing Android user interfaces?
4. Describe the advantages of declarative design over procedural design.
5. What does "designing by declaration" refer to in Android development?
6. What is the purpose of an About Box in an app?
7. What are the two kinds of menus supported in Android?
8. What are the different approaches of debugging an Android App?
9. What is an Importance of Debugging with a Debugger?

Section - B

1. Explain the key characteristics and components of an opening screen for an app.
2. What are the UI requirements for the Sudoku opening screen?

3. Outline the step-by-step process for creating the Sudoku opening screen, including project setup and defining the layout in XML.
4. Why is it important to use alternate resources in Android development?
5. How does a ScrollView help in handling different screen sizes in Android?
6. Describe how to create a different layout specifically for landscape mode in Android.
7. Outline the steps to implement an About Box in the Sudoku app, including defining a new activity and creating a layout file.
8. How is the About button wired up in the main activity to display the About Box?
9. Explain the importance and benefits of applying a theme in Android development.
10. Describe the process of applying a default theme to an activity in Android.
11. How can a custom theme be created and applied to an activity in Android?
12. Outline the steps to add an options menu in the Sudoku app, including changing the theme and defining the menu in XML.
13. How is the 'onOptionsItemSelected' method used to handle menu item selections in Android?
14. Explain the purpose of preferences in an Android app.
15. Outline the step-by-step process for defining a preferences activity in the Sudoku app.
16. How are preference entries defined in arrays XML and used in a ListPreference?
17. Discuss the importance of debugging in Android development.
18. Mention the differences between a Style and Theme

Section - C

1. Explain how log messages can be used for debugging in Android. Provide examples of different log levels.
2. Describe the process of using the debugger in Android Studio.
3. Discuss the importance of UI design in mobile application development, citing specific examples from the Sudoku app project.
4. Provide a step-by-step guide on how to create the opening screen for the Sudoku app.
5. Explain the concept of alternate resources in Android and how they can be used to create a responsive UI, using examples from the Sudoku app.
6. Provide a detailed explanation on the implementation of an About Box in the Sudoku app.
7. Discuss the process and significance of creating a settings menu in the Sudoku app. Include the steps for defining preferences in XML, setting up the preferences activity, and handling user interactions.

UNIT - IV

CHAPTER 7

SMS MESSAGING, EMAIL AND LOCATION BASED SERVICES

CONTENTS

- SMS Messaging
  - Basic Steps to Send an SMS in Android:
  - Step-by-Step Implementation of Sending SMS
- Sending Email
  - Email Related Classes and Methods in Android
  - Basic Steps to Send an Email in Android
  - Step-by-Step Implementation of Sending an Email
- Location-Based Services (LBS)
  - Displaying Maps
  - Getting Location Data
  - Monitoring a Location
- Review Questions

7.1 SMS Messaging

SMS (Short Message Service) is a text messaging service that allows mobile devices to exchange short text messages. It is a widely used communication method in mobile networks and is supported by most telephone, Internet, and mobile device systems. In Android, developers can implement SMS messaging functionality using the **SmsManager** class provided by the Android SDK.

Key Points on SMS Messaging in Android

1. **SmsManager Class:**
   - The 'SmsManager' class in Android is responsible for managing SMS operations, including sending various types of SMS messages such as data, text, and PDU (Protocol Data Unit) messages.
   - Developers can utilize methods provided by 'SmsManager' to send SMS messages programmatically from their applications.
2. **Permissions:**
   - To send SMS messages from an Android application, the app must declare the 'SEND_SMS' permission in the 'AndroidManifest.xml' file.
   - The 'SEND_SMS' permission is considered a dangerous permission and requires explicit user consent at runtime on Android devices running API level 23 (Android 6.0) and above.
   - Additionally, to monitor incoming SMS messages, the 'RECEIVE_SMS' permission must be declared in the manifest file. This permission allows the application to receive and process incoming SMS messages.
3. **BroadcastReceiver:**

   While not mandatory for sending SMS messages, a 'BroadcastReceiver' can be used to listen for incoming SMS messages.

   By registering a 'BroadcastReceiver' to intercept the 'SMS_RECEIVED' broadcast action, the application can process incoming SMS messages as needed.
4. **Handling SMS Delivery Reports:**
   - Developers can set delivery intents when sending SMS messages using 'SmsManager' to receive delivery reports indicating the status of message delivery.
   - This feature allows applications to track the delivery status of SMS messages and take appropriate actions based on the delivery report.

SMS Related Classes and Methods in Android

The below classes and methods are used to implement various SMS-related functionalities in the Android applications including sending SMS messages, receiving and processing incoming SMS messages, querying SMS content providers, and interacting with SMS message data.

1. **SmsManager Class:**

   Description: Manages SMS operations such as sending text messages.

Methods:

- **'getDefault()':** Obtains the default 'SmsManager' instance.
- **'sendTextMessage()':** Sends a text message to a specified destination.
- **'sendDataMessage()':** Sends a data-based SMS message.
- **'divideMessage()':** Divides a text message into multiple parts if it exceeds the SMS character limit.
- **'sendMultipartTextMessage()':** Sends a multi-part text message.

2. **Telephony.Sms Class:**

   Description: Provides access to SMS related information and operations.

   Methods:

   - **'getDefault()':** Obtains the default 'Sms' instance.
   - **'getMessagesFromIntent()':** Retrieves SMS messages from an intent.

3. **SmsMessage Class:**

   Description: Represents an SMS message.

   Methods:

   - **'createFromPdu()':** Creates an 'SmsMessage' object from a Protocol Data Unit (PDU).
   - **'getDisplayMessageBody()':** Retrieves the message body.
   - **'getDisplayOriginatingAddress()':** Retrieves the sender's address.

4. **BroadcastReceiver for SMS:**

   Description: Listens for incoming SMS messages.

   Methods:

   - **'onReceive()':** Handles incoming SMS messages received by the broadcast receiver.

7.1.1 Basic Steps to Send an SMS in Android

The below steps need to be follwed to send SMS messages.

1. **Declare Necessary Permissions:**
   - Ensure that the 'SEND_SMS' permission is declared in the 'AndroidManifest.xml' file to send SMS messages.
   - Additionally, declare the 'RECEIVE_SMS' permission if the application needs to receive and process incoming SMS messages.
2. **Design a Simple User Interface (UI):**
   - Create a user interface that allows users to input the phone number and message they want to send via SMS.
   - Include appropriate input fields and a button to trigger the SMS sending process.
3. **Implement Runtime Permission Checks:**
   - Check for and request the 'SEND_SMS' permission at runtime on devices running Android 6.0 (API level 23) and above.

- Handle permission responses to ensure that the application has the necessary permissions to send SMS messages.

4. **Use SmsManager to Send the SMS:**
   - Obtain an instance of 'SmsManager' using 'SmsManager.getDefault()'.
   - Use 'SmsManager' methods like 'sendTextMessage()' to send the SMS with the specified phone number and message content.
   - Handle any exceptions that may occur during the SMS sending process.
5. **Optional: Handle SMS Delivery Reports:**
   - If needed, set up delivery intents when sending SMS messages to receive delivery reports indicating the status of message delivery.
   - Process the delivery reports to track the status of SMS message delivery.

7.1.2 Step-by-Step Implementation of Sending SMS

Step 1: Project Setup:

Create an Android project by selecting "Empty Views Activity", naming the project "SMSExample", and choosing Java as the language.

Step 2: Declare Permissions:

Declare the necessary permissions to send SMS in the AndroidManifest.xml file. This step ensures the application has the required permissions to perform SMS operations. Include <use-permission> tag outside the <application> tag.

Code AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.smsexample">

    <uses-permission android:name="android.permission.SEND_SMS"/>
    <application
        -------------
        -------------
    </application>
</manifest>
```

Explanation

- The 'SEND_SMS' permission is necessary to allow the app to send SMS messages.

Step 3: Create the UI:

Design the UI to input the recipient's phone number and the message content. This UI will allow users to enter the necessary details to send an SMS.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <EditText
        android:id="@+id/phone_number"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Phone Number"
        android:inputType="phone" />

    <EditText
        android:id="@+id/message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:id="@+id/send_sms"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Send SMS" />
</LinearLayout>
```

Explanation

- The 'LinearLayout' provides a vertical layout for the UI components.
- The 'EditText' fields allow the user to input the recipient's phone number and the message content.
- The 'Button' triggers the SMS sending functionality when clicked.

Step 4: Implement MainActivity:

Implement the activity to handle sending SMS with runtime permission checks. This activity manages the user interactions and handles the SMS sending process.

Code MainActivity.java

```
package com.example.smsexample;

import android.Manifest;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.telephony.SmsManager;
```

```
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

public class MainActivity extends AppCompatActivity {

    private static final int PERMISSION_REQUEST_CODE = 1;
    private EditText phoneNumberEditText;
    private EditText messageEditText;
    private Button sendSmsButton;

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

        phoneNumberEditText = findViewById(R.id.phone_number);
        messageEditText = findViewById(R.id.message);
        sendSmsButton = findViewById(R.id.send_sms);

        sendSmsButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if (ContextCompat.checkSelfPermission(MainActivity.this,
                    Manifest.permission.SEND_SMS)!= PackageManager.PERMISSION_GRANTED) {
                     ActivityCompat.requestPermissions(MainActivity.this, new String[]
                       {Manifest.permission.SEND_SMS}, PERMISSION_REQUEST_CODE);
                } else {
                    sendSms();
                }
            }
        });
    }

    private void sendSms() {
        String phoneNumber = phoneNumberEditText.getText().toString();
        String message = messageEditText.getText().toString();
```

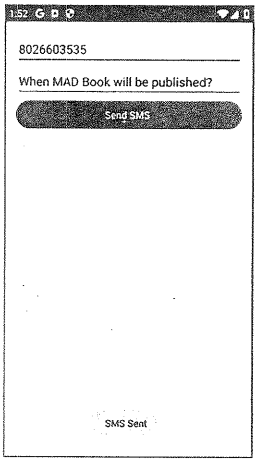
```
        try {
            SmsManager smsManager = SmsManager.getDefault();
            smsManager.sendTextMessage(phoneNumber, null, message, null, null);
            Toast.makeText(this, "SMS Sent", Toast.LENGTH_SHORT).show();
        } catch (Exception e) {
            Toast.makeText(this, "SMS Failed to Send", Toast.LENGTH_SHORT).show();
            e.printStackTrace();
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[]
                                    permissions, @NonNull int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (requestCode == PERMISSION_REQUEST_CODE) {
            if (grantResults.length > 0 && grantResults[0]
                    == PackageManager.PERMISSION_GRANTED) {
                sendSms();
            } else {
                Toast.makeText(this, "Permission Denied", Toast.LENGTH_SHORT).show();
            }
        }
    }
}
```

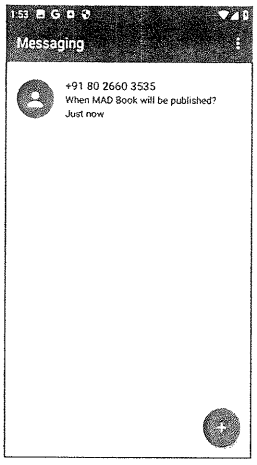
Explanation

1. **onCreate:**
   - Initializes the UI components.
   - Sets up the 'sendSmsButton' click listener to check for permissions and send the SMS if permissions are granted.
2. **sendSms:**
   - Retrieves the phone number and message content from the input fields.
   - Uses 'SmsManager' to send the SMS.
   - Displays a toast message indicating whether the SMS was sent successfully or failed.
3. **onRequestPermissionsResult:**
   - Handles the result of the permission request. If granted, it proceeds to send the SMS; otherwise, it shows a permission denied message.

Output

The SMSExample app allows users to input a phone number and a message, and then send the SMS by clicking the "Send SMS" button. Once the SMS is sent, a toast message confirms the successful sending. The message can then be viewed on the recipient's mobile device,





Note : This app has been tested with both a physical device and using the LDPlayer9 emulator.

How to Test the SMS Sending Functionality in LDPlayer9

External emulators like LDPlayer9 offer more comprehensive testing capabilities, including the ability to simulate SMS functionalities. By configuring a virtual mobile number (e.g., '8026603535') in LDPlayer9, developers can input this same number in the SMSExample app's phone number field and send an SMS. The delivery of the SMS can then be verified by opening the Messaging App within LDPlayer9, where the sent message should be displayed. This method effectively bridges the gap between basic emulation and real-world testing, making it a valuable tool for thorough functional testing.

Steps to Test the SMS Sending Functionality in LDPlayer9

1. **Configure LDPlayer9:**
   - Open LDPlayer9.
   - Go to the Settings in LDPlayer9.
   - Configure a virtual mobile number (e.g., '8026603535').
2. **Install a Messaging App:**
   - Go to the Play Store in LDPlayer9.
   - Download and install a messaging app (if not already installed).
3. **Run the SMSExample App:**
   - Open Android Studio.
   - Connect to LDPlayer9 as the running device.
   - Run the SMSExample app by selecting LDPlayer9 from the device dropdown menu in Android Studio.
4. **Send an SMS:**
   - Open the SMSExample app in LDPlayer9.
   - Enter the virtual mobile number '8026603535' and a test message in the app.
   - Press the "Send SMS" button.

5. **Verify the SMS Delivery:**
   - Open the Messaging App within LDPlayer9.
   - Verify that the sent message is displayed in the messaging app.

By following these steps, the SMS sending functionality can be tested effectively using the LDPlayer9 emulator, ensuring that the app works as expected in a simulated real-world environment.

7.2 Sending Email

Email messaging is a vital communication tool in Android applications to send and receive messages over the internet. Developers commonly implement email sending functionality using **Intents** to open the user's preferred email client with pre-filled details. To enhance this feature, developers can incorporate advanced capabilities such as attachment support, inline images, and email templates. By integrating with email APIs and services like JavaMail API, developers can customize the email sending process to meet specific requirements.

Key Points on Email Messaging in Android

1. **Utilizing Intents for Email Sending:**
   - Intents with the **'ACTION_SEND'** action are commonly used to initiate the email sending process in Android applications.
   - This approach allows developers to seamlessly open the user's chosen email client, providing a familiar environment for composing emails.
2. **Pre-filling Email Details:**
   - By setting appropriate data and extras in the Intent, developers can pre-fill essential email components such as recipient email addresses, subject lines, and message bodies.
   - Pre-populating these details enhances user convenience and simplifies the email composition experience.
3. **Enhancing User Experience:**
   - Customizing Intents with pre-filled email details improves user interaction by reducing manual input and streamlining the sending process.
   - A user-friendly approach to email sending fosters efficient communication and enhances the overall usability of the application.
4. **Promoting Efficiency and Engagement:**
   - The use of Intents for email sending not only enhances functionality but also encourages users to engage more actively within the app.
   - By providing a seamless email sending experience, developers can promote user engagement and facilitate effective communication channels.
5. **Integration with Advanced Features:**
   - Developers can further enhance email sending capabilities by integrating advanced features like attachment support, inline images, and email templates.
   - Exploring additional functionalities enriches the email sending experience and offers users more customization options.

7.2 1 Email Related Classes and Methods in Android

These classes and methods are used to implement email functionalities in Android applications.

- **Intent Class:**

  Description: Used to perform an action, such as sending an email.

  Methods:

  - **'setAction(String action)':** Sets the action to be performed, such as 'Intent.ACTION_SEND'.
  - **'putExtra(String name, String value)':** Adds extra data to the intent, such as email recipient, subject, and body.

7.2.2 Basic Steps to Send an Email in Android

1. **Design a Simple User Interface (UI):**
   - Create a user interface that allows users to input the recipient's email address, subject, and message body.
   - Include appropriate input fields and a button to trigger the email sending process.
2. **Use Intent to Open Email Client:**
   - Create an intent with the action 'Intent.ACTION_SEND'.
   - Add extras to the intent, such as email recipient, subject, and message body.
   - Start the activity with the intent to open the user's preferred email client.

7.2.3 Step-by-Step Implementation of Sending an Email

Step 1: Project Setup

Create an Android project by selecting "Empty Views Activity", naming the project "EmailExample", and choosing Java as the language.

Step 2: Create the UI

Design the UI to input the recipient's email address, subject, and message body. This UI will allow users to enter the necessary details to send an email.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <EditText
        android:id="@+id/email_address"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:hint="Email Address"
        android:inputType="textEmailAddress" />

    <EditText
        android:id="@+id/email_subject"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Subject"
        android:inputType="text" />

    <EditText
        android:id="@+id/email_message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Message"
        android:inputType="textMultiLine" />

    <Button
        android:id="@+id/send_email"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Send Email" />
</LinearLayout>
```

Explanation

- The 'LinearLayout' provides a vertical layout for the UI components.
- The 'EditText' fields allow the user to input the recipient's email address, the email subject, and the message content.
- The 'Button' triggers the email sending functionality when clicked.

Step 3: Implement MainActivity

Implement the activity to handle sending email using intents. This activity manages the user interactions and handles the email sending process.

Code MainActivity.java

```
package com.example.emailexample;

import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
```

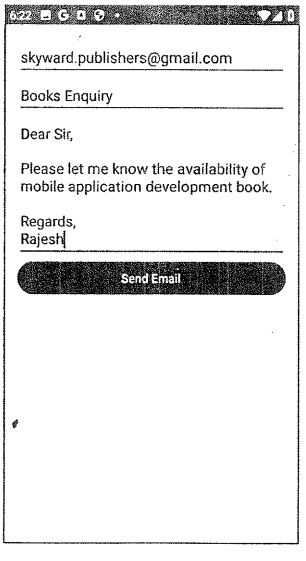
```
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private EditText emailAddressEditText;
    private EditText emailSubjectEditText;
    private EditText emailMessageEditText;
    private Button sendEmailButton;

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

        emailAddressEditText = findViewById(R.id.email_address);
        emailSubjectEditText = findViewById(R.id.email_subject);
        emailMessageEditText = findViewById(R.id.email_message);
        sendEmailButton = findViewById(R.id.send_email);

        sendEmailButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                sendEmail();
            }
        });
    }

    private void sendEmail() {
        String emailAddress = emailAddressEditText.getText().toString();
        String subject = emailSubjectEditText.getText().toString();
        String message = emailMessageEditText.getText().toString();

        Intent emailIntent = new Intent(Intent.ACTION_SEND);
        emailIntent.setType("message/rfc822");
        emailIntent.putExtra(Intent.EXTRA_EMAIL, new String[]{emailAddress});
        emailIntent.putExtra(Intent.EXTRA_SUBJECT, subject);
        emailIntent.putExtra(Intent.EXTRA_TEXT, message);

        if (emailIntent.resolveActivity(getPackageManager()) != null) {
            startActivity(emailIntent);
        }else{
            Toast.makeText(this, "Email Client Not Installed",
                           Toast.LENGTH_SHORT).show();
        }
    }
}
```

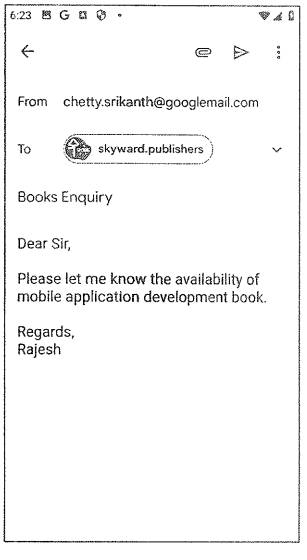
Explanation

1. **onCreate(Bundle savedInstanceState):**
   - Initializes the activity when it is first created.
   - Sets the content view to the layout defined in 'activity_main.xml'.
   - Initializes EditText fields for email address, subject, and message, along with the send button.
   - Sets a click listener on the send button to call the 'sendEmail()' method when clicked.
2. **sendEmail() Method:**
   - Retrieves the email address, subject, and message entered by the user from the respective EditText fields.
   - Creates an Intent with the action 'Intent.ACTION_SEND' to send an email.
   - Sets the type of the Intent to '"message/rfc822"' to indicate an email message.
3. **Adds extras to the Intent:**
   - 'Intent.EXTRA_EMAIL': Specifies the recipient email address.
   - 'Intent.EXTRA_SUBJECT': Specifies the email subject.
   - 'Intent.EXTRA_TEXT': Specifies the email message body.
   - Checks if there is an activity that can handle the email Intent using 'resolveActivity()'.
   - If an email client is available, starts the activity with the email Intent to allow the user to send the email.
   - If no email client is available, displays a toast message indicating that an email client is not installed.

Output

The output demonstrates the successful implementation of the email sending functionality in the EmailExample app. The first output shows the app's UI, where the user has entered the recipient's email address, subject, and message content. After clicking the "Send Email" button, the second output shows the Gmail app opened with the recipient's email address, subject, and message body correctly populated. This confirms that the app correctly captures and passes the email details to the email client.





Note : This app has been tested with both a Physical Device and using the LDPlayer9 emulator.

How to Test the Email Sending Functionality in LDPlayer9

1. **Install Gmail Client:**
   - Open LDPlayer9.
   - Go to the Play Store in LDPlayer9.
   - Download and install the Gmail client.
2. **Set Up Gmail:**
   - Open the Gmail app in LDPlayer9.
   - Set up a Google account in the Gmail app.
   - Verify that Gmail opens up and the account is set up correctly.
3. **Run the EmailExample App:**
   - Open Android Studio.
   - Connect to LDPlayer9 as the running device.
   - Run the EmailExample app by selecting LDPlayer9 from the device dropdown menu in Android Studio.
   - Enter the email details in the app and click "Send Email".
   - Verify that the Gmail client opens with the pre-filled email details.

By following these steps, the email sending functionality can be tested effectively using the LDPlayer9 emulator.

7.3 Location-Based Services (LBS)

In today's digital era, the incorporation of geographic location into diverse technologies has led to the emergence of Location-Based Services (LBS). These services utilize the geographic position of devices to offer personalized and contextually appropriate information to users. LBS have become essential in everyday life, ranging from navigation applications assisting in city navigation to social media platforms that label locations in posts,

The LBS offer enhanced user experiences by tailoring content based on the user's current or specified location by enabling features like mapping, navigation, local search, geotagging, and location-based notifications. LBS play a crucial role in creating location-aware applications that offer convenience, efficiency, and value to users in various domains such as social networking, travel, retail, emergency services, and more.

Location-Based Services (LBS) in mobile applications utilize the device's location data to offer personalized and context-aware services to users. By leveraging location information, apps can deliver relevant content, navigation assistance, social interactions, and more based on the user's current or specified location.

What are Location Based Services?

Location-Based Services (LBS) refer to mobile applications or services that utilize the geographical location of a user to provide personalized and contextually relevant information, features, or functionality. LBS leverage technologies such as GPS, Wi-Fi, cellular networks, and geospatial data to deliver location-specific services to users on their mobile devices.

Key Components of LBS

1. **Positioning Technology:** The underlying technology used to determine the device's location. This can include GPS, Wi-Fi, and cellular network positioning.
2. **Geographic Information Systems (GIS):** Systems that capture, store, manipulate, analyze, manage, and present spatial or geographic data.
3. **Location Data:** The geographic coordinates (latitude and longitude) obtained from the positioning technology.
4. **LBS Applications:** Software applications that utilize location data to provide various services to users.

Key Features of Location-Based Services:

+ **Map Integration:** Map integration embeds interactive maps within applications to allow users to visualize their current location and navigate seamlessly. Google Maps is commonly used for its rich features and customization options.
+ **Real-Time Location Tracking:** Real-time location tracking continuously updates the user's position to ensure accurate and dynamic location information.
+ **Geocoding and Reverse Geocoding:** Geocoding converts addresses into geographic coordinates, while reverse geocoding converts coordinates back into addresses to enhance location searches and contextual information retrieval.
+ **Location-Based Notifications:** Location-based notifications use geofencing and proximity alerts to send messages based on the user's location. It is useful for marketing, security, and event-driven applications.
+ **Route Planning and Navigation:** Route planning and navigation provide turn-by-turn directions and route optimization, helping users reach destinations efficiently by considering real-time traffic data.
+ **Location Sharing:** Location sharing allows users to broadcast their real-time location to friends and family to enhance social networking and safety applications.
+ **Points of Interest (POI):** POI features help users find nearby amenities and provide detailed information such as reviews and ratings to enhance the map's utility with valuable context.
+ **Location-Based Advertising:** Location-based advertising targets users with relevant ads based on their location to increase the effectiveness of promotions and offers.
+ **Emergency Services:** Emergency services enable users to send their location to emergency contacts or services to ensure quick assistance in distress situations.
+ **Fitness and Health Tracking:** Fitness and health tracking apps use location data to monitor physical activities, map routes, and analyze performance metrics for health and fitness goals.
+ **Asset and Vehicle Tracking:** Asset and vehicle tracking provides real-time monitoring of movable objects. It is essential for logistics, supply chain management, and security applications.

Important Applications of Location-Based Services (LBS)

1. **Navigation and Maps:** LBS power navigation apps like Google Maps, Waze, and Apple Maps to provide real-time directions, traffic updates, and location-based services.
2. **Location-Based Advertising:** Targeted advertising based on user location, delivering relevant ads for nearby businesses and services.
3. **Geotagging:** Social media platforms use LBS for geotagging posts and photos, allowing users to share their location with friends and followers.
4. **Location-Based Gaming:** Games like Pokémon GO and geocaching use LBS to create interactive experiences based on the player's physical location.
5. **Emergency Services:** LBS assist emergency responders in locating individuals in distress and providing accurate directions to the scene.
6. **Retail and Marketing:** Retailers use LBS for targeted promotions, personalized offers, and location-based notifications to attract customers in proximity to their stores.
7. **Fleet Management:** LBS are used in logistics and transportation industries for tracking vehicles, optimizing routes, and improving fleet efficiency.
8. **Healthcare:** LBS enable healthcare providers to track patients, locate medical facilities, and send alerts in case of emergencies.
9. **Tourism and Travel:** Travel apps utilize LBS to recommend nearby attractions, provide location-specific information, and offer personalized travel itineraries.
10. **Field Service Management:** LBS help businesses in managing field operations, dispatching technicians based on proximity to service locations, and optimizing service routes.

Examples of Location-Based Services (LBS)

- **Google Maps:**
  - **Features:** Real-time navigation, route planning, traffic updates, and detailed maps.
  - **Use Case:** Helping users find directions, explore new areas, and avoid traffic. Widely used for navigation in urban and rural areas.
- **Facebook:**
  - **Features:** Location-based check-ins, friend-finding, and real-time location sharing.
  - **Use Case:** Enhancing social interactions by allowing users to share their location, check-in at various places, and find friends nearby. Popular among users in India for social networking and event planning.
- **Zomato:**
  - **Features:** Discovering nearby restaurants, user reviews and ratings, food delivery.
  - **Use Case:** Assisting users in finding the best local restaurants and ordering food based on their current location. Extremely popular for exploring dining options and delivery services.

- **Uber:**
  - **Features:** Connecting passengers with nearby drivers, real-time ride tracking, accurate pick-up and drop-off locations.
  - **Use Case:** Providing efficient and convenient transportation services. Widely used for ride-hailing, offering various services including UberGo, UberX, and UberMoto.
- **Google AdWords:**
  - **Features:** Delivering targeted ads and promotions based on user location.
  - **Use Case:** Increasing marketing effectiveness by reaching users with relevant ads at the right place and time. Utilized by businesses to drive local traffic and increase sales through location-specific advertising.

Technical Aspects of LBS

1. **Location Detection Methods:**
   - **GPS (Global Positioning System):** Provides high accuracy outdoors by receiving signals from satellites.
   - **Cell Tower Triangulation:** Estimates location based on the distance from nearby cell towers, useful in urban areas.
   - **Wi-Fi Positioning:** Determines location based on the proximity to Wi-Fi networks, useful indoors.
2. **APIs and Frameworks:**
   - **Google Maps API:** Allows developers to integrate maps and location services into their applications.
   - **Android LocationManager:** Provides access to system location services to determine the device's location.
3. **Privacy and Security:**
   - Ensuring user consent and control over location data.
   - Implementing data encryption and anonymization to protect user privacy.

7.3.1 Displaying Maps

Integrating Google Maps into an Android application significantly enhances its functionality by leveraging real-time location and mapping features. This capability is crucial for applications that rely on navigation, geolocation services, or visual representation of geographic data.

When an Android application displays maps using Google Maps integration, it allows users to interact with dynamic maps that provide detailed geographical information. Users can view their current location, search for specific places, get directions for navigation, and explore points of interest. The visual representation of maps not only enhances the user experience but also adds a layer of context to the information presented within the application.

Key Points to Remember when Integrating Google Maps in an Android Application:

1. **API Key:** Obtaining an API key is essential for utilizing Google Maps services within the application. This key acts as a unique identifier and authorization token for accessing the mapping functionalities.
2. **Permissions:** Configuring the necessary permissions in the 'AndroidManifest.xml' file is crucial to enable the application to access location data and connect to the internet. Permissions like 'ACCESS_FINE_LOCATION' and 'INTERNET' are commonly required for Google Maps integration.
3. **Google Maps SDK:** Integrating the Google Maps SDK into the project dependencies is a fundamental step to leverage the full range of features and capabilities offered by Google Maps. The SDK provides access to mapping tools, location services, and interactive functionalities within the application.

Importance of the API Key

The API key is a critical component when accessing Google's API services including Google Maps. Some key points about the importance of the API key are listed below.

1. **Identification:** The API key uniquely identifies the application to allow Google to attribute requests to a specific project, which helps in tracking usage and billing.
2. **Access Control:** The API key ensures that only authorized requests are processed by controlling access to various services and functionalities.
3. **Security:** By preventing unauthorized use of an application's resources, the API key adds a layer of security. It can restrict access based on criteria like IP addresses, HTTP referrers, or mobile apps.
4. **Usage Monitoring:** Google uses the API key to monitor the number of requests made by the application. It helps to track usage and ensure it stays within set limits.
5. **Billing:** For paid services, the API key is essential for billing purposes. It enable Google to charge for services based on the requests made using the key.
6. **Customization and Control:** The API key allows for configuring quotas, managing usage, and accessing advanced features of Google's APIs by providing better customization and control over the services used by the application.

How to Obtain an API Key?

1. **Go to the Google Cloud Console:**
   - Open [Google Cloud Console](https://console.cloud.google.com/).
   - Sign in with your Google account.
2. **Create a New Project:**
   - Click on the project drop-down and select "New Project".
   - Enter the project name (e.g., 'MyMapsApp').
   - Click "Create".

3. **Enable Maps SDK for Android:**
   - In the Google Cloud Console, navigate to 'API & Services' > 'Library'.
   - Search for "Maps SDK for Android".
   - Click "Enable".
4. **Obtain the API Key:**
   - Go to 'API & Services' > 'Credentials'.
   - Click on "Create credentials" and select "API key".
   - An API key will be generated. Copy this key for later use.

Step-by-Step Implementation of Displaying Maps

Step 1: Project Setup:

Create an Android project by selecting "Empty Views Activity", naming the project "MyMapsApp", and choosing Java as the language.

Step 2: Obtaining and Adding the API Key:

Obtain the API key from Google Cloud Console and add it to the project.

Code AnroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <uses-permission android:name="android.permission.INTERNET"/>
   <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
 <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MyMapsApp"
        tools:targetApi="31">

        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

```
        <meta-data
            android:name="com.google.android.geo.API_KEY"
            android:value="YOUR_API_KEY_HERE"/>
    </application>
</manifest>
```

Explanation

This file includes the necessary permissions for internet access and location services. The API key is added as meta-data.

- The '<uses-permission android:name="android.permission.INTERNET"/>' tag grants the application permission to access the internet, which is required for fetching map data and other online resources.
- The '<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>' and '<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>' tags allow the application to access the device's precise and approximate location, respectively. These permissions are necessary for location-based functionalities in the app.
- The API key obtained from Google Cloud Console should replace 'YOUR_API_KEY_HERE' in the '<meta-data android:name="com.google.android.geo.API_KEY" android:value="YOUR_API_KEY_HERE"/>' tag. This key enables the application to use Google Maps services.

Step 3: Adding Google Maps Dependencies:

Add necessary dependencies to the project to use Google Maps.

Code Add dependencies to 'build.gradle.kts (Module: app)

```
dependencies {
    implementation(libs.play.services.maps)
}
```

Explanation

This dependency is necessary to include Google Maps SDK in the project.

Step 4: Creating the Layout for the Map

Design the layout to display Google Maps.

Code 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"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
```

```
    <com.google.android.gms.maps.MapView
        android:id="@+id/mapView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Explanation

1. The layout uses 'ConstraintLayout' as the root element to provide a flexible layout system that allows for positioning and sizing of child views in a more efficient and accurate way.
2. Inside the 'ConstraintLayout', a 'MapView' from the Google Maps SDK for Android is added. This view is used to display the Google Map within the application.

   MapView:

   - 'MapView' is a part of the Google Maps SDK for Android and is used to display a map within an Android application.
   - The 'android:id="@+id/mapView"' attribute assigns a unique ID to the 'MapView', which can be referenced in the code.
   - The 'android:layout_width="match_parent"' and 'android:layout_height="match_parent"' attributes make the 'MapView' fill the entire screen, ensuring the map takes up all available space within the 'ConstraintLayout'.
   - 'MapView' provides various lifecycle methods that need to be managed in the activity or fragment to ensure proper operation. These methods include 'onCreate()', 'onResume()', 'onPause()', 'onDestroy()', and 'onSaveInstanceState()', which correspond to the activity's lifecycle methods.

This layout sets up the basic structure needed to display Google Maps in an Android application. The next steps would involve initializing the 'MapView' in the 'MainActivity' and managing its lifecycle methods.

Step 5: Initializing the Map in MainActivity:

Set up and initialize the 'MapView' in the main activity.

Code MainActivity.java

```
package com.example.mymapsapp;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;
import com.google.android.gms.maps.MapView;
import com.google.android.gms.maps.MapsInitializer;
import com.google.android.gms.maps.OnMapReadyCallback;
import com.google.android.gms.maps.GoogleMap;

public class MainActivity extends AppCompatActivity {
    private MapView mapView;
    private GoogleMap googleMap;
```

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

    mapView = findViewById(R.id.mapView);
    mapView.onCreate(savedInstanceState);
    mapView.onResume();

    try {
        MapsInitializer.initialize(this);
    } catch (Exception e) {
        e.printStackTrace();
    }

    mapView.getMapAsync(new OnMapReadyCallback() {
        @Override
        public void onMapReady(GoogleMap mMap) {
            googleMap = mMap;
            googleMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
            googleMap.getUiSettings().setZoomControlsEnabled(true);
        }
    });
}

@Override
protected void onResume() {
    super.onResume();
    mapView.onResume();
}

@Override
protected void onPause() {
    super.onPause();
    mapView.onPause();
}

@Override
protected void onDestroy() {
    super.onDestroy();
    mapView.onDestroy();
}
```

```
    @Override
    public void onLowMemory() {
        super.onLowMemory();
        mapView.onLowMemory();
    }
}
```

Explanation

1. **Instance Variables:**

   'MapView' and 'GoogleMap' objects are declared to handle the map display and interaction. 'MapView' is used to display the map, while 'GoogleMap' provides the map functionality.

2. **'onCreate' Method:**
   - 'super.onCreate(savedInstanceState)' initializes the activity.
   - 'setContentView(R.layout.activity_main)' sets the layout to 'activity_main.xml', which includes the 'MapView'.
   - 'mapView' is initialized using 'findViewById(R.id.mapView)', and its lifecycle methods 'onCreate' and 'onResume' are called to ensure the map is displayed correctly.
   - 'MapsInitializer.initialize(this)' is called to initialize the Google Maps SDK.
   - 'mapView.getMapAsync' sets an 'OnMapReadyCallback' to be notified when the map is ready to use.

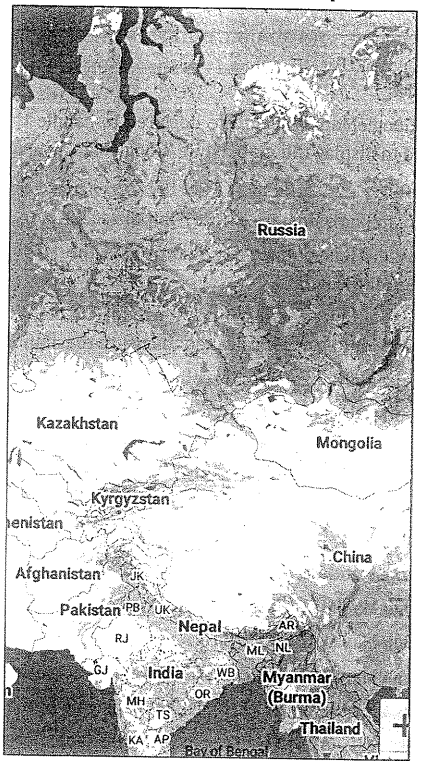
3. **'OnMapReadyCallback' Implementation:**
   - 'googleMap' is assigned the map instance when the map is ready.
   - 'googleMap.setMapType(GoogleMap.MAP_TYPE_NORMAL)' sets the map type to normal, displaying the standard map view. Other possible map types include:
     - GoogleMap.MAP_TYPE_SATELLITE for a satellite view.
     - GoogleMap.MAP_TYPE_TERRAIN for a terrain view.
     - GoogleMap.MAP_TYPE_HYBRID for a combination of satellite and normal views.
     - GoogleMap.MAP_TYPE_NONE to display no base map tiles.
   - 'googleMap.getUiSettings().setZoomControlsEnabled(true)' enables zoom controls on the map, allowing users to zoom in and out.

4. **Lifecycle Methods:**
   - 'onResume', 'onPause', 'onDestroy', and 'onLowMemory' methods are overridden to manage the 'mapView' lifecycle in sync with the activity lifecycle, ensuring proper resource management.
     - 'onResume()': Calls 'mapView.onResume()' to resume the map.
     - 'onPause()': Calls 'mapView.onPause()' to pause the map.
     - 'onDestroy()': Calls 'mapView.onDestroy()' to destroy the map.
     - 'onLowMemory()': Calls 'mapView.onLowMemory()' to handle low memory situations.

This setup initializes and displays a Google Map within the application, handles map lifecycle events, sets the map type to normal, and enables UI settings like zoom controls, compass, and My Location button.

Output

When the application is run, the 'MapView' should display a Google Map. Users can interact with the map, zoom in and out, and see their current location if location permissions are granted. The map type is set to normal, and basic UI controls like zoom and compass are enabled.



Changing Views

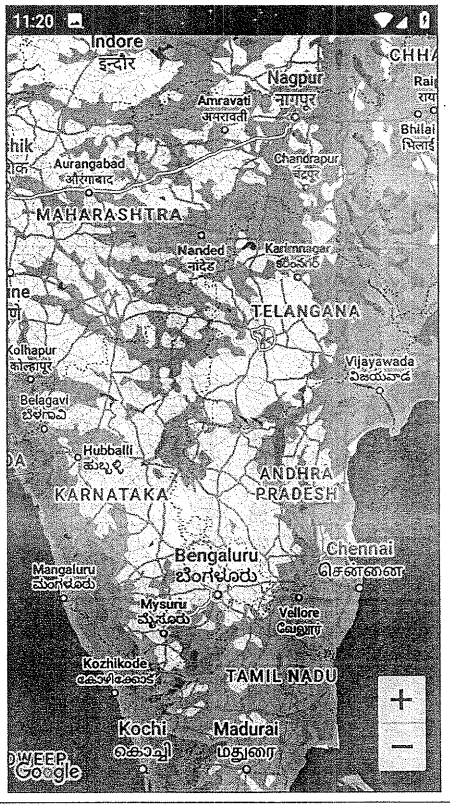
Google Maps provides various view types that can be switched to offer different perspectives and levels of detail. Each of these map types provides a unique way of viewing and interacting with the map, catering to different needs and enhancing user experience based on the specific requirements of the application.

1. **Normal View:**
    - **Description:** This is the standard roadmap view, which displays streets, road names, and geographical features. It is ideal for everyday navigation and understanding the layout of roads and neighborhoods.
    - **Use Case:** Use this view for applications that require clear street-level details, such as driving directions or locating addresses.

Code

```
googleMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
```

Output



2. **Satellite View:**
   - **Description:** This view displays satellite images of the area, showing detailed and realistic images of the terrain and buildings.
   - **Use Case:** Useful for applications that need to show actual imagery of locations, such as real estate apps, outdoor activity planners, or any app where users benefit from seeing real-world visuals.

Code

```
googleMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
```

Output

3. **Terrain View:**
   - **Description:** This view emphasizes terrain information, showing topography and physical landscape features such as mountains, valleys, and bodies of water.
   - **Use Case:** Ideal for outdoor enthusiasts, hikers, and geographers who need to understand the physical layout of the land. This view helps in planning outdoor activities like hiking, mountain biking, or camping.

Code

```
googleMap.setMapType(GoogleMap.MAP_TYPE_TERRAIN);
```

Output

4. **Hybrid View:**
   - **Description:** Combines the normal roadmap view with satellite images, providing street names and geographical features overlaid on real-world imagery.
   - **Use Case:** Best for applications where both detailed real-world imagery and street information are important. This view can be used in tourism apps, urban planning tools, or navigation systems that benefit from both satellite and roadmap details.

Code

```
googleMap.setMapType(GoogleMap.MAP_TYPE_HYBRID);
```

Output

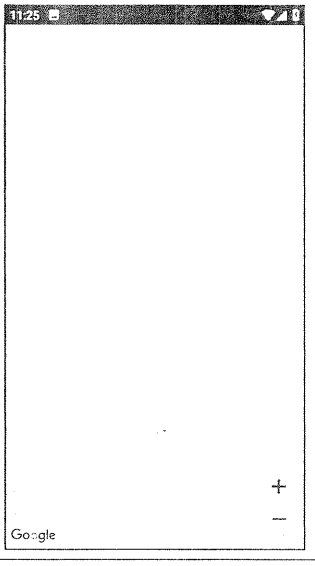
5. None:

- **Description:** Displays no base map tiles, allowing developers to overlay custom data on a blank canvas.
- **Use Case:** Useful for custom mapping applications where developers want full control over what is displayed. This could include specialized mapping solutions for indoor maps, custom geographical analyses, or any scenario where standard map tiles are unnecessary or unwanted.

Code

```
googleMap.setMapType(GoogleMap.MAP_TYPE_NONE);
```

Output



Adding a Button to Toggle Between Views

To provide an option for toggling between different map views, a button can be added to the layout and its OnClickListener updated to change the map type. This involves initializing the button in the onCreate method and setting up a click listener that cycles through the available map types (Normal, Satellite, Terrain, Hybrid) when the button is pressed. This allows for a dynamic and interactive user experience, enabling the user to easily switch the map view according to their preference.

Navigating to a Specific Location in Google Maps

By default, Google Maps displays the map of the United States when it is first loaded. However, it is possible to set Google Maps to display a specific location programmatically. To navigate to a particular location, the 'animateCamera()' method of the 'GoogleMap' class can be used. This method allows for animating the map's camera to a specific location with a smooth transition.

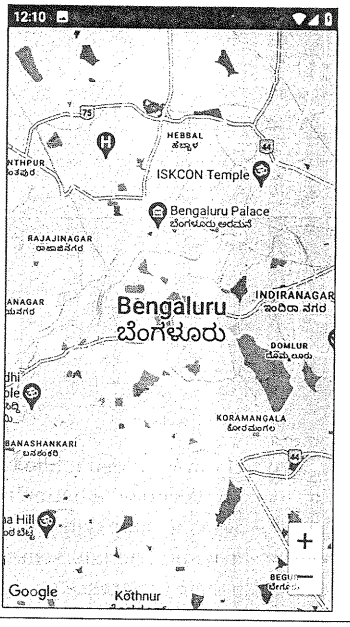
To navigate to a specific location, such as Bangalore, India, the following code snippet should be added inside the 'onMapReady()' method:

Code

```
mapView.getMapAsync(new OnMapReadyCallback() {
    @Override
    public void onMapReady(GoogleMap mMap) {
        googleMap = mMap;
        googleMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
        googleMap.getUiSettings().setZoomControlsEnabled(true);
```

```
        // Animate camera to Bangalore
        LatLng bangalore = new LatLng(12.9716, 77.5946);
        googleMap.animateCamera(CameraUpdateFactory.newLatLngZoom(bangalore, 12));
    }
});
```

Output



Explanation

- 'LatLng bangalore = new LatLng(12.9716, 77.5946);': This line creates a 'LatLng' object representing the coordinates of Bangalore.
- 'googleMap.animateCamera(CameraUpdateFactory.newLatLngZoom(bangalore, 12));': This line animates the camera to the specified location ('bangalore') with a zoom level of 12. The 'animateCamera()' method provides a smooth transition to the new location.
- **Include the necessary imports:**
  - import com.google.android.gms.maps.CameraUpdateFactory;
  - import com.google.android.gms.maps.model.LatLng;

Adding Markers in Google Maps

Markers can be added to a Google Map to indicate points of interest, such as landmarks or user-specified locations. This can be done programmatically using the 'addMarker()' method of the 'GoogleMap' class.

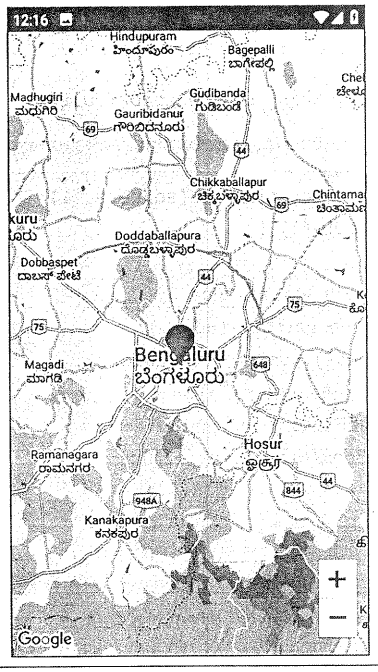
To add a marker at a specific location, such as Bangalore, India, the following code snippet should be added inside the 'onMapReady()' method:

Code

```
mapView.getMapAsync(new OnMapReadyCallback() {
    @Override
    public void onMapReady(GoogleMap mMap) {
        googleMap = mMap;
        googleMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
        googleMap.getUiSettings().setZoomControlsEnabled(true);
```

```
        // Add a marker in Bangalore
        LatLng bangalore = new LatLng(12.9716, 77.5946);
        googleMap.addMarker(new MarkerOptions().position(bangalore)
                                               .title("Marker in Bangalore"));
    }
});
```

Output



Explanation

- 'LatLng bangalore = new LatLng(12.9716, 77.5946);': This line creates a 'LatLng' object representing the coordinates of Bangalore.
- 'googleMap.addMarker(new MarkerOptions(). position(bangalore).title("Marker in Bangalore"));': This line adds a marker at the specified location ('bangalore') with a title "Marker in Bangalore". The 'MarkerOptions' object allows setting the position and additional properties of the marker.
- **Include the necessary imports:**
  - import com.google.android.gms.maps.model. MarkerOptions;
  - import com.google.android.gms.maps.model. LatLng;

Geocoding and Reverse Geocoding

By default, Google Maps displays the map of the United States when it is first loaded. However, it is possible to set Google Maps to display a specific location programmatically. Additionally, geocoding can be used to convert an address into geographic coordinates to place markers on a map, and reverse geocoding can convert geographic coordinates into a human-readable address.

- **Geocoding:** The process of converting an address or place name into geographic coordinates (latitude and longitude), which can then be used to place markers on a map.
- **Reverse Geocoding:** The process of converting geographic coordinates into a human-readable address, such as a street address or place name.

To add geocoding and reverse geocoding functionality inside the 'onMapReady()' method, the following code snippet should be used:

Code

```
mapView.getMapAsync(new OnMapReadyCallback() {
    @Override
    public void onMapReady(GoogleMap mMap) {
        googleMap = mMap;
        googleMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
        googleMap.getUiSettings().setZoomControlsEnabled(true);

        // Geocoding an address to coordinates
        Geocoder geocoder = new Geocoder(MainActivity.this, Locale.getDefault());
        String address = "Bangalore, India";
        try {
            List<Address> addresses = geocoder.getFromLocationName(address, 1);
            if (addresses != null && !addresses.isEmpty()) {
                Address location = addresses.get(0);
                LatLng bangalore = new LatLng(location.getLatitude(),
                                              location.getLongitude());
                googleMap.addMarker(new MarkerOptions().position(bangalore).
                                    title("Marker in Bangalore"));
                googleMap.animateCamera(CameraUpdateFactory
                                        .newLatLngZoom(bangalore, 12));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Reverse geocoding coordinates to an address
        googleMap.setOnMapClickListener(new GoogleMap.OnMapClickListener() {
            @Override
            public void onMapClick(LatLng latLng) {
                try {
                  List<Address> addresses = geocoder.getFromLocation(latLng.latitude,
                                                                     latLng.longitude, 1);
                    if (addresses != null && !addresses.isEmpty()) {
                        Address address = addresses.get(0);
                        String addressText = address.getAddressLine(0);
                        googleMap.addMarker(new MarkerOptions()
                                .position(latLng).title(addressText));
                    }
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });
    }
});
```

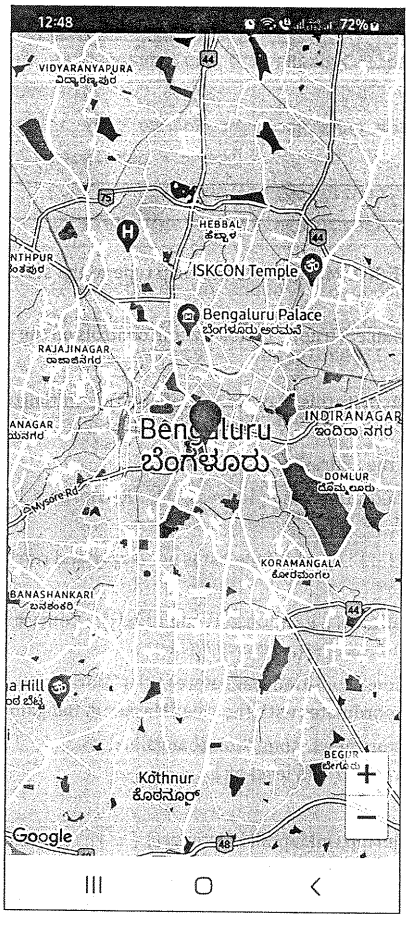
Explanation

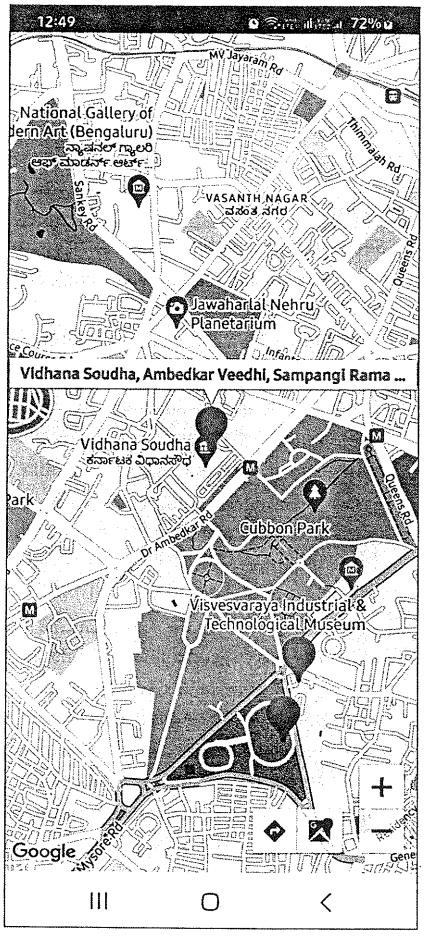
1. **Setting Up the Map:**
   - 'googleMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);': Sets the map type to the standard roadmap view.
   - 'googleMap.getUiSettings().setZoomControlsEnabled(true);': Enables zoom controls on the map.
2. **Geocoding an Address to Coordinates:**
   - 'Geocoder geocoder = new Geocoder(MainActivity.this, Locale.getDefault());': Initializes a 'Geocoder' instance with the default locale.
   - 'String address = "Bangalore, India";': Specifies the address to be geocoded.
   - 'List<Address> addresses = geocoder.getFromLocationName(address, 1);': Retrieves a list of addresses matching the given location name.
   - 'if (addresses != null && !addresses.isEmpty()) { ... }': Checks if the list is not empty and retrieves the first result.
   - 'LatLng bangalore = new LatLng(location.getLatitude(), location.getLongitude());': Converts the address to geographic coordinates.
   - 'googleMap.addMarker(new MarkerOptions().position(bangalore).title("Marker in Bangalore"));': Adds a marker at the specified coordinates with the title "Marker in Bangalore".
   - 'googleMap.animateCamera(CameraUpdateFactory.newLatLngZoom(bangalore, 12));': Animates the camera to the specified location with a zoom level of 12.
3. **Reverse Geocoding Coordinates to an Address:**
   - 'googleMap.setOnMapClickListener(new GoogleMap.OnMapClickListener() { ... });': Sets a map click listener to perform reverse geocoding when the map is clicked.
   - 'List<Address> addresses = geocoder.getFromLocation(latLng.latitude, latLng.longitude, 1);': Retrieves a list of addresses for the given coordinates.
   - 'if (addresses != null && !addresses.isEmpty()) { ... }': Checks if the list is not empty and retrieves the first result.
   - 'String addressText = address.getAddressLine(0);': Converts the coordinates to a human-readable address.
   - 'googleMap.addMarker(new MarkerOptions().position(latLng).title(addressText));': Adds a marker at the specified coordinates with the retrieved address as the title.

Output

When the map opens, it should automatically zoom into Bangalore, India, displaying a marker titled "Marker in Bangalore." Clicking on different locations on the map will trigger reverse geocoding, adding a new marker at the clicked location with a title showing the address of that location.

Note: Geocoder sometimes do not work in emulators and hence this code has been tested with Physical device.





1. **First Image (initial zoom to Bangalore):** Shows the map centered on Bangalore with a single marker. This confirms that the app correctly initializes by zooming into Bangalore and adding the initial marker.
2. **Second Image (with multiple markers around Vidhana Soudha):** Shows multiple markers added at different clicked locations.

   Each marker's title shows the address of the clicked location, confirming that reverse geocoding is working as expected.

7.3.2 Getting Location Data

Mobile devices today are equipped with multiple technologies to determine their location. These technologies enable various applications, such as navigation, location-based services, and geotagging, to function accurately and efficiently.

Different Technologies to Determine Device Location

The primary technologies used to determine a device's location include GPS, cell tower triangulation, and Wi-Fi triangulation. Each of these technologies has its strengths and weaknesses.

- **GPS (Global Positioning System):**
  - **Functionality:** Utilizes satellites orbiting the Earth to pinpoint the device's location.
  - **Advantages:** Highly accurate, especially in outdoor settings with a clear sky.
  - **Disadvantages:** Requires an unobstructed line of sight to satellites, rendering it ineffective indoors or in areas where satellite signals are blocked (e.g., tunnels, dense urban regions).
- **Cell Tower Triangulation:**
  - **Functionality:** Estimates the device's location by identifying nearby cell towers. When a mobile phone is active, it continually communicates with surrounding base stations (cell towers).
  - **Advantages:** Effective indoors and doesn't rely on satellite signals. Particularly useful in densely populated areas with many cell towers.
  - **Disadvantages:** Less accurate than GPS, as accuracy depends on the density and overlapping coverage of cell towers.
- **Wi-Fi Triangulation:**
  - **Functionality:** Connects to a Wi-Fi network and cross-references the service provider's information with databases to determine the location served by the provider.
  - **Advantages:** Effective indoors and usable when cell tower signals are unavailable.
  - **Disadvantages:** Least accurate of the three methods. Accuracy depends on the availability and proximity of Wi-Fi networks.

LocationManager in Android

On Android devices, the LocationManager class is used to determine the user's physical location. It allows the device to access different location providers such as GPS, network (cell tower and Wi-Fi), and passive providers to get the best available location data.

By utilizing the LocationManager class and its interaction with different location providers, Android applications can offer location-based services tailored to various scenarios and environments. Developers can choose the most suitable location provider based on factors like accuracy requirements, availability of satellite signals, and indoor or outdoor usage to provide users with accurate and relevant location information.

Step-by-Step Implementation of Getting Location Data Using LocationManager

Step 1: Project Setup:

Create an Android project by selecting "Empty Views Activity", naming the project "MyLocationApp", and choosing Java as the language

Step 2: Add Permissions in 'AndroidManifest.xml':

Add the necessary permissions to access the user's location. Obtain the API key from Google Cloud Console and add it to the project.

Code AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MyLocationApp"
        tools:targetApi="31">
        <activity android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <meta-data
            android:name="com.google.android.geo.API_KEY"
            android:value="YOUR_API_KEY_HERE"/>
    </application>
</manifest>
```

Step 3: Update Layout File ('activity_main.xml'):

Update layout file to include a MapView to display the map.

Code 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"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <com.google.android.gms.maps.MapView
        android:id="@+id/mapView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Step 4: MainActivity.java:

Update MainActivity.java to initialize MapView, request location permissions, and display the current location on the map.

Code MainActivity.java

```
package com.example.mylocationapp;

import android.Manifest;
import android.content.pm.PackageManager;
import android.location.Location;
import android.os.Bundle;
import android.widget.Toast;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import com.google.android.gms.maps.CameraUpdateFactory;
import com.google.android.gms.maps.GoogleMap;
import com.google.android.gms.maps.MapView;
import com.google.android.gms.maps.MapsInitializer;
import com.google.android.gms.maps.OnMapReadyCallback;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.MarkerOptions;

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity";
    private static final int REQUEST_LOCATION_PERMISSION = 1;
    private MapView mapView;
    private GoogleMap googleMap;

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

```
        mapView = findViewById(R.id.mapView);
        mapView.onCreate(savedInstanceState);
        mapView.onResume();

        try {
            MapsInitializer.initialize(this);
        } catch (Exception e) {
            e.printStackTrace();
        }

        mapView.getMapAsync(new OnMapReadyCallback() {
            @Override
            public void onMapReady(GoogleMap mMap) {
                googleMap = mMap;
                googleMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
                googleMap.getUiSettings().setZoomControlsEnabled(true);

                requestLocationPermission();
            }
        });
    }

    private void requestLocationPermission() {
        if (ContextCompat.checkSelfPermission(this,
            Manifest.permission.ACCESS_FINE_LOCATION) !=
                                    PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this, new String[]
                                {Manifest.permission.ACCESS_FINE_LOCATION},
                                 REQUEST_LOCATION_PERMISSION);
        } else {
            startLocationUpdates();
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode,
                @NonNull String[] permissions,
                @NonNull int[] grantResults) {
      super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (requestCode == REQUEST_LOCATION_PERMISSION) {
            if (grantResults.length > 0 &&
                 grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                startLocationUpdates();
            } else {
```

```
            Toast.makeText(this, "Location permission required",
                           Toast.LENGTH_SHORT).show();
        }
    }
}

  private void startLocationUpdates() {
    if (ContextCompat.checkSelfPermission(this,
        Manifest.permission.ACCESS_FINE_LOCATION)
                        == PackageManager.PERMISSION_GRANTED) {
      googleMap.setMyLocationEnabled(true);
      googleMap.getUiSettings().setMyLocationButtonEnabled(true);

      googleMap.setOnMyLocationChangeListener(new GoogleMap.
                                        OnMyLocationChangeListener() {
          @Override
          public void onMyLocationChange(Location location) {
              LatLng currentLocation = new LatLng
                 (location.getLatitude(), location.getLongitude());
              googleMap.addMarker(new MarkerOptions().
                 position(currentLocation).title("You are here"));
              googleMap.moveCamera(CameraUpdateFactory.
                 newLatLngZoom(currentLocation, 15));
          }
      });
    }
}
```

Explanation

1. **Permissions Handling:**

   requestLocationPermission(): Checks for location permissions and requests them if not granted.

   onRequestPermissionsResult(): Handles the user's response to the permission request.

2. **Map Initialization:**

   mapView.getMapAsync(new OnMapReadyCallback() { ... }): Initializes the map when ready.

   startLocationUpdates(): Enables the location layer and sets up a listener for location changes.

3. **Displaying Current Location:**

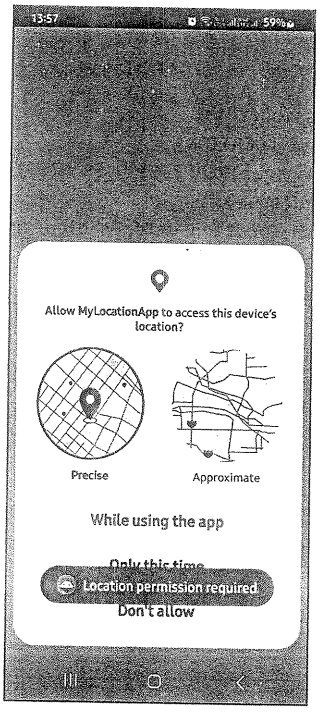
   googleMap.setOnMyLocationChangeListener(): Updates the map with the user's urrent location, adds a marker, and moves the camera to the current location.

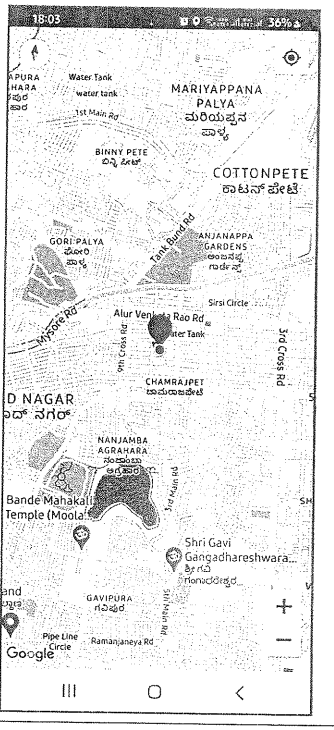
4. **Lifecycle Methods:**

   onResume(), onPause(), onDestroy(), and onLowMemory(): Manage the mapView lifecycle.

Output

When the application runs, it will display a Google Map with the current location indicated by a marker. Users can see their current location on the map and interact with it, including zooming in and out.





Note: This has been tested with a Physical Device.

7.3.3 Monitoring a Location

Monitoring a user's location in Android applications is a powerful feature that allows developers to track the device's position in real-time and trigger actions based on the user's location. The LocationManager class in Android provides methods to monitor location updates and define proximity alerts for specific geographic areas. This functionality is commonly used in applications that require location-based notifications, geofencing, or context-aware features based on the user's proximity to specific locations.

addProximityAlert() Method

To monitor a specific location in Android, developers can use the 'addProximityAlert()' method provided by the 'LocationManager' class. This method sets up a proximity alert for a defined geographic point and triggers a specified action when the device enters or exits the proximity of that location.

The 'addProximityAlert()' method takes the following parameters:

- **Latitude:** The latitude of the monitored location.
- **Longitude:** The longitude of the monitored location.
- **Radius:** The radius in meters around the location within which the alert will be triggered.
- **Expiration:** Time for the proximity alert to be valid, after which it will be deleted (-1 for no expiration).
- **PendingIntent:** The intent to be triggered when the user enters or exits the defined location.

Step by Step Implementation of Location Monitoring

Step 1: Project Setup:

Create an Android project by selecting "Empty Views Activity", naming the project "locationmonitoringdemo", and choosing Java as the language

Step 2: Adding Permissions:

Add the necessary permissions to access the user's location and the internet in the 'AndroidManifest.xml' file.

Code 'AndroidManifest.xml'

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.INTERNET"/>
```

Explanation

- The 'ACCESS_FINE_LOCATION' permission allows the app to access precise location data from GPS.
- The 'ACCESS_COARSE_LOCATION' permission allows the app to access approximate location data from Wi-Fi and cell towers.
- The 'INTERNET' permission is necessary for network operations.

Step 3: UI Code:

Ensure that the layout file contains the necessary UI components. For this example, a simple 'TextView' will display location monitoring status.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:text="Monitoring Location"
        android:textSize="18sp"
        android:layout_centerInParent="true"/>
</RelativeLayout>
```

Explanation

- The 'TextView' with ID 'textView' is used to display a message indicating that location monitoring is active.
- The 'RelativeLayout' is used to center the 'TextView' in the middle of the screen.

Step 4: Activity Code:

Set up and initialize 'LocationManager', request location permissions, and set up the proximity alert in the 'MainActivity.java'.

Code MainActivity.java

```
package com.example.locationmonitoringdemo;

import android.Manifest;
import android.app.AlertDialog;
import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.location.LocationManager;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;
import android.widget.Toast;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity";
    private static final int REQUEST_FINE_LOCATION_PERMISSION = 1;
    private static final String PROXIMITY_ALERT_ACTION
                    = "com.example.locationmonitoringdemo.PROXIMITY_ALERT";
    private LocationManager locationManager;
    private TextView textView;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    textView = findViewById(R.id.textView);
    locationManager = (LocationManager) getSystemService
                                        (Context.LOCATION_SERVICE);

    requestFineLocationPermission();

    // Check if the intent action is the proximity alert action
    Intent intent = getIntent();
    if (PROXIMITY_ALERT_ACTION.equals(intent.getAction())) {
        showAlert();
    }
}

private void requestFineLocationPermission() {
    Log.d(TAG, "Requesting fine location permission");
    if (ContextCompat.checkSelfPermission(this,
          Manifest.permission.ACCESS_FINE_LOCATION)
                != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
              new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
              REQUEST_FINE_LOCATION_PERMISSION);
    } else {
        Log.d(TAG, "Fine location permission already granted");
        addProximityAlert();
    }
}

@Override
public void onRequestPermissionsResult(int requestCode,
          @NonNull String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult
                              (requestCode, permissions, grantResults);
    if (requestCode == REQUEST_FINE_LOCATION_PERMISSION) {
        if (grantResults.length > 0 && grantResults[0] == PackageManager.
                                                    PERMISSION_GRANTED) {
            Log.d(TAG, "Fine location permission granted");
```

```
                addProximityAlert();
            } else {
                Toast.makeText(this, "Location permission required",
                                                    Toast.LENGTH_SHORT).show();
                Log.d(TAG, "Fine location permission denied");
            }
        }
    }

    private void addProximityAlert() {
        if (ContextCompat.checkSelfPermission(this,
              Manifest.permission.ACCESS_FINE_LOCATION) ==
                                            ackageManager.PERMISSION_GRANTED) {
            double latitude = 12.95899; // Replace with the desired latitude
            double longitude = 77.56113; // Replace with the desired longitude
            float radius = 5.0f; // 5 meter radius
            long expiration = -1; // No expiration

            Intent intent = new Intent(this, MainActivity.class);
            intent.setAction(PROXIMITY_ALERT_ACTION);
            PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
                                intent, PendingIntent.FLAG_UPDATE_CURRENT |
                                            PendingIntent.FLAG_MUTABLE);

            try {
                locationManager.addProximityAlert(latitude, longitude, radius,
                                                    expiration, pendingIntent);
                Log.d(TAG, "Proximity alert added");
            } catch (SecurityException e) {
                Log.e(TAG, "SecurityException: " + e.getMessage());
            }

        } else {
            Log.d(TAG, "Fine location permission not granted. ");
        }
    }
```

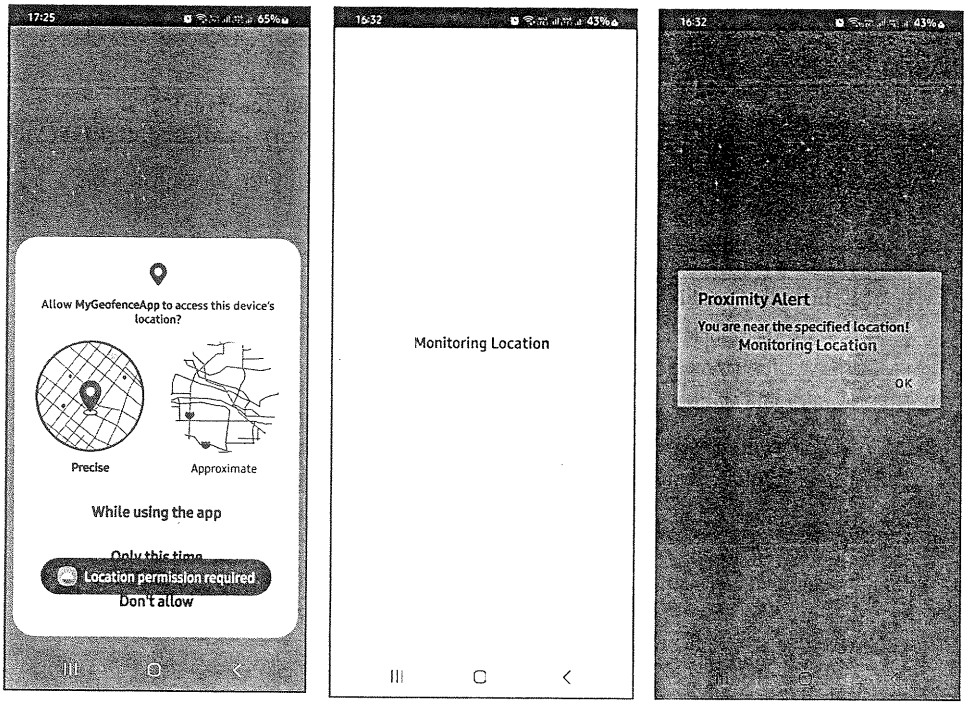
```
    private void showAlert() {
        new AlertDialog.Builder(this)
                .setTitle("Proximity Alert")
                .setMessage("You are near the specified location!")
                .setPositiveButton("OK", (dialog, which) -> dialog.dismiss())
                .show();
    }
}
```

Explanation

1. **Instance Variables:**
   - 'LocationManager' and 'TextView' objects are declared to handle location services and display messages.
2. **onCreate Method:**
   - Initializes the 'LocationManager' and 'TextView'.
   - Calls 'requestFineLocationPermission()' to ensure location permissions are granted.
   - Checks if the activity was launched with the proximity alert action and displays an alert if true.
3. **requestFineLocationPermission Method:**
   - Checks if 'ACCESS_FINE_LOCATION' permission is granted.
   - If not, requests the permission from the user.
   - If granted, calls 'addProximityAlert()' to set up the proximity alert.
4. **onRequestPermissionsResult Method:**
   - Handles the result of the permission request.
   - If permission is granted, calls 'addProximityAlert()'.
   - If denied, shows a toast message indicating that location permission is required.
5. **addProximityAlert Method:**
   - Sets up a proximity alert for the specified latitude, longitude, and radius.
   - Creates a 'PendingIntent' that launches the activity with a custom action when the proximity alert is triggered.
6. **showAlert Method:**
   - Displays an alert dialog when the proximity alert is triggered.

Output

When the application runs, it requests location permissions from the user. Once permissions are granted, the application begins monitoring the specified location, which is the Skyward Publishers Address with coordinates (12.95899, 77.56113).



1. **Outside the Specified Location:**
   - Initially, the application displays a message "Monitoring Location" on the screen.
   - The user will see this message as long as they are outside the 5-meter radius of the specified coordinates.
2. **Entering the Specified Location:**
   - When the user enters the 5-meter radius around the specified coordinates, the proximity alert is triggered.
   - The application displays an alert dialog with the title "Proximity Alert" and the message "You are near the specified location! Monitoring Location."

7.4 Review Questions

Section - A

1. What is the purpose of the `SmsManager` class in Android?
2. Which permission is required to send and receive SMS messages in an Android application?
3. What is the basic use of an Intent with the action `ACTION_SEND` in Android?

4. What is the role of BroadcastReceiver in SMS Messaging?
5. What is the role of SmsManager Class?
6. How Intents are used for Email Sending?
7. What is the use of putExtra() in email sending?
8. What are the Basic Steps to Send an Email in Android?
9. What are Location Based Services?
10. Write any two applications of Location Based Services.
11. List Location Detection Methods.
12. What is the significance of Google API Key?
13. Name two key components of Location-Based Services (LBS).
14. Which Android class is used to manage location updates and proximity alerts?
15. What method is used to add a proximity alert in the `LocationManager` class?
16. What are the basic features of the Google Maps API integration in Android applications?

Section -B

1. Explain SMS Related Classes and Methods in Android.
2. Explain the basic steps to send an SMS using `SmsManager` in an Android application.
3. Explain the Key Points on Email Messaging in Android.
4. How can an application pre-fill email details like recipient, subject, and body using an Intent? Provide an example.
5. What are Location Based Services? Explain the Key Components of LBS.
6. Write the Key Features of Location-Based Services.
7. Describe the role of the API key in integrating Google Maps into an Android application and explain how to obtain one.
8. Explain Lifecycle Methods of Map View.
9. What are the different types of views available in Google Maps? Describe their uses.
10. Explain how geocoding and reverse geocoding are used in Android applications. Provide a brief code example.
11. What steps are involved in setting up location monitoring using `LocationManager` in an Android application?
12. Describe the process of adding a marker to a specific location on Google Maps in an Android application.
13. Discuss the importance of real-time location tracking in location-based services (LBS) and provide an example of its application.
14. How to implement a basic UI for sending SMS in an Android application? Explain with code example.

Section - C

1. Describe the step-by-step implementation of sending a SMS Message in an Android application.
2. Describe the step-by-step implementation of sending an Email in an Android application.
3. Provide a detailed explanation of how to integrate Google Maps into an Android application.
4. Explain Navigating to a Specific Location in Google Maps with detailed code example.
5. Explain the process of monitoring a location using the `LocationManager` class in Android.
6. How can developers use geocoding and reverse geocoding to enhance the functionality of location-based applications? Provide code examples and discuss potential use cases.
7. Detail the implementation of real-time location tracking in an Android application.
8. Explain the different map views provided by Google Maps and demonstrate how to switch between these views programmatically in an Android application.
9. Describe the process of creating and managing a proximity alert in an Android application.

UNIT - IV

CHAPTER 8

PUTTING SQL TO WORK

CONTENTS

- Introduction to SQLite
  - Features of SQLite
  - Common Applications of SQLite Database
  - SQLite in Android
  - Advantages and Disadvantages of SQLite
- In and Out of SQLite
- Hello, Database
- Data Binding
- Review Questions

8.1 Introduction to SQLite

SQLite is a compact, self-contained, and efficient database engine designed by Dr. Richard Hipp in 2000. SQLite has been widely embraced across various platforms and applications, including Android and iOS devices, Mozilla Firefox, Skype, PHP, Adobe AIR, Mac OS X, and Solaris. Its self-contained nature, ease of integration, and support for standard SQL queries make it a popular choice for developers seeking a reliable and efficient database solution.

What is SQLite ?

SQLite is a lightweight, serverless, self-contained, and open-source relational database management system that is embedded within the application. It is widely used in mobile and embedded devices due to its simplicity, speed, reliability, and small footprint. SQLite databases are stored as a single file on the disk, making them easy to manage and deploy.

8.1.1 Features of SQLite

SQLite offers several key features that make it a popular choice for developers and applications:

1. **Self-Contained:** SQLite is a serverless database system that operates without the need for a separate server process. The entire database is contained within a single disk file, simplifying deployment and management.
2. **Zero Configuration:** SQLite requires minimal setup and administration, making it easy to use out of the box. There is no need for complex configuration files or setup procedures.
3. **Transactional Support:** SQLite supports ACID properties (Atomicity, Consistency, Isolation, Durability) to ensure data integrity and reliability. Transactions can be managed to maintain the consistency of the database.
4. **Cross-Platform Compatibility:** SQLite databases can be used across different platforms without modification. This allows for seamless data sharing and portability between various operating systems and devices.
5. **Small Footprint:** The SQLite library is lightweight and has a minimal memory footprint and hence it is suitable for embedded systems, mobile devices, and applications where resources are limited.
6. **Standard SQL Support:** SQLite supports most of the standard SQL syntax for querying and manipulating data. Developers can leverage familiar SQL commands to interact with the database.
7. **High Performance:** SQLite is known for its speed and efficiency in handling database operations. It is optimized for read and write operations, making it suitable for applications requiring fast data access.
8. **Open-Source:** SQLite is open-source software released into the public domain, allowing developers to use it freely without any licensing fees. This open nature encourages community contributions and continuous improvement.

8.1.2 Common Applications of SQLite Database

SQLite is commonly used in a wide range of applications and scenarios due to its versatility, simplicity, and efficiency. Some common applications where SQLite is utilized include:

1. **Mobile Apps:** SQLite is extensively used in mobile applications for storing local data, such as user preferences, settings, offline content, and cached information. Both Android and iOS platforms rely on SQLite as the default database engine for app development.
2. **Web Browsers:** Many web browsers, including Mozilla Firefox, use SQLite to store browsing history, bookmarks, cookies, and other user-related data. SQLite's speed and reliability make it well-suited for managing browser data efficiently.
3. **Embedded Systems:** SQLite is a popular choice for embedded systems and IoT devices where a lightweight and self-contained database solution is required. It is used in various embedded applications, such as smart devices, industrial automation, and consumer electronics.
4. **Desktop Applications:** SQLite is commonly integrated into desktop applications for local data storage and management. It is used in software applications across different industries, including productivity tools, media players, project management software, and more.
5. **Data Analysis Tools:** SQLite is employed in data analysis and reporting tools for storing and querying structured data. It provides a reliable and efficient way to manage datasets, perform data manipulation, and generate insights for decision-making.

These applications demonstrate the diverse use cases of SQLite across different domains, highlighting its flexibility and effectiveness as a database solution for various software projects.

8.1.3 SQLite in Android

SQLite in Android provides a lightweight and efficient way to manage local data storage and offers full relational database capabilities directly on the device. Each Android application can create and manage its own SQLite database.

Key Features of SQLite in Android

1. **Integration:** SQLite comes integrated into the Android operating system and hence it eliminates the need for additional setup or configuration. Developers can start using SQLite databases within their applications without any extra installation steps.
2. **Performance:** SQLite is optimized for performance in mobile environments and offers fast read and write operations with minimal memory and disk space usage. Its lightweight nature makes it well-suited for resource-constrained devices.
3. **ACID Compliance:** SQLite ensures ACID properties, which stand for Atomicity, Consistency, Isolation, and Durability. This means that database transactions are reliable, ensuring that operations are either fully completed or rolled back in case of failures, maintaining data integrity.
4. **Flexibility:** SQLite supports a wide range of SQL features, allowing developers to create complex queries, define relationships between data tables, and perform various data manipulation tasks efficiently. It provides a full set of relational database capabilities for managing structured data.

5. **Security:** SQLite in Android provides built-in security features to protect data stored in the database. Developers can implement encryption mechanisms to secure sensitive information to ensure that data remains confidential and protected from unauthorized access.

Typical Use Cases of SQLite in Android

1. **Storing User Preferences and Settings:** SQLite databases are commonly used to store user-specific preferences, settings, and configurations within an application. This allows users to customize their experience and save their preferences for future sessions.
2. **Caching Data for Offline Access:** Applications often use SQLite to cache data locally for offline access. By storing frequently accessed data in a SQLite database, apps can provide seamless functionality even when the device is not connected to the internet.
3. **Maintaining a Local Copy of Remote Data:** SQLite is utilized to maintain a local copy of data fetched from remote servers or APIs. This local storage reduces network dependencies, speeds up data retrieval, and enables offline usage of data that is regularly synchronized with the server.
4. **Handling Complex Queries and Data Relationships:** SQLite's support for SQL queries and relational database features allows developers to manage complex data structures, perform joins across multiple tables, and establish relationships between different data entities. This capability is beneficial for applications dealing with intricate data models.
5. **Offline Data Synchronization:** SQLite is commonly used in Android applications to facilitate offline data synchronization processes. In scenarios where the app needs to synchronize local data with a remote server when connectivity is available, SQLite databases serve as a reliable storage medium for managing synchronization tasks.

8.1.4 Advantages and Disadvantages of SQLite

Advantages of SQLite in Android

1. **Ease of Use:** SQLite is easy to set up and use within Android applications, making it accessible to developers of all skill levels.
2. **Zero Configuration:** As an embedded database, SQLite requires no server setup or administration, simplifying the deployment process.
3. **Lightweight:** SQLite has a small footprint, consuming minimal memory and disk space, which is ideal for mobile devices with limited resources.
4. **ACID Compliance:** SQLite ensures data integrity through ACID properties, providing reliable transaction support for maintaining database consistency.
5. **Portability:** SQLite databases are portable and can be easily transferred between devices, allowing for seamless data migration and sharing.

Disadvantages of SQLite in Android

1. **Limited Scalability:** SQLite may not be suitable for applications requiring high scalability or concurrent access to the database by multiple users.
2. **Performance Constraints:** In scenarios with complex queries or large datasets, SQLite's performance may degrade compared to server-based databases like MySQL or PostgreSQL.

3. **Lack of Client-Server Architecture:** SQLite lacks a client-server architecture, which can limit its suitability for applications requiring centralized data management and multi-user access.
4. **Security Concerns:** While SQLite supports encryption, it may not offer the same level of security features as dedicated database servers, raising potential security vulnerabilities.
5. **Data Recovery:** In case of database corruption or failure, SQLite may have limitations in terms of data recovery and restoration compared to server-based databases with robust backup mechanisms.

8.2 In and Out of SQLite

Understanding the basics of SQLite is essential for developing Android applications because it provides a robust and efficient way to manage local data storage. SQLite is integrated into the Android operating system to create, read, update, and delete data in a relational database with minimal overhead. Its lightweight, serverless architecture makes it perfect for mobile environments where resources are limited. Mastering SQLite's core concepts, such as creating tables, inserting data, updating records, querying information, and enforcing data integrity through constraints are needed to build powerful and responsive Android apps that handle data effectively.

Data Definition Language (DDL) in SQLite

Data Definition Language (DDL) in SQLite is used to define and manage the structure of the database and its objects, such as tables, indexes, and views. DDL statements include 'CREATE', 'ALTER', and 'DROP'. These statements are crucial for setting up the schema of a database to ensure that it is well-organized and that data integrity is maintained.

DDL Statement and Description	Example and Explanation
CREATE TABLE: The 'CREATE TABLE' statement is used to create a new table in the database. It defines the structure of the table, including the names of the columns, their data types, and any constraints on the columns.	**Example:** To create a table named 'employees': CREATE TABLE employees (id INTEGER PRIMARY KEY, name TEXT NOT NULL, position TEXT, salary REAL CHECK(salary > 0)); **Explanation:** • **'id INTEGER PRIMARY KEY':** Creates a column named 'id' with an integer data type and sets it as the primary key. The primary key constraint ensures that each value in the 'id' column is unique and not null. • **'name TEXT NOT NULL':** Creates a column named 'name' with a text data type and a NOT NULL constraint. This ensures that every row must have a value for this column. • **'position TEXT':** Creates a column named 'position' with a text data type. • **'salary REAL CHECK(salary > 0)':** Creates a column named 'salary' with a real number data type and a CHECK constraint. This constraint ensures that the salary must be greater than zero.

ALTER TABLE: The 'ALTER TABLE' statement is used to modify the structure of an existing table. It can be used to add, remove, or rename columns in a table.	**Example:** To add a new column named 'hire_date' to the 'employees' table: ALTER TABLE employees ADD COLUMN hire_date TEXT; **Explanation:** This statement adds a new column named 'hire_date' with a text data type to the 'employees' table.
DROP TABLE: The 'DROP TABLE' statement is used to delete an existing table from the database. This statement permanently removes the table and all of its data, so it should be used with caution.	**Example:** To delete the 'employees' table if it exists: DROP TABLE IF EXISTS employees; **Explanation:** This statement deletes the 'employees' table if it exists. If the table does not exist, the 'IF EXISTS' clause prevents an error from occurring.

These DDL statements provide the fundamental operations required to define and manage the structure of a database in SQLite. Each statement plays a critical role in setting up, modifying, and maintaining the database schema.

Constraints in SQLite

Constraints are rules enforced on data columns to ensure the accuracy and reliability of the data within the database. Common constraints in SQLite include 'PRIMARY KEY', 'NOT NULL', 'UNIQUE', 'CHECK', and 'FOREIGN KEY'.

Constraint	Explanation	Example and Explanation
PRIMARY KEY	A primary key constraint uniquely identifies each row in the table. Primary keys must contain unique values and cannot be null.	`id INTEGER PRIMARY KEY` Ensures that the 'id' column uniquely identifies each row.
NOT NULL	A NOT NULL constraint ensures that a column cannot have a null value.	`name TEXT NOT NULL` Ensures that the 'name' column cannot contain null values.
UNIQUE	A UNIQUE constraint ensures that all values in a column are unique.	`email TEXT UNIQUE` Ensures that all values in the 'email' column are unique.
CHECK	A CHECK constraint ensures that all values in a column satisfy a specific condition.	`salary REAL CHECK(salary > 0)` Ensures that the 'salary' column contains only positive values.
FOREIGN KEY	A FOREIGN KEY constraint ensures that the value in a column matches a value in another table, establishing a relationship between the two tables.	`FOREIGN KEY (department_id) REFERENCES departments(id)` Ensures that the value in the 'department_id' column matches a value in the 'id' column of the 'departments' table.

Example Schema with Explanation

Code A complete example that combines multiple DDL statements and constraints.

```
CREATE TABLE departments (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL UNIQUE
);

CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    position TEXT,
    salary REAL CHECK(salary > 0),
    department_id INTEGER,
    FOREIGN KEY (department_id) REFERENCES departments(id)
);
```

Explanation

1. **'departments' table:**
   - **'id INTEGER PRIMARY KEY':** A unique identifier for each department. This column cannot be null and must contain unique values.
   - **'name TEXT NOT NULL UNIQUE':** The name of the department. This column cannot be null and must contain unique values, ensuring no two departments have the same name.
2. **'employees' table:**
   - **'id INTEGER PRIMARY KEY':** A unique identifier for each employee. This column cannot be null and must contain unique values.
   - **'name TEXT NOT NULL':** The name of the employee. This column cannot be null.
   - **'position TEXT':** The job position of the employee.
   - **'salary REAL CHECK(salary > 0)':** The salary of the employee. This column must contain positive values.
   - **'department_id INTEGER':** A foreign key that references the 'id' column in the 'departments' table, establishing a relationship between the 'employees' and 'departments' tables.

Data Manipulation Language (DML) in SQLite

Data Manipulation Language (DML) in SQLite is used to manage data within schema objects. The primary DML statements in SQLite include 'INSERT', 'UPDATE', and 'DELETE'. These statements allow for the addition, modification, and removal of data in the database.

DML Statement and Description	Example and Explanation
INSERT: The 'INSERT' statement is used to add new rows to a table. It specifies the table name, the columns to insert data into, and the corresponding values. This operation adds new data to the database.	`INSERT INTO employees (name, position, salary) VALUES ('Swati', 'Developer', 60000);` Inserts a new row into the 'employees' table with the values "Swati", "Developer", and '60000'.

UPDATE: The 'UPDATE' statement is used to modify existing rows in a table. It specifies the table name, the columns to update with new values, and a condition to determine which rows should be updated. This operation changes existing data in the database.	`UPDATE employees SET salary = 65000 WHERE name = 'Swati';` Updates the 'salary' of the employee named 'Swati' to '65000'.
DELETE: The 'DELETE' statement is used to remove rows from a table. It specifies the table name and a condition to determine which rows should be deleted. This operation removes data from the database.	`DELETE FROM employees WHERE name = 'Swati';` Deletes the row from the 'employees' table where the 'name' is 'Swati'.

These DML statements provide the essential operations for managing and manipulating data within SQLite databases. Each statement serves a specific purpose in handling the data stored in the database tables.

Query Statements (Select Statements) in SQLite

Query statements in SQLite are used to retrieve data from the database. The primary query statement is 'SELECT', which allows for the selection of specific data from one or more tables based on various conditions. Query statements can be simple or complex, involving joins, filtering, sorting, and grouping.

Query Statement	Description	Example and Explanation
SELECT	Retrieves specific data from one or more tables.	`SELECT name, position FROM employees WHERE salary > 50000 ORDER BY name LIMIT 10;` Retrieves the 'name' and 'position' of employees with a salary greater than 50000, sorted by 'name' and limited to 10 rows.
JOIN	Combines rows from two or more tables based on a related column.	`SELECT employees.name, departments.name AS department FROM employees JOIN departments ON employees.department_id = departments.id;` Retrieves employee names and their corresponding department names.
GROUP BY	Arranges identical data into groups.	`SELECT department_id, COUNT(*) FROM employees GROUP BY department_id;` Groups employees by 'department_id' and counts the number of employees in each department.
HAVING	Filters groups based on a condition, often used with 'GROUP BY'.	`SELECT department_id, AVG(salary) FROM employees GROUP BY department_id HAVING AVG(salary) > 50000;` Filters departments to include only those with an average salary greater than 50000.

8.3 Hello, Database

The title "Hello, Database" is used as an introduction to the concept of working with databases in Android applications. Similar to how "Hello, World" programs introduce the basics of a programming language or environment, "Hello, Database" aims to introduce the fundamentals of using SQLite databases in Android. It signifies the starting point of learning how to create, manage, and interact with a database within an Android application.

SQLite is an embedded relational database management system that is widely used in Android applications for local data storage. Understanding how to create and manage a SQLite database is essential for developing robust Android applications. This section explains the step-by-step implementation of creating a database in android.

Key Classes involved in Managing SQLite Databases in Android

SQLite in Android involves two primary classes to manage and interact with the database: **'SQLiteOpenHelper'** and **'SQLiteDatabase'**. These classes provide the necessary methods to create, upgrade, and perform various operations on the database.

1. **SQLiteOpenHelper:**

   'SQLiteOpenHelper' is an abstract class that simplifies database management tasks, such as creating and upgrading the database. It provides a framework to handle database operations without having to manage the underlying SQLite database manually.

Method	Description
SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)	Constructor used to create a helper object to manage a database.
onCreate(SQLiteDatabase db)	Called when the database is created for the first time. This is where the creation of tables and the initial population of the tables should happen.
onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)	Called when the database needs to be upgraded. This method is used to handle changes to the database schema, such as adding new columns or tables.
getReadableDatabase()	Returns a database object that can be used for reading from the database. If the database cannot be opened for reading, it will be opened for writing.
getWritableDatabase()'	Returns a database object that can be used for writing to the database. If the database cannot be opened for writing, it will throw an exception.
close()	Closes the database and releases any system resources associated with it.

2. **SQLiteDatabase:**

   'SQLiteDatabase' provides methods to perform CRUD (Create, Read, Update, Delete) operations on the database. It represents the database itself and offers methods to execute SQL commands and queries.

Method	Description
insert(String table, String nullColumnHack, ContentValues values)	Inserts a new row into the database table. Returns the row ID of the newly inserted row, or -1 if an error occurred.
update(String table, ContentValues values, String whereClause, String[] whereArgs)	Updates rows in the database table. Returns the number of rows affected.
delete(String table, String whereClause, String[] whereArgs)	Deletes rows from the database table. Returns the number of rows affected.
query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy)	Queries the database table and returns a 'Cursor' object.
rawQuery(String sql, String[] selectionArgs)	Executes a raw SQL query and returns a 'Cursor' object.
execSQL(String sql)	Executes a single SQL statement that is not a SELECT or any other SQL statement that returns data.
beginTransaction()	Begins a transaction.
setTransactionSuccessful()	Marks the current transaction as successful.
endTransaction()	Ends a transaction. If the transaction was marked as successful, it is committed; otherwise, it is rolled back.
close()	Closes the database.

SQLite Database Operations in Android

It's important to understand the basic database operations in SQLite. Let us understand the creating, inserting, updating, deleting, and selecting data in an SQLite database. We will create a Contacts table to store the contact details. The operations we will perform on this table include creating the table, inserting data, updating data, deleting data, and selecting (retrieving) data. Each of these operations will be explained with code snippets and detailed explanations.

1. **Creating a Database:**

   Creating a database in SQLite involves defining the structure of the database by creating tables. This is typically done in the 'onCreate' method of a helper class that extends 'SQLiteOpenHelper'. The table creation is done using SQL 'CREATE TABLE' statements.

Code Snippet

```
public class DBHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "ContactsDB";
    private static final int DATABASE_VERSION = 1;
    private static final String TABLE_CONTACTS = "contacts";
    private static final String KEY_ID = "id";
    private static final String KEY_NAME = "name";
```

```
    private static final String KEY_PHONE = "phone";

    public DBHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        String CREATE_CONTACTS_TABLE = "CREATE TABLE " + TABLE_CONTACTS + "("
                + KEY_ID + " INTEGER PRIMARY KEY AUTOINCREMENT,"
                + KEY_NAME + " TEXT,"
                + KEY_PHONE + " TEXT" + ")";
        db.execSQL(CREATE_CONTACTS_TABLE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_CONTACTS);
        onCreate(db);
    }
}
```

Explanation

- **onCreate(SQLiteDatabase db):** This method is called when the database is created for the first time. It executes the SQL statement to create a table named 'contacts' with three columns: 'id' (primary key, auto-incremented), 'name' (text), and 'phone' (text).
- **onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion):** This method is called when the database needs to be upgraded. It drops the existing table and calls 'onCreate' to create a new one.

2. **Inserting Data:**

Inserting data into an SQLite database involves creating a 'ContentValues' object, putting the data into it, and then calling the 'insert' method on the database instance.

Code Snippet

```
public void addContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put(KEY_NAME, contact.getName());
    values.put(KEY_PHONE, contact.getPhoneNumber());
    db.insert(TABLE_CONTACTS, null, values);
    db.close();
}
```

Explanation

- **getWritableDatabase():** This method is called to get a writable instance of the database.
- **ContentValues values = new ContentValues():** 'ContentValues' is used to store the values that will be inserted into the database.
- **values.put(KEY_NAME, contact.getName()):** The 'put' method adds a value to the 'ContentValues' object. Here, it adds the contact's name.
- **values.put(KEY_PHONE, contact.getPhoneNumber():** Similarly, this adds the contact's phone number.
- **db.insert(TABLE_CONTACTS, null, values):** The 'insert' method inserts the data into the specified table.
- **db.close():** The database is closed to release resources.

3. **Updating Data:**

Updating data in SQLite involves creating a 'ContentValues' object with the updated data and calling the 'update' method on the database instance.

Code Snippet

```
public void updateContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put(KEY_NAME, contact.getName());
    values.put(KEY_PHONE, contact.getPhoneNumber());
     db.update(TABLE_CONTACTS, values, KEY_ID + " = ?", new String[]{String.
valueOf(contact.getId())});
    db.close();
}
```

Explanation

- **ContentValues values = new ContentValues():** A 'ContentValues' object is created to store the updated values.
- **values.put(KEY_NAME, contact.getName()):** The updated name is put into the 'ContentValues' object.
- **values.put(KEY_PHONE, contact.getPhoneNumber()):** The updated phone number is put into the 'ContentValues' object.
- **db.update(TABLE_CONTACTS, values, KEY_ID + "=?", new String[]{String.valueOf(contact.getId())}):** The 'update' method updates the row in the table where the 'id' matches the contact's id.
- **db.close():** The database is closed to release resources.

4. **Deleting Data:**

Deleting data from an SQLite database involves calling the 'delete' method on the database instance with a condition to specify which rows to delete.

Code Snippet

```
public void deleteContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();
    db.delete(TABLE_CONTACTS, KEY_ID + " = ?", new String[]
                                    {String.valueOf(contact.getId())});
    db.close();
}
```

Explanation

- **db.delete(TABLE_CONTACTS, KEY_ID + " = ?", new String[]{String.valueOf(contact.getId())}):** The 'delete' method deletes the row in the table where the 'id' matches the contact's id.
- **db.close():** The database is closed to release resources.

5. **Selecting Data:**

Selecting data from an SQLite database involves querying the database and retrieving the results using a 'Cursor' object.

Code Snippet

```
public List<Contact> getAllContacts() {
    List<Contact> contactList = new ArrayList<>();
    String selectQuery = "SELECT * FROM " + TABLE_CONTACTS;
    SQLiteDatabase db = this.getWritableDatabase();
    Cursor cursor = db.rawQuery(selectQuery, null);

    if (cursor.moveToFirst()) {
        do {
            Contact contact = new Contact(cursor.getInt(0),
                                cursor.getString(1), cursor.getString(2));
            contactList.add(contact);
        } while (cursor.moveToNext());
    }

    cursor.close();
    return contactList;
}
```

Explanation

- **String selectQuery = "SELECT * FROM " + TABLE_CONTACTS:** This SQL query selects all rows from the 'contacts' table.
- **Cursor cursor = db.rawQuery(selectQuery, null):** The 'rawQuery' method executes the SQL query and returns a 'Cursor' object containing the results.
- **if (cursor.moveToFirst()):** This checks if the cursor is not empty and moves it to the first row.
- **do { ... } while (cursor.moveToNext()):** This loop iterates over all rows in the cursor.

- **Contact contact = new Contact(cursor.getInt(0), cursor.getString(1), cursor.getString(2)):** A new 'Contact' object is created for each row in the cursor.
- **contactList.add(contact):** The contact is added to the list.
- **cursor.close():** The cursor is closed to release resources.

Step-by-Step Implementation of Creating and Using a Database in Android

In this section, we'll create a simple Android application that allows users to add, delete, update contacts, and view them in a list view. We'll use SQLite for database operations. Below are the step-by-step instructions and code snippets for each functionality:

Step 1: Create a New Android Project:

Create a new Android project named "**ContactsApp**" with an Empty View Activity. This sets up the basic structure of the project where we will implement our functionalities.

Step 2: Define the Contact Model:

Create a 'Contact' class to represent a contact with name and phone number fields. This model will be used to encapsulate the contact data.

Code Contact.java

```
package com.example.contactsapp;

public class Contact {
    private int id;
    private String name;
    private String phoneNumber;

    public Contact(int id, String name, String phoneNumber) {
        this.id = id;
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    @Override
    public String toString() {
        return name + " - " + phoneNumber;
    }
}
```

Explanation

This class has private fields for id, name, and phone number, along with their getters and setters. The 'toString()' method is overridden to provide a readable representation of the contact.

Step 3: Create the User Interface:

Create the user interface with EditText fields for name and phone number, buttons for adding, deleting, and updating contacts, and a ListView to display contacts.

Code activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <EditText
        android:id="@+id/editTextName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Name" />

    <EditText
        android:id="@+id/editTextPhone"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Phone Number" />

    <Button
        android:id="@+id/btnAdd"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Add Contact" />
```

```
    <Button
        android:id="@+id/btnUpdate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Update Contact" />

    <Button
        android:id="@+id/btnDelete"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Delete Contact" />

    <ListView
        android:id="@+id/listView"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:choiceMode="singleChoice" />
</LinearLayout>
```

Explanation

This layout file defines the user interface with input fields for name and phone number, buttons for CRUD operations, and a ListView to display the contacts.

Step 4: Create a Database Helper Class:

Create a 'DBHelper' class to handle database operations like creating tables, inserting, updating, and deleting contacts. This class extends 'SQLiteOpenHelper' which helps manage database creation and version management.

Code DBHelper.java

```
package com.example.contactsapp;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

import java.util.ArrayList;
import java.util.List;

// DBHelper.java
public class DBHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "ContactsDB";
```

```
    private static final int DATABASE_VERSION = 1;
    private static final String TABLE_CONTACTS = "contacts";
    private static final String KEY_ID = "id";
    private static final String KEY_NAME = "name";
    private static final String KEY_PHONE = "phone";

    public DBHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {

        String CREATE_CONTACTS_TABLE = "CREATE TABLE " + TABLE_CONTACTS + "("
                + KEY_ID + " INTEGER PRIMARY KEY AUTOINCREMENT,"
                + KEY_NAME + " TEXT,"
                + KEY_PHONE + " TEXT" + ")";
        db.execSQL(CREATE_CONTACTS_TABLE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_CONTACTS);
        onCreate(db);
    }

    public void addContact(Contact contact) {
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(KEY_NAME, contact.getName());
        values.put(KEY_PHONE, contact.getPhoneNumber());
        db.insert(TABLE_CONTACTS, null, values);
        db.close();
    }

    public List<Contact> getAllContacts() {
        List<Contact> contactList = new ArrayList<>();
        String selectQuery = "SELECT * FROM " + TABLE_CONTACTS;
        SQLiteDatabase db = this.getWritableDatabase();
        Cursor cursor = db.rawQuery(selectQuery, null);

        if (cursor.moveToFirst()) {
            do {
```

```
                Contact contact = new Contact(cursor.getInt(0),
                                        cursor.getString(1), cursor.getString(2));
                contactList.add(contact);
            } while (cursor.moveToNext());
        }

        cursor.close();
        return contactList;
    }

    public void updateContact(Contact contact) {
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(KEY_NAME, contact.getName());
        values.put(KEY_PHONE, contact.getPhoneNumber());
        db.update(TABLE_CONTACTS, values, KEY_ID + " = ?",
                                new String[]{String.valueOf(contact.getId())});
        db.close();
    }

    public void deleteContact(Contact contact) {
        SQLiteDatabase db = this.getWritableDatabase();
        db.delete(TABLE_CONTACTS, KEY_ID + " = ?",
                                new String[]{String.valueOf(contact.getId())});
        db.close();
    }
}
```

Explanation

- **onCreate:** This method is called when the database is created for the first time. It executes an SQL statement to create the contacts table.
- **onUpgrade:** This method is called when the database needs to be upgraded. It drops the existing table and creates a new one.
- **addContact:** This method inserts a new contact into the database.
- **getAllContacts:** This method retrieves all contacts from the database and returns them as a list.
- **updateContact:** This method updates an existing contact in the database.
- **deleteContact:** This method deletes a contact from the database.

Step 5: Implement MainActivity:

Implement the MainActivity to interact with the database and update the UI accordingly. This activity will handle adding, updating, deleting contacts, and displaying them in a ListView.

Code MainActivity.java

```
package com.example.contactsapp;

import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;

import androidx.appcompat.app.AppCompatActivity;
import java.util.ArrayList;
import java.util.List;

// MainActivity.java
public class MainActivity extends AppCompatActivity {
    private EditText editTextName, editTextPhone;
    private Button btnAdd, btnUpdate, btnDelete;
    private ListView listView;
    private List<Contact> contactList = new ArrayList<>();
    private ArrayAdapter<Contact> adapter;
    private DBHelper dbHelper;
    private Contact selectedContact;

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

        editTextName = findViewById(R.id.editTextName);
        editTextPhone = findViewById(R.id.editTextPhone);
        btnAdd = findViewById(R.id.btnAdd);
        btnUpdate = findViewById(R.id.btnUpdate);
        btnDelete = findViewById(R.id.btnDelete);
        listView = findViewById(R.id.listView);

        dbHelper = new DBHelper(this);

        // Load contacts from the database
        loadContacts();

        btnAdd.setOnClickListener(v -> {
            String name = editTextName.getText().toString();
            String phone = editTextPhone.getText().toString();
```

```
            dbHelper.addContact(new Contact(0, name, phone));
            loadContacts();
            clearInputs();
        });

        btnUpdate.setOnClickListener(v -> {
            if (selectedContact != null) {
                selectedContact.setName(editTextName.getText().toString());
              selectedContact.setPhoneNumber(editTextPhone.getText().toString());
                dbHelper.updateContact(selectedContact);
                loadContacts();
                clearInputs();
            }
        });

        btnDelete.setOnClickListener(v -> {
            if (selectedContact != null) {
                dbHelper.deleteContact(selectedContact);
                loadContacts();
                clearInputs();
            }
        });

        listView.setOnItemClickListener((parent, view, position, id) -> {
            selectedContact = contactList.get(position);
            editTextName.setText(selectedContact.getName());
            editTextPhone.setText(selectedContact.getPhoneNumber());
        });
    }

    private void loadContacts() {
        contactList.clear();
        contactList.addAll(dbHelper.getAllContacts());
        adapter = new ArrayAdapter<>(this,
                 android.R.layout.simple_list_item_single_choice, contactList);
        listView.setAdapter(adapter);
    }

    private void clearInputs() {
        editTextName.setText("");
        editTextPhone.setText("");
        selectedContact = null;
    }
}
```

Explanation

1. **UI Components:**
   - 'EditText' fields ('editTextName' and 'editTextPhone') are used for inputting the contact's name and phone number.
   - 'Button' components ('btnAdd', 'btnUpdate', and 'btnDelete') are used to perform add, update, and delete operations, respectively.
   - 'ListView' is used to display the list of contacts.
2. **Database Operations:**
   - **Add Contact:**
     - Gets the name and phone number from the EditText fields.
     - Calls 'dbHelper.addContact' to insert the new contact into the database.
     - Reloads the contacts to update the ListView and clears the input fields.
   - **Update Contact:**
     - Checks if a contact is selected.
     - Updates the selected contact's details in the database using 'dbHelper.updateContact'.
     - Reloads the contacts to update the ListView and clears the input fields.
   - **Delete Contact:**
     - Checks if a contact is selected.
     - Deletes the selected contact from the database using 'dbHelper.deleteContact'.
     - Reloads the contacts to update the ListView and clears the input fields.
3. **ListView Item Click:**
   - When a contact is clicked in the ListView, the selected contact's details are populated in the EditText fields for editing.
4. **loadContacts Method:**
   - Clears the current list of contacts.
   - Adds all contacts from the database to the contactList.
   - Initializes the 'ArrayAdapter' with the updated list and sets it to the ListView.
5. **clearInputs Method:**
   - Clears the EditText fields and resets the selected contact to 'null'.
6. **ArrayAdapter Initialization:**

   adapter = new ArrayAdapter<>(this, android.R.layout.simple_list_item_single_choice, contactList);

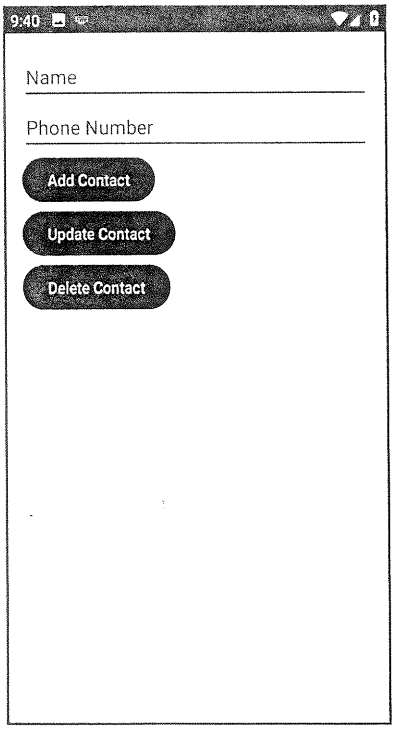
listView.setAdapter(adapter);

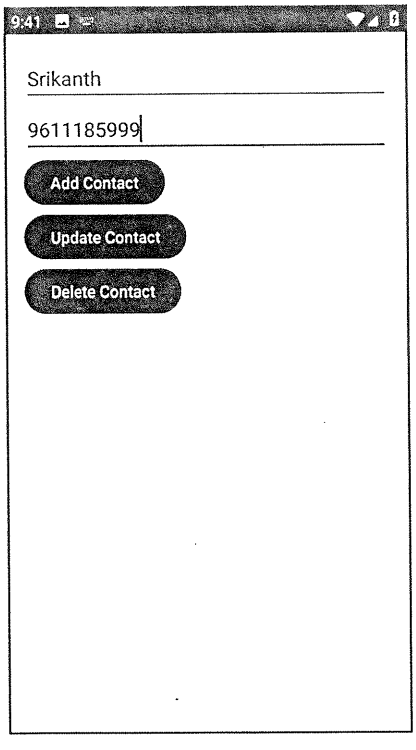
- 'ArrayAdapter' is used to bind the list of contacts to the ListView.
- 'android.R.layout.simple_list_item_single_choice' is the layout resource used for each item in the ListView.
- 'contactList' is the list of contacts to be displayed.

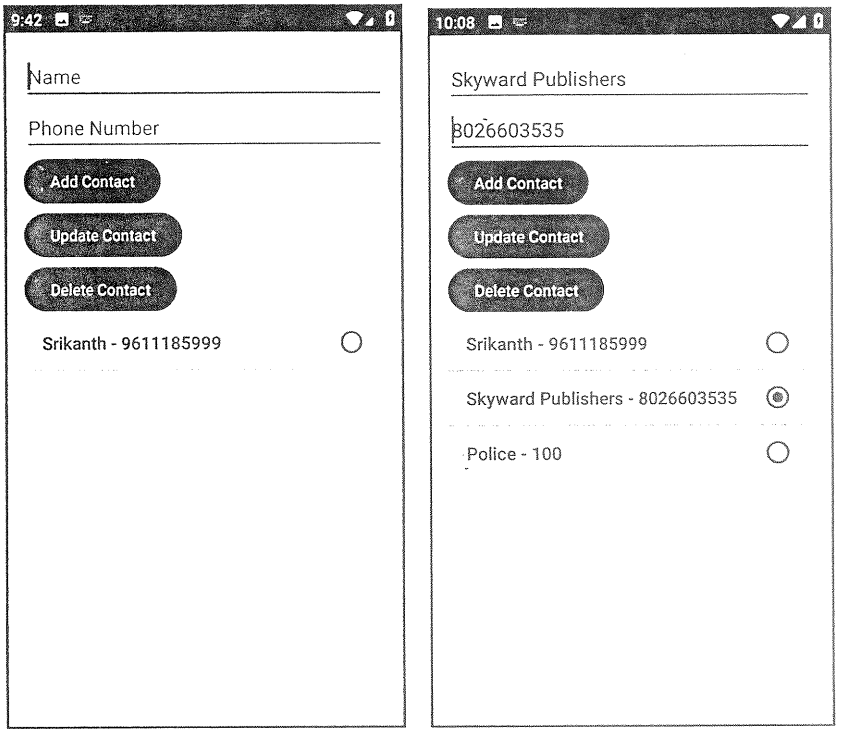
Output

When the application is launched:

- The user can enter a name and phone number in the EditText fields.
- By clicking the "Add Contact" button, the contact is added to the database, and the ListView is updated to display the new contact.
- When a contact in the ListView is clicked, the contact's details are populated in the EditText fields for editing.
- By clicking the "Update Contact" button, the selected contact's details are updated in the database and the ListView.
- By clicking the "Delete Contact" button, the selected contact is removed from the database and the ListView.







8.4 Data Binding

Data Binding is a powerful feature in Android development that simplifies the process of connecting user interface components to data sources in an application. It allows developers to establish a direct link between the UI elements and the underlying data to enable seamless synchronization and updates between the two.

Data Binding feature can eliminate the need for manual view lookups and repetitive code for updating UI elements. Changes made to the data source automatically reflect in the UI, and vice versa, without the developer having to manage these updates explicitly. This two-way data binding simplifies the development process and enhances the efficiency of Android app development.

What is Data Binding in Android ?

Data Binding in Android refers to the technique of establishing a connection between the UI elements (such as TextViews, EditTexts, RecyclerViews) in the layout files and the data sources (variables, objects) in the application code. This binding is set up using binding expressions in XML layout files, which define how the UI elements should display or interact with the data.

Examples Traditional Approach vs Data Binding

In the traditional approach, setting up UI components involves repetitive findViewById calls. With Data Binding, this Repetitive code is eliminated. For example, consider a scenario where there are multiple TextViews in a layout:

```
<TextView
    android:id="@+id/textViewName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/textViewAge"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

In the traditional approach, developers would need to find these views in the activity:

```
TextView textViewName = findViewById(R.id.textViewName);
TextView textViewAge = findViewById(R.id.textViewAge);
```

Similarly, event listeners are set programmatically:

```
Button btnAdd = findViewById(R.id.btnAdd);
   btnAdd.setOnClickListener(v -> {
       // add button logic
   });
```

This approach can become cumbersome as the number of UI components grows. Additionally, managing the state and updating the UI components requires more code.

With Data Binding, bindings are declared directly in the XML layout file, and the framework takes care of the rest. This reduces the amount of boilerplate code and makes it easier to manage the UI.

Features and Benefits of Data Binding

The features and benefits of Data Binding include:

1. **Reduced Repetitive Code:**

   In the traditional approach, setting up UI components involves repetitive 'findViewById' calls. With Data Binding, this Repetitive code is eliminated. For example, consider a scenario where there are multiple TextViews in a layout and developers need to find these views in the activity. With Data Binding, developers can directly reference these views in the layout XML and bind them to variables in the activity or fragment.

2. **Two-Way Data Binding:**

   Data Binding allows for two-way data binding, meaning changes in the UI automatically update the data source and vice versa. For example, if an EditText field is bound to a variable 'name', any changes made in the EditText will automatically update the 'name' variable, and changes to the 'name' variable will reflect in the EditText.

3. **Improved Performance:**

   Data Binding improves performance by reducing the overhead of frequent view lookups. Once the bindings are established, the framework handles the updates efficiently.

4. **Type Safety:**

   Data Binding provides compile-time checking of IDs and types, reducing the chances of runtime errors. This ensures that the correct views are bound to the correct variables.

5. **Lifecycle Awareness:**

   Data Binding automatically manages the lifecycle of bound data. This ensures that data consistency is maintained across configuration changes and throughout the app's lifecycle.

6. **Declarative Layouts:**

   By declaring bindings in XML layout files, Data Binding allows for more expressive and maintainable layouts. For example, consider binding a list of items to a RecyclerView:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:items="@{viewModel.itemList}" />
```

In this scenario, the RecyclerView is bound to a list of items from the ViewModel, making the layout more declarative and easier to understand.

Key Concepts and Classes

1. **Binding Layouts:**
   - Data Binding starts with the layout files, which must be wrapped inside a '<layout>' tag. Inside this tag, there can be a '<data>' element and multiple '<variable>' elements. The '<data>' element is where variables and their types are declared, to be used within the layout.
   - For example, a layout might declare a variable for a ViewModel, which is then bound to UI components like EditText, TextView, etc.
2. **Binding Class:**
   - The Data Binding Library generates a binding class for each layout file that includes data binding. This class is named based on the layout file name, converted to camel case and suffixed with "Binding". For instance, for a layout file named 'activity_main.xml', the generated class will be 'ActivityMainBinding'.
   - This binding class holds references to the views in the layout and allows setting and retrieving data directly. It also contains methods to bind the data to the views.
3. **ViewModel:**
   - A ViewModel class is a lifecycle-aware component that holds and manages UI-related data. It separates UI data from UI controllers (like activities or fragments) and helps handle configuration changes.
   - ViewModel properties are usually made observable so that the UI components bound to these properties are automatically updated when the data changes.
4. **Observable Properties:**
   - To make properties observable, the ViewModel can extend 'BaseObservable' and use '@Bindable' annotations or use 'LiveData' objects. This enables the binding to automatically update the UI when the data changes.

5. **Binding Adapters:**
    - Binding Adapters are methods annotated with '@BindingAdapter', which define custom binding logic for specific attributes. These adapters allow defining how a view attribute is set or modified when bound to data.

How It Works?

1. **XML Layout:** The developer defines a layout file with the necessary UI components and binds them to properties in the ViewModel using the '@={}' or '@{}' syntax. For example, an EditText's text property can be bound to a ViewModel's 'name' property.
2. **Generated Binding Class:** When the project is built, the Data Binding Library generates a binding class for each layout file that includes data binding. This class includes methods to bind the UI components to the data in the ViewModel.
3. **Setting Up Binding in Activity/Fragment:** In the Activity or Fragment, the generated binding class is used to inflate the layout and bind the ViewModel. The 'DataBindingUtil.setContentView' method is used to set up the binding. For example, in an activity, 'ActivityMainBinding binding = DataBindingUtil.setContentView(this, R.layout.activity_main);' is used.
4. **Updating the UI:** When data in the ViewModel changes, the bound UI components are automatically updated. This is because the properties in the ViewModel are observable, and any change in these properties triggers an update in the corresponding UI components.
5. **Lifecycle Awareness:** Data binding works seamlessly with lifecycle-aware components such as ViewModel. This ensures that the data and UI remain synchronized even during lifecycle changes like screen rotations.
6. **Custom Binding Logic with Binding Adapters:** Binding Adapters allow defining custom logic for setting attributes on views. For instance, if there's a custom view attribute that needs special handling, a Binding Adapter can be created to handle it.

Example Workflow

1. **Define Layout:** The layout file ('activity_main.xml') includes UI components bound to the ViewModel properties using data binding syntax.
2. **Generate Binding Class:** The Data Binding Library generates 'ActivityMainBinding' class during the build process.
3. **Setup Binding:** In the 'MainActivity', the 'ActivityMainBinding' class is used to bind the layout and the ViewModel.
4. **Automatic UI Updates:** Changes in the ViewModel's observable properties automatically reflect in the UI components.
5. **Lifecycle Management:** The binding integrates with lifecycle-aware components to manage data and UI synchronization efficiently.

Step by Step Implemenation of Data Binding

ıis application demonstrates live data binding in Android. It includes two EditText fields for the ;er's name and phone number. As the user types, the entered data is displayed in a ListView below

the EditText fields. The goal is to bind data directly from the UI components to the ViewModel using Android's Data Binding Library.

Step 1: Creating the Project:

1. Open Android Studio.
2. Create a new project by selecting "Empty Views Activity".
3. Name the project "livedatabindingexample".

Step 2: Enable Data Binding in Gradle:

Enable Data Binding in the app's 'build.gradle' file by adding the following:

Code

```
Code :
android {
    ...
    buildFeatures {
        dataBinding = true
    }
}
```

Explanation

Sync the project by clicking "Sync Now" at the top of the window.

Step 3: Define the Layout File:

Create and define the layout file 'activity_main.xml' with EditText fields for name and phone number and a ListView for displaying the data.

Code activity_main.xml

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

    <data>
        <variable
            name="viewModel"
            type="com.example.livedatabindingexample.UserViewModel" />
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp">

        <EditText
            android:id="@+id/editTextName"
```

```
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Enter Name"
            android:text="@={viewModel.name}" />

        <EditText
            android:id="@+id/editTextPhone"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Enter Phone Number"
            android:text="@={viewModel.phone}" />

        <ListView
            android:id="@+id/listView"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            app:items="@{viewModel.userList}" />
    </LinearLayout>
</layout>
```

Explanation

1. **Data Binding Setup:**

   The layout file begins with the '<layout>' tag, which enables data binding for this layout. Inside this tag, the '<data>' element declares a variable named 'viewModel' of type 'com.example.livedatabindingexample.UserViewModel'. This sets up a link between the UI and the ViewModel.

2. **EditText for Name:**

   An 'EditText' component is defined for entering the name, with an 'android:id' of '@+id/editTextName'. The text property is bound to the 'name' property of the ViewModel using '@={viewModel.name}'. This two-way binding ensures that any changes in the EditText update the ViewModel and vice versa.

3. **EditText for Phone Number:**

   Similar to the name field, another 'EditText' component is defined for entering the phone number, with an 'android:id' of '@+id/editTextPhone'. Its text property is bound to the 'phone' property of the ViewModel using '@={viewModel.phone}', ensuring two-way data synchronization.

4. **ListView for Displaying Users:**

   A 'ListView' is defined to display a list of users. The 'app:items' attribute is bound to the 'userList' property of the ViewModel. This requires a custom 'BindingAdapter' to convert the 'userList' data into a format that the ListView can display. The ListView's layout attributes are set to make it expand to fill the remaining space in the layout.

Step 4: Create a Binding Adapter:

Create a binding adapter to bind the list of users to the ListView.

Code BindingAdapters.java

```
package com.example.livedatabindingexample;

import android.widget.ArrayAdapter;
import android.widget.ListView;

import androidx.databinding.BindingAdapter;

import java.util.List;

public class BindingAdapters {
    @BindingAdapter("items")
    public static void setItems(ListView listView, List<String> items) {
       ArrayAdapter<String> adapter = new ArrayAdapter<>(listView.getContext(),
                                    android.R.layout.simple_list_item_1, items);
        listView.setAdapter(adapter);
    }
}
```

Explanation

1. Purpose of '@BindingAdapter':

   The '@BindingAdapter' annotation is used to create custom binding logic in data binding. It allows the definition of how specific attributes should be set or modified when bound to data. In this case, it is used to define how a 'ListView' should be populated with a list of items.

2. **Creating the 'setItems' Method:**

   The 'setItems' method is a static method annotated with '@BindingAdapter("items")'. This means that whenever the 'items' attribute is used in a layout file, this method will be called to handle the binding.

3. **Parameters:**

   The method takes two parameters:

   - 'ListView listView': The 'ListView' instance where the data will be set.
   - 'List<String> items': The list of strings to be displayed in the 'ListView'.

4. Setting Up the Adapter:

Inside the 'setItems' method, an 'ArrayAdapter<String>' is created. The 'ArrayAdapter' is a common adapter used to convert an array of strings into views (in this case, text views) that are displayed in the 'ListView'.

The 'ArrayAdapter' is initialized with:

- 'listView.getContext()': The context of the 'ListView'.
- 'android.R.layout.simple_list_item_1': A predefined layout resource for a simple list item (just a single 'TextView').

5. **Binding the Adapter to the ListView:**

   Finally, the 'setAdapter' method of the 'ListView' is called to set the 'ArrayAdapter' created in the previous step. This binds the list of strings to the 'ListView', causing the 'ListView' to display the items.

Step 5: Create the ViewModel

Create a ViewModel to manage the data for the UI.

Code UserViewModel.java

```
package com.example.livedatabindingexample;

import androidx.databinding.BaseObservable;
import androidx.databinding.Bindable;

import java.util.ArrayList;
import java.util.List;

public class UserViewModel extends BaseObservable {
    private String name = "";
    private String phone = "";
    private List<String> userList = new ArrayList<>();

    @Bindable
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
        notifyPropertyChanged(BR.name);
        updateUserList();
    }

    @Bindable
    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
        notifyPropertyChanged(BR.phone);
        updateUserList();
    }
```

```
    @Bindable
    public List<String> getUserList() {
        return userList;
    }

    private void updateUserList() {
        userList.clear();
        if (!name.isEmpty() && !phone.isEmpty()) {
            userList.add(name + " - " + phone);
        }
        notifyPropertyChanged(BR.userList);
    }
}
```

Explanation

1. **Extending 'BaseObservable':** The 'UserViewModel' class extends 'BaseObservable', which is a key component in the data binding framework. This extension allows the class to notify the data binding system about changes in its properties, enabling the UI to react to data changes dynamically and automatically.
2. **Observable Properties:** The class defines observable properties: 'name', 'phone', and 'userList'. These properties are annotated with '@Bindable', making them bindable to the UI components. The '@Bindable' annotation signals the data binding library to generate a corresponding entry in the 'BR' class, facilitating property change notifications.
3. **Updating 'userList':** When the 'name' or 'phone' properties are updated, the 'updateUserList' method is called. This method updates the 'userList' by clearing it and adding a new entry that combines 'name' and 'phone', provided neither is empty. This ensures that the list displayed in the UI reflects the most recent data entered by the user.
4. **Property Change Notifications:** The 'notifyPropertyChanged(BR.name)' and 'notifyPropertyChanged(BR.phone)' methods are used to inform the data binding system that the respective properties have changed. This triggers the data binding library to update any UI components bound to these properties, ensuring the UI stays in sync with the underlying data model without requiring explicit updates in the activity or fragment code.

Step 6: Update MainActivity:

Update 'MainActivity' to set up data binding.

Code MainActivity.java

```
package com.example.livedatabindingexample;

import android.os.Bundle;

import androidx.appcompat.app.AppCompatActivity;
import androidx.databinding.DataBindingUtil;
```

```
import com.example.livedatabindingexample.databinding.ActivityMainBinding;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ActivityMainBinding binding = DataBindingUtil.setContentView(this,
                                                    R.layout.activity_main);
        UserViewModel viewModel = new UserViewModel();
        binding.setViewModel(viewModel);
        binding.setLifecycleOwner(this);
    }
}
```

Explanation

1. **Initializing Data Binding and Setting the ViewModel:**

   In 'MainActivity', data binding is initialized with 'DataBindingUtil.setContentView(this, R.layout.activity_main);'. This replaces the traditional 'setContentView' and returns a binding object that is tied to the layout. The 'UserViewModel' is instantiated and set as the binding's view model with 'binding.setViewModel(viewModel);', linking the layout directly to the view model's data.

2. **Setting Up the Binding:**

   The method 'DataBindingUtil.setContentView' not only inflates the layout but also initializes the binding object. This object allows direct access to the UI components and enables data binding expressions in the XML layout to work. It effectively bridges the layout and the view model, allowing UI components to automatically reflect changes in the view model's data.

3. **Making Binding Lifecycle-Aware:**

   By calling 'binding.setLifecycleOwner(this);', the binding becomes aware of the lifecycle of the activity. This ensures that the data binding framework can handle lifecycle events (like activity pause, resume, or destroy) seamlessly. It allows the binding to observe LiveData properties and update the UI components accordingly, ensuring consistent and up-to-date UI throughout the lifecycle changes.

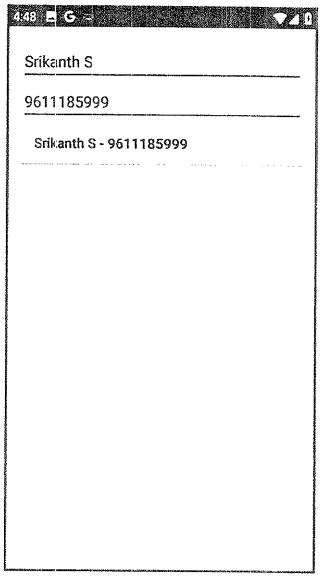
Step 7: Clean and Rebuild the Project:

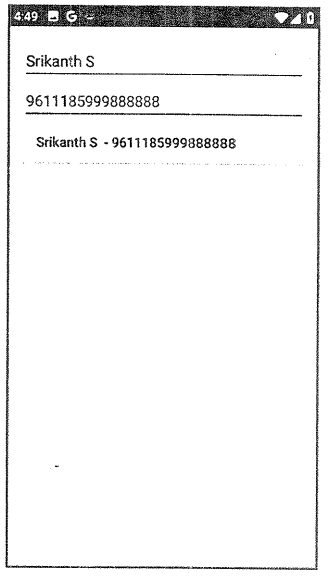
1. **Clean the Project:** Go to 'Build' -> 'Clean Project'.
2. **Rebuild the Project:** Go to 'Build' -> 'Rebuild Project'.

Output

When the application runs:

- The user can enter a name and phone number in the EditText fields.
- As the user types, the name and phone number are displayed in the ListView below.
- This demonstrates live data binding, where changes in the ViewModel automatically update the UI.





8.5 Review Questions

Section - A

1. What is SQLite?
2. List two platforms where SQLite is commonly used.
3. Describe two key features of SQLite.
4. Why is SQLite considered a good choice for mobile apps?
5. Provide an example of a CREATE TABLE statement in SQLite.
6. Describe the purpose of the PRIMARY KEY constraint.
7. What is the role of the SQLiteOpenHelper class in Android?
8. Describe the purpose of the onCreate method in SQLiteOpenHelper.
9. What does the getWritableDatabase method do?
10. How is data inserted into a SQLite database in Android?
11. What is the role of the ContentValues class in database operations?
12. What is data binding in Android?
13. How does data binding improve performance in Android applications?
14. Describe the purpose of the @Bindable annotation.
15. What is a BindingAdapter in Android data binding?
16. Explain the role of the ViewModel in data binding.

Section - B

1. Explain the features of SQLite and its benefits.
2. Explain the common applications of SQLite databases.
3. Explain DDL Statements in SQLite.
4. Explain DML Statements in SQLite.
5. Explain Query Statements in SQLite.
6. Explain the use of constraints in SQLite with examples.
7. Describe the process of creating and managing a database schema in SQLite, including creating tables and defining constraints.
8. Explain the steps to create a database in an Android application.
9. How can data be updated in a SQLite database? Provide an example.
10. Describe the process of deleting data from a SQLite database.
11. Explain how to retrieve data from a SQLite database using the query method.
12. Provide a step-by-step guide to implementing a database helper class in Android.
13. Explain and compare the traditional approach of setting up UI components with data binding.
14. Provide an example of binding an EditText to a ViewModel property.
15. Discuss the lifecycle management in data binding and its importance.
16. Explain how to create a custom BindingAdapter and its use cases.

Section - C

1. Write the advantages and disadvantages of using SQLite in Android applications.
2. Discuss the lifecycle of a SQLite database in an Android application.
3. Explain the complete workflow of creating, reading, updating, and deleting data in a SQLite database with code examples.
4. Describe the implementation of a contact management app using SQLite in Android.
5. Describe the step-by-step implementation of data binding in an Android application.
6. Discuss the benefits and challenges of using data binding in complex Android applications.
7. Provide a detailed example of implementing live data binding in an Android application.

UNIT - IV

CHAPTER 9

CONTENT PROVIDERS

CONTENTS

- Content Providers
- Features of Content Provider
- Advantages and Disadvantages of Content Provider
- Content Provider URI
- Content Resolver
- Content Values
- Cursor
- Permissions in Content Providers
- Using Android Built-in Content Providers (Content Provider Demo Application)
- Creating Your Own Content Provider (Custom Content Provider)
- Using Custom Content Provider
- Review Questions

9.1 Content Providers

Content providers are one of the fundamental building blocks of Android applications. They provide a standardized way to share data across different applications. They serve as a crucial building block by encapsulating data and providing a secure and controlled interface for accessing and manipulating that data. Unlike other data storage methods like shared preferences, files, or SQLite databases, content providers excel in facilitating data sharing and inter-process communication between different applications on the Android platform.

In simple terms, a content provider in Android is like a gatekeeper for data. It manages access to a specific set of data and allows different parts of an app or even different apps to securely interact with that data. It acts as a bridge that controls how data is shared and accessed, ensuring that only authorized entities can read or modify the information.

What is Content Provider?

A content provider in Android is a component that manages access to a structured set of data. It serves as an intermediary between the data source and different applications that need to access or modify that data. The content provider provides a standardized and secure way for apps to share data with each other. It encapsulates the data and exposes it through a consistent interface.

Example Understanding Content Provider

Imagine a library system where students want to access information about books. In this scenario:

1. **Library (Data Storage):**
   - The central repository of books (data storage).
   - The library stores all the books, similar to how data is stored in a database, files, or other formats in an application.
2. **Librarian (Content Provider):**
   - The intermediary who provides access to the books (content provider).
   - The librarian knows where each book is located and how to access it.
   - The librarian acts as the intermediary between the students and the books, providing a standardized way to request and retrieve books.
   - In the Android world, the content provider knows how to access and manage data. It provides a standardized interface for querying, inserting, updating, and deleting data.
3. **Students (Applications):**
   - The applications that need information about the books (applications accessing data).
   - The students approach the librarian with their requests. They might ask for all books by a certain author, a specific book by title, or a list of available genres.
   - Similarly, applications query the content provider using URIs to get the required data. They might request all contacts, a specific contact, or data filtered by certain criteria.

When purchasing a new mobile device running on the Android operating system, such as Samsung, OnePlus, Oppo, or Vivo, it comes preloaded with essential applications and features like contacts, maps, browser, settings, calling features, messaging, and a gallery for images. These built-in functionalities provide users with a seamless experience right out of the box. Additionally, when users download popular apps like Google Pay, Ola, or Uber, these applications leverage the device's existing features through content providers. For example, Google Pay utilizes the contacts feature to facilitate transactions, while Ola and Uber rely on the device's map functionality to provide location-based services for navigation and ride-hailing.

Examples | **Real World Scenarios**

Content providers are essential for sharing data between applications in Android. Let us understand some scenarios where content providers play a crucial role:

1. **Contact Sharing**

   Suppose a mobile payment app similar to Google Pay or PhonePe is being built. This app needs access to the device's contact list to enable users to easily connect with their contacts for sending or receiving money. In this scenario, the content provider is the component that helps the app to access the contacts stored on the mobile device. By querying the Contacts content provider, the app can retrieve the necessary contact information securely and efficiently.

2. **Media Access**

   Consider developing a media player or an image editor. This app needs to access and share audio, video, and image files stored on the device. For example, a media player needs to list all the songs available on the device, while an image editor might need to access photos stored in the gallery. The MediaStore content provider helps the app to access and manage these media files.

3. **Google Maps Integration**

   Imagine building a travel app that integrates with Google Maps to provide location-based services. This app needs to access geographical data, such as places visited or marked locations, to offer features like trip planning or location recommendations. The Google Maps content provider enables the app to query and manage geographical data, facilitating seamless integration with map services and enhancing the overall user experience.

4. **Call Logs**

   Suppose a call management or analytics app is being developed to provide call-related features and insights, such as tracking call durations or identifying frequent callers. This app needs access to the call logs stored on the device. The CallLog content provider enables the app to query and manage call log data, facilitating the retrieval of call details like missed calls, call durations, and contact information associated with the calls.

In each of these scenarios, content providers act as intermediaries that manage access to structured data stored on the device. By using content providers, applications can securely and efficiently access and manipulate data to ensure a consistent and standardized approach to data sharing and management across different apps.

9.2 Features of Content Provider

Content providers in Android offer several key features that make them essential for data sharing and management between applications. Some of the key features of content providers include:

1. **Data Encapsulation:** Content providers encapsulate data and provide a structured interface for interacting with it. This abstraction layer shields the underlying data implementation details and enforces controlled access to the data.
2. **Data Sharing:** Content providers enable sharing of data between different parts of an application or even across multiple applications. They serve as a centralized mechanism for accessing and manipulating shared data.
3. **Data Security:** Content providers enforce data security by regulating access to shared data through permissions. They ensure that only authorized applications can access specific data sets to enhance data privacy and security.
4. **CRUD Operations :** Content providers support Create, Read, Update, and Delete (CRUD) operations to enable comprehensive data management. These operations allow applications to perform necessary modifications to the data stored in the content provider. They ensure that data operations are performed in a consistent and reliable manner.
5. **Content URI Handling:** Content providers use content URIs to uniquely identify data resources. By using content URIs, applications can specify the data they want to access or manipulate.
6. **Query Support:** Content providers support querying data using the ContentResolver class to enable applications to retrieve specific data sets based on criteria such as filters, sorting, and projections. This querying mechanism enhances data retrieval efficiency and flexibility.
7. **Data Permissions:** Content providers allow developers to define permissions for accessing specific data sets. By setting appropriate permissions, developers can control which applications have read or write access to particular data to ensure data security and privacy.
8. **Content Provider Registry:** Android maintains a registry of content providers for applications to discover and access available content providers. This centralized registry simplifies the process of locating and interacting with shared data resources.
9. **Inter-Process Communication:** Content providers facilitate inter-process communication by allowing data sharing between different applications running in separate processes. This capability enables seamless data exchange and collaboration between diverse applications.
10. **Content Provider Lifecycle:** Content providers follow a lifecycle that includes methods for initialization, data query, data modification, and shutdown. This lifecycle management ensures proper handling of data operations and resource cleanup.
11. **Standardized Interface :** Content providers provide a standardized programming interface for accessing and manipulating data. This consistency simplifies the development process and ensures compatibility across different applications and devices.
12. **Support for Various Data Sources :** Content providers can manage data stored in various formats, including databases, files, or even data retrieved from a network. This flexibility allows developers to use content providers for a wide range of data storage scenarios.
13. **Batch Operations :** Content providers support batch operations by allowing multiple changes to be made in a single transaction. This feature improves efficiency and consistency when performing bulk data operations.

Example | **Role of Content Providers in Managing Contact Data in Android Applications**

In managing contact data in an Android application, a content provider acts as a central component for handling interactions with contact information. By defining a content URI, the content provider establishes a standardized and secure way for applications to query, insert, update, and delete contact records. It ensures that only authorized applications can access and modify contact information through permissions.

When an application queries the contact content provider, it uses the content URI to specify the data fields to retrieve. The content provider processes the query, fetches the requested contact data, and returns the results to the application. This querying mechanism ensures efficient retrieval and data integrity.

Content providers also facilitate seamless data management by allowing applications to add new contacts, update existing ones, and delete contacts as needed. Leveraging content URIs, data encapsulation, and query support, developers can create robust contact management systems that promote data reusability, security, and interoperability within the Android ecosystem.

9.3 Advantages and Disadvantages of Content Provider

Advantages of Content Provider

1. **Data Sharing:** Content providers facilitate seamless data sharing between different applications for efficient collaboration and interoperability.
2. **Data Security:** By enforcing permissions and access controls, content providers enhance data security and privacy to ensure that sensitive information is protected from unauthorized access.
3. **Data Abstraction:** Content providers encapsulate data to provide a secure and consistent interface for data access and manipulation. This abstraction simplifies the interaction with complex data structures.
4. **Data Integrity:** Content providers help maintain data integrity by providing methods for controlled data operations and reduces the risk of data corruption or inconsistencies.
5. **Inter-Process Communication:** Content providers enable inter-process communication between applications running in separate processes to enhance functionality and user experience.
6. **Standardization:** Content providers follow a standardized API to promote consistency and uniformity in data management practices across applications.
7. **Flexibility :** Content providers can manage data stored in various formats, including databases, files, or network-based data. This flexibility allows developers to choose the most appropriate storage mechanism for their application's needs.

Disadvantages of Content Provider

1. **Complexity:** Implementing content providers can be complex and time-consuming, especially for applications with simple data access requirements. The need to define URIs, manage permissions, and handle CRUD operations adds to the development effort.
2. **Performance Overhead:** The abstraction and encapsulation provided by content providers can introduce performance overhead. Frequent data access through content providers may be slower compared to direct access methods.
3. **Security Risks :** Misconfigured permissions or improper handling of content providers can expose sensitive data to unauthorized applications. Ensuring proper security measures and rigorous testing are essential to mitigate these risks.

4. **Limited Functionality :** Content providers may not support all the advanced features of the underlying data storage mechanisms. For example, complex database transactions or specific file operations may not be fully supported.
4. **Dependency on Content Providers :** Applications relying heavily on content providers may face issues if the content provider changes or becomes unavailable. This dependency can impact the application's stability and functionality.
5. **Testing Complexity:** Testing and debugging content provider interactions can be challenging due to the need to simulate various data access scenarios and handle asynchronous operations. Ensuring comprehensive test coverage requires additional effort.

9.4 Content Provider URI

A Uniform Resource Identifier (URI) is a string of characters used to uniquely identify a resource. Content Provider URI play a crucial role in identifying and accessing data stored by the provider. They act as the primary means for applications to interact with the data to ensure a standardized and consistent approach to data management.

By using Content Provider URIs, developers can interact with shared data across applications, access system resources, and perform CRUD (Create, Read, Update, Delete) operations on structured data in a secure and standardized manner within the Android framework.

What is Content Provider URI ?

A Content Provider URI (Uniform Resource Identifier) in Android is a unique identifier that specifies the data source or location within a content provider. It follows a specific format that allows applications to access and manipulate structured data in a consistent and secure manner.

Structure of a Content Provider URI

The structure of a content provider URI is typically as follows:

```
content://<authority>/<data_path>/<id>
```

Each part of the URI serves a specific purpose:

- **content://** : This is the standard prefix for content provider URIs in Android.
- **<authority>** : The authority specifies the name of the content provider. It is often the package name of the application hosting the content provider. This part of the URI ensures that the request is routed to the correct content provider.
- **<data_path>** : The data path indicates the type of data being accessed. It specifies the subset or table of data within the content provider. For example, it could be "contacts" for a contacts content provider or "images" for a media content provider.
- **<id>** : The id is an optional specifier for a particular data item. It uniquely identifies a specific record within the data set. If omitted, the URI refers to the entire data set or table.

Example Content Provider URI

Example 1: Accessing Contacts:

URI: 'content://com.android.contacts/contacts'

content: //: Standard prefix for content provider URIs.

com.android.contacts: Authority specifying the contacts content provider.

contacts: Data path indicating access to the contacts table.

(No id): Refers to the entire contacts table.

URI: 'content://com.android.contacts/contacts/1'

content://: Standard prefix for content provider URIs.

com.android.contacts: Authority specifying the contacts content provider.

contacts: Data path indicating access to the contacts table.

1: ID specifying access to the contact with ID 1.

Example 2: Accessing Images in Media Store:

URI: 'content://media/external/images/media'

content://: Standard prefix for content provider URIs.

media: Authority specifying the media content provider.

external/images/media: Data path indicating access to the external images table in the media store.

(No id): Refers to the entire images table on the external storage.

URI: 'content://media/external/images/media/42'

content://: Standard prefix for content provider URIs.

media: Authority specifying the media content provider.

external/images/media: Data path indicating access to the external images table in the media store.

42: ID specifying access to the image with ID 42.

Built-in Content Providers

Android comes with several built-in content providers that facilitate the sharing and management of common data types across applications. These built-in providers offer standardized APIs for accessing and manipulating data such as contacts, calendar events, media files, and more. By using these content providers, we can integrate core functionalities into our applications for efficient data retrieval and management. These content providers not only simplify data access but also ensure data consistency and security through well-defined permission mechanisms.

Some of the available content providers in Android are listed in the below table.

Content Provider	Description	Authority	URI Examples	Data Types
Contacts Provider	Manages the user's contact information	'com.android.contacts'	content://com.android.contacts/contacts	Names, phone numbers, emails, addresses

Calendar Provider	Manages events, reminders, and calendar data	'com.android.calendar'	'content://com.android.calendar/events' 'content://com.android.calendar/reminders'	Event details, reminders, calendar metadata
Call Log Provider	Stores call history details	'call_log'	'content://call_log/calls' 'content://call_log/calls/1'	Incoming, outgoing, and missed calls
Media Store Provider	Manages audio, video, and image files	'media'	'content://media/external/images/media' 'content://media/internal/audio/media'	Images, audio files, video files, metadata
Browser Provider	Stores browser-related data such as bookmarks and history	'com.android.browser'	content://com.android.browser/bookmarks	Bookmarks, browser history, search history
Settings Provider	Manages system and application settings	'settings'	'content://settings/system' 'content://settings/secure'	System settings, secure settings
User Dictionary Provider	Stores words added to the user's custom dictionary	'user_dictionary'	'content://user_dictionary/words'	Words, frequency of usage, locale information
Downloads Provider	Manages the files downloaded by the user	'downloads'	'content://downloads/my_downloads'	File paths, download statuses, sizes
Telephony Provider	Manages telephony-related data such as SMS and MMS	'sms', 'mms', 'mms-sms'	'content://sms/inbox' 'content://mms' content://mms-sms/conversati	SMS, MMS, message threads
Documents Provider	Manages documents and other files	'com.android.externalstorage.documents'	'content://com.android.externalstorage.documents/document'	Document files, metadata, file paths
Dictionary Provider	Provides access to the dictionary and spell checker data	'user_dictionary'	'content://user_dictionary/words'	Words, frequency, locale
Voicemail Provider	Manages voicemail messages	'voicemail'	'content://voicemail/voicemails'	Voicemail metadata, audio data
App Usage Stats Provider	Provides access to the usage statistics of apps	'com.android.settings.applications'	'content://com.android.settings.applications.usage'	App usage stats, usage history
Music Provider	Manages music files and playlists	'com.android.music'	'content://com.android.music/audio'	Music files, playlists, metadata

9.5 Content Resolver

The Content Resolver in Android provides an abstract layer for accessing content providers. It acts as a client-side interface that helps applications interact with content providers to perform CRUD (Create, Read, Update, Delete) operations on data. The Content Resolver simplifies data access and ensures that the application communicates with the content provider in a consistent and secure manner.

What is Content Resolver ?

The Content Resolver in Android provides an abstract layer for accessing content providers. The ContentResolver class in Android serves as the primary interface for interacting with content providers. Its main purpose is to facilitate communication between Android applications and content providers.

Features of Content Resolver

1. **Abstract Interface for Data Access:** The Content Resolver provides a unified API for accessing different types of content providers for consistent data access across various data sources.
2. **CRUD Operations:** The Content Resolver supports all basic database operations including Create, Read, Update, and Delete. It ensures that these operations are performed securely and efficiently. ContentResolver provides methods like insert(), update(), and delete() for data manipulation operations.
3. **URI-Based Data Access:** Data access through the Content Resolver is based on URIs, which uniquely identify the data within the content provider. This approach allows for precise and flexible querying of data.
4. **Querying Data:** The query() method retrieves data from content providers and returns a Cursor object for result set traversal.
5. **Asynchronous Operations:** The Content Resolver supports asynchronous data operations, which help in preventing blocking the main thread and improving application performance.
6. **Inter-Application Data Sharing:** It enables secure and controlled sharing of data between different applications by enforcing permission checks and URI validation.
7. **Observer Notifications:** Applications can register observers with the Content Resolver to get notified about changes in the data they are interested in. This feature is useful for keeping the application UI in sync with the underlying data.

Methods of Content Resolver

The Content Resolver provides several methods to interact with content providers.

Method	Description
insert()	Inserts a new row of data into the content provider.
query()	Executes a query on a content provider and returns a Cursor object containing the result set.
update()	Updates existing rows in the content provider that match the selection criteria.

delete()	Deletes rows from the content provider that match the selection criteria.
getType()	Returns the MIME type of the data at the given URI.
bulkInsert()	Performs a batch insert of multiple rows into the content provider.
registerContentObserver()	Registers a ContentObserver to listen for changes to the data at a specific URI.
unregisterContentObserver()	Unregisters a previously registered ContentObserver.

Example Usage of Content Resolver Methods

1. **Insert a New Contact:**

   The 'insert' method adds a new row to the content provider. It requires a URI representing the target table and a 'ContentValues' object containing the data to be inserted. The method returns the URI of the newly created row.

Code

```
ContentResolver resolver = getContentResolver();
Uri uri = Uri.parse("content://com.android.contacts/contacts");
ContentValues values = new ContentValues();
values.put("display_name", "Srikanth");
Uri newUri = resolver.insert(uri, values);
```

Explanation

- 'ContentResolver resolver = getContentResolver();': Obtains the content resolver instance.
- 'Uri uri = Uri.parse("content://com.android.contacts/contacts");': Defines the URI for the contacts table.
- 'ContentValues values = new ContentValues();': Creates a 'ContentValues' object to hold the new contact data.
- 'values.put("display_name", "Srikanth");': Adds the contact name to the 'ContentValues' object.
- 'Uri newUri = resolver.insert(uri, values);': Inserts the new contact and returns the URI of the newly created row.

2. **Query Contacts:**

   The 'query' method retrieves data from the content provider. It requires a URI representing the target table and optional parameters for projection, selection, selection arguments, and sort order. The method returns a 'Cursor' object that allows access to the query results.

Code

```
ContentResolver resolver = getContentResolver();
Uri uri = Uri.parse("content://com.android.contacts/contacts");
Cursor cursor = resolver.query(uri, null, null, null, null);
while (cursor.moveToNext()) {
    String name = cursor.getString(cursor.getColumnIndex("display_name"));
    // Process the retrieved data
}
cursor.close();
```

Explanation

- 'ContentResolver resolver = getContentResolver();': Obtains the content resolver instance.
- 'Uri uri = Uri.parse("content://com.android.contacts/contacts");': Defines the URI for the contacts table.
- 'Cursor cursor = resolver.query(uri, null, null, null, null);': Queries the contacts table and returns a cursor for the results.
- 'while (cursor.moveToNext()) { ... }': Iterates over the query results.
- 'String name = cursor.getString(cursor.getColumnIndex("display_name"));': Retrieves the contact name from the current row.
- 'cursor.close();': Closes the cursor to release resources.

3. **Update a Contact:**

The 'update' method modifies existing rows in the content provider. It requires a URI representing the target table, a 'ContentValues' object containing the updated data, and optional parameters for selection and selection arguments. The method returns the number of rows updated.

Code

```
ContentValues values = new ContentValues();
values.put("display_name", "Jane Doe");
int rowsUpdated = resolver.update(uri, values, "contact_id=?", new String[]{"1"});
```

Explanation

- 'ContentValues values = new ContentValues();': Creates a 'ContentValues' object to hold the updated contact data.
- 'values.put("display_name", "Srikanth.S");': Updates the contact name in the 'ContentValues' object.
- 'int rowsUpdated = resolver.update(uri, values, "contact_id=?", new String[]{"1"});': Updates the contact with ID "1" and returns the number of rows updated.

4. **Delete a Contact:**

The 'delete' method removes rows from the content provider. It requires a URI representing the target table and optional parameters for selection and selection arguments. The method returns the number of rows deleted.

Code

```
int rowsDeleted = resolver.delete(uri, "contact_id=?", new String[]{"1"});
```

Explanation

- 'int rowsDeleted = resolver.delete(uri, "contact_id=?", new String[]{"1"});': Deletes the contact with ID "1" and returns the number of rows deleted.

5. **Get MIME Type of Data:**

The 'getType' method returns the MIME type of the data at the given URI. This is useful for determining the type of data being accessed.

Code

```
String type = resolver.getType(uri);
```

Explanation

- 'String type = resolver.getType(uri);': Retrieves the MIME type of the data at the specified URI.

6. **Bulk Insert:**

The `bulkInsert` method performs a batch insert of multiple rows into the content provider. It requires a URI representing the target table and an array of `ContentValues` objects containing the data to be inserted. The method returns the number of rows inserted.

Code

```
ContentValues[] bulkValues = new ContentValues[2];
bulkValues[0] = new ContentValues();
bulkValues[0].put("display_name", "Mary");
bulkValues[1] = new ContentValues();
bulkValues[1].put("display_name", "Nirmala");
int rowsInserted = resolver.bulkInsert(uri, bulkValues);
```

Explanation

- 'ContentValues[] bulkValues = new ContentValues[2];': Creates an array to hold multiple 'ContentValues' objects.
- 'bulkValues[0] = new ContentValues(); bulkValues[0].put("display_name", "Mary");': Adds the first contact.
- 'bulkValues[1] = new ContentValues(); bulkValues[1].put("display_name", "Nirmala");': Adds the second contact.
- 'int rowsInserted = resolver.bulkInsert(uri, bulkValues);': Inserts multiple contacts and returns the number of rows inserted.

7. **Register a Content Observer:**

The 'registerContentObserver' method registers an observer to monitor changes to the content provider. It requires a URI representing the target table, a boolean indicating whether to observe descendants, and the 'ContentObserver' instance.

Code

```
ContentObserver observer = new ContentObserver(new Handler()) {
    @Override
    public void onChange(boolean selfChange, Uri uri) {
        // Handle the change notification
    }
};
resolver.registerContentObserver(uri, true, observer);
```

Explanation
• 'ContentObserver observer = new ContentObserver(new Handler()) { ... };': Creates a 'ContentObserver' to handle change notifications. • 'resolver.registerContentObserver(uri, true, observer);': Registers the observer to monitor changes to the contacts table.

8. **Unregister a Content Observer:**

   The 'unregisterContentObserver' method removes a previously registered content observer. It requires the 'ContentObserver' instance to be unregistered.

Code

```
resolver.unregisterContentObserver(observer);
```

Explanation
• 'resolver.unregisterContentObserver(observer);': Unregisters the observer to stop receiving change notifications.

9.6 Content Values

'ContentValues' is a key-value pair data structure in Android used for inserting and updating rows in a database or a content provider. It is a class in android specifically designed for handling the data types that are compatible with SQLite databases and content providers. Each key is a column name and each value is the data to be stored in that column.

What is Content Values ?

ContentValues is a class in Android that provides a key-value store for various types of data that can be inserted or updated in a ContentProvider. It is commonly used when working with databases in Android to define the column-value pairs for database operations like insert or update.

Features and Importance of ContentValues

- Simplified Data Handling: 'ContentValues' makes it easy to prepare data for database operations by allowing developers to insert and update data in a structured manner.
- **Type Safety:** It provides methods to put various data types (integers, strings, booleans, etc.), ensuring type safety.
- **Compatibility:** It is designed to work seamlessly with SQLite databases and content providers,. It is a standard way to handle data in Android.
- **Key-Value Pair:** It stores data as key-value pairs, where the key is the column name and the value is the data to be stored in that column.

Methods of ContentValues

Method	Description	Example Code
put(String key, String value)	nserts a 'String' value into the 'ContentValues' object.	`values.put("name", "Srikanth");`
put(String key, Integer value)	Inserts an 'Integer' value into the 'ContentValues' object.	`values.put("age", 30);`
put(String key, Boolean value)	Inserts a 'Boolean' value into the 'ContentValues' object.	`values.put("is_active", true);`
put(String key, Float value)	Inserts a 'Float' value into the 'ContentValues' object.	`values.put("height", 5.9f);`
put(String key, Long value)	Inserts a 'Long' value into the 'ContentValues' object.	`values.put("timestamp", System.currentTimeMillis());`
put(String key, byte[] value)	Inserts a 'byte[]' value into the 'ContentValues' object.	`values.put("image", imageBytes);`

Examples of Using ContentValues

1. **Example: Inserting a New Contact:**

   To insert a new contact into the contacts content provider, 'ContentValues' is used to prepare the data:

Code

```
ContentValues values = new ContentValues();
values.put("display_name", "Srikanth");
values.put("phone_number", "9611185999");
values.put("email", "skyward.publishers@gmail.com");

Uri uri = Uri.parse("content://com.android.contacts/contacts");
ContentResolver resolver = getContentResolver();
Uri newUri = resolver.insert(uri, values);
```

Explanation

- 'ContentValues values = new ContentValues();': Creates a new instance of 'ContentValues'.
- 'values.put("display_name", "Srikanth");': Adds the contact's display name.
- 'values.put("phone_number", "9611185999");': Adds the contact's phone number.
- 'values.put("email", "skyward.publishers@gmail.com");': Adds the contact's email.
- 'Uri uri = Uri.parse("content://com.android.contacts/contacts");': Defines the URI for the contacts table.
- 'ContentResolver resolver = getContentResolver();': Gets the content resolver instance.
- 'Uri newUri = resolver.insert(uri, values);': Inserts the new contact and returns the URI of the newly created row.

2. **Example: Updating an Existing Contact:**

To update an existing contact's phone number in the contacts content provider:

Code

```
ContentValues values = new ContentValues();
values.put("phone_number", "8026603535");

Uri uri = Uri.parse("content://com.android.contacts/contacts/1");
ContentResolver resolver = getContentResolver();
int rowsUpdated = resolver.update(uri, values, null, null);
```

Explanation

- 'ContentValues values = new ContentValues();': Creates a new instance of 'ContentValues'.
- 'values.put("phone_number", "8026603535");': Updates the contact's phone number.
- 'Uri uri = Uri.parse("content://com.android.contacts/contacts/1");': Defines the URI for the specific contact with ID 1.
- 'ContentResolver resolver = getContentResolver();': Gets the content resolver instance.
- 'int rowsUpdated = resolver.update(uri, values, null, null);': Updates the contact and returns the number of rows updated.

3. **Example: Using ContentValues in a Database Operation:**

To insert a new record into a SQLite database table:

Code

```
ContentValues values = new ContentValues();
values.put("name", "Mary");
values.put("age", 25);
values.put("email", "mary.margeret@gmail.com");

SQLiteDatabase db = dbHelper.getWritableDatabase();
long newRowId = db.insert("users", null, values);
```

Explanation

- 'ContentValues values = new ContentValues();': Creates a new instance of 'ContentValues'.
- 'values.put("name", "Mary");': Adds the user's name.
- 'values.put("age", 25);': Adds the user's age.
- 'values.put("email", "mary.margeret@gmail.com");': Adds the user's email.
- 'SQLiteDatabase db = dbHelper.getWritableDatabase();': Gets a writable database instance.
- 'long newRowId = db.insert("users", null, values);': Inserts the new user record and returns the new row ID.

9.7 Cursor

A Cursor in Android is an interface that provides read-only access to the result set returned by a database query. It allows to iterate over the rows of the result set and retrieve data from each row. Cursors are commonly used in Android when working with databases through ContentProviders or SQLite databases. A Cursor does not directly support write operations, but it can be used in conjunction with ContentResolver or SQLiteDatabase to perform write operations such as inserts, updates, and deletes.

What is a Cursor ?

A Cursor in Android is an interface that provides read-only access to the result set returned by a database query. It represents the rows of data resulting from a database query and allows for traversal and retrieval of information from the result set. Cursors are commonly used in Android when interacting with SQLite databases, ContentProviders, and ContentResolvers.

Features and Importance of Cursor

- **Efficient Data Access:** Cursors allow efficient traversal and manipulation of large result sets without loading all data into memory at once.
- **Random Access:** Provides methods for moving to specific rows in the result set to access data randomly.
- **Iteration:** Cursors can move back and forth within the result set using methods like moveToFirst(), moveToNext(), moveToPosition(), etc.
- **Column Access:** Data in a Cursor can be accessed by referencing the column index or column name.
- **Data Retrieval:** Cursors offer methods like getInt(), getString(), getLong(), getBlob(), etc., to retrieve data from the current row based on the column index or name.
- **Lifecycle Management:** Proper management of Cursors is essential to release system resources. Cursors should be closed when they are no longer needed to avoid memory leaks.

Methods of Cursor

Method	Description	Example Code
moveToFirst()	Moves the cursor to the first row in the result set.	`if (cursor.moveToFirst()) { ... }`
moveToLast()	Moves the cursor to the last row in the result set.	`if (cursor.moveToLast()) { ... }`
moveToNext()	Moves the cursor to the next row in the result set.	`while (cursor.moveToNext()) { ... }`
moveToPrevious()	Moves the cursor to the previous row in the result set.	`if (cursor.moveToPrevious()) { ... }`
moveToPosition(int pos)	Moves the cursor to an absolute position in the result set.	`if (cursor.moveToPosition(position)) { ... }`

getCount()	Returns the number of rows in the result set.	`int count = cursor.getCount();`
getColumnIndex(String columnName)	Returns the zero-based index for the given column name.	`int index = cursor.getColumnIndex("name");`
getString(int columnIndex)	Returns the value of the specified column as a String.	`String name = cursor.getString(cursor.getColumnIndex("name"));`
getInt(int columnIndex)	Returns the value of the specified column as an int.	`int age = cursor.getInt(cursor.getColumnIndex("age"));`
getDouble(int columnIndex)	Returns the value of the specified column as a double.	`double salary = cursor.getDouble(cursor.getColumnIndex("salary"));`
getBlob(int columnIndex)	Returns the value of the specified column as a byte array.	`byte[] image = cursor.getBlob(cursor.getColumnIndex("image"));`
close()	Closes the cursor, releasing all of its resources and making it invalid.	`cursor.close();`

Examples of Using Cursor

1. Example: Querying Contacts:

To query the contacts content provider and retrieve contact names:

Code

```
ContentResolver resolver = getContentResolver();
Uri uri = Uri.parse("content://com.android.contacts/contacts");
Cursor cursor = resolver.query(uri, null, null, null, null);
if (cursor != null) {
    while (cursor.moveToNext()) {
        String name = cursor.getString(cursor.getColumnIndex("display_name"));
        // Process the retrieved data
    }
    cursor.close();
}
```

Explanation

- 'ContentResolver resolver = getContentResolver();': Obtains the content resolver instance.
- 'Uri uri = Uri.parse("content://com.android.contacts/contacts");': Defines the URI for the contacts table.
- 'Cursor cursor = resolver.query(uri, null, null, null, null);': Queries the contacts table and returns a cursor for the results.
- 'while (cursor.moveToNext()) { ... }': Iterates over the query results.
- 'String name = cursor.getString(cursor.getColumnIndex("display_name"));': Retrieves the contact name from the current row.
- 'cursor.close();': Closes the cursor to release resources.

2. **Example: Using Cursor with SQLite Database:**

To query a SQLite database and retrieve user data:

Code

```
SQLiteDatabase db = dbHelper.getReadableDatabase();
Cursor cursor = db.query("users", null, null, null, null, null, null);
if (cursor != null) {
    while (cursor.moveToNext()) {
        String name = cursor.getString(cursor.getColumnIndex("name"));
        int age = cursor.getInt(cursor.getColumnIndex("age"));
        // Process the retrieved data
    }
    cursor.close();
}
```

Explanation

- 'SQLiteDatabase db = dbHelper.getReadableDatabase();': Obtains a readable database instance.
- 'Cursor cursor = db.query("users", null, null, null, null, null, null);': Queries the users table and returns a cursor for the results.
- 'while (cursor.moveToNext()) { ... }': Iterates over the query results.
- 'String name = cursor.getString(cursor.getColumnIndex("name"));': Retrieves the user's name from the current row.
- 'int age = cursor.getInt(cursor.getColumnIndex("age"));': Retrieves the user's age from the current row.
- 'cursor.close();': Closes the cursor to release resources.

9.8 Permissions in Content Providers

Permissions in ContentProviders in Android play a crucial role in ensuring that only authorized applications can access specific data managed by the ContentProvider. Permissions help in maintaining the security and privacy of sensitive information stored in the ContentProvider.

Types of Permissions

1. **Read Permissions:** Allow applications to read data from the content provider.
2. **Write Permissions:** Allow applications to write data to the content provider.

Declaring and Requesting Permissions in Android

Accessing sensitive data or device features in Android requires explicit permission from the user. Permissions must be declared in the 'AndroidManifest.xml' file and, for newer versions of Android, also requested at runtime.

1. **Declaring Permissions in the Manifest File:**

   Permissions are declared in the 'AndroidManifest.xml' file to inform the Android system and the user about the permissions an application needs. This is the first step in gaining access to protected resources.

Example: Declaring Permission to Read Contacts

Step-by-Step Process:

1. Open the AndroidManifest.xml File:
2. Use a '<uses-permission>' element within the '<manifest>' element to declare the required permission.

Code

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name">
        <!-- Your activities and other components go here -->
    </application>

    <!-- Declare permission to read contacts -->
    <uses-permission android:name="android.permission.READ_CONTACTS" />

</manifest>
```

Explanation

- '<uses-permission>': Declares the permission that the application needs. In this example, 'android.permission.READ_CONTACTS' is required to read the device's contacts.

2. **Requesting Permissions at Runtime:**

Starting from Android 6.0 (API level 23), permissions need to be requested at runtime. This involves checking if the permission is already granted and, if not, requesting the user to grant it.

Example: Requesting Permission to Read Contacts at Runtime:

Step-by-Step Process:

1. **Check if Permission is Granted:**

   Before accessing the contacts, check if the 'READ_CONTACTS' permission is already granted.

2. **Request Permission if Not Granted:**

   If the permission is not granted, request it from the user.

3. **Handle the User's Response:**

   Implement the 'onRequestPermissionsResult' method to handle the user's response to the permission request.

Code

```
package com.example.myapp;

import android.content.pm.PackageManager;
import android.os.Bundle;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

    private static final int PERMISSIONS_REQUEST_READ_CONTACTS = 100;

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

        // Check if the READ_CONTACTS permission is already granted
       if (ContextCompat.checkSelfPermission(this, android.Manifest.permission.
                        READ_CONTACTS) != PackageManager.PERMISSION_GRANTED) {
            // Request the READ_CONTACTS permission
            ActivityCompat.requestPermissions(this,
                    new String[]{android.Manifest.permission.READ_CONTACTS},
                    PERMISSIONS_REQUEST_READ_CONTACTS);
        } else {
            // Permission is already granted, access contacts
            accessContacts();
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode,
              @NonNull String[] permissions, @NonNull int[] grantResults) {
      super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (requestCode == PERMISSIONS_REQUEST_READ_CONTACTS) {
            if (grantResults.length > 0 && grantResults[0]
                                    == PackageManager.PERMISSION_GRANTED) {
```

```
                // Permission granted, access contacts
                accessContacts();
            } else {
                // Permission denied, show a message to the user
                Toast.makeText(this, "Permission to read contacts was denied",
                                        Toast.LENGTH_SHORT).show();
            }
        }
    }

    private void accessContacts() {
        // Code to access contacts goes here
    }
}
```

Explanation

- Checking Permission: 'ContextCompat.checkSelfPermission` checks if the `READ_CONTACTS` permission is already granted.
- Requesting Permission: `ActivityCompat.requestPermissions` requests the `READ_CONTACTS` permission if it is not already granted.
- Handling the Response: `onRequestPermissionsResult` handles the user's response. If the permission is granted, the `accessContacts` method is called to access the contacts.

9.9 Using Android Built-in Content Providers (Content Provider Demo Application)

In this section, we'll demonstrate how to create a simple Android application named ContentProviderDemo that uses built-in content providers to load and display contacts and SMS messages. This application will request the necessary permissions from the user, query the appropriate content providers, and display the results in a dialog box. The steps include setting up the project, adding necessary permissions, defining the UI, and implementing the functionality in the main activity.

Step 1. Project Setup

✦ **Create a new project in Android Studio:**

1. Open Android Studio and select 'Create New Project'.
2. Choose 'Empty Views Activity' and click 'Next'.
3. Set the 'Name' to 'ContentProviderDemo'.
4. Set the 'Package Name' to 'com.example.contentproviderdemo'.
5. Set the 'Save location' and 'Language' to 'Java'.
6. Click 'Finish'.

Step 2. Add Permissions to 'AndroidManifest.xml':

Permissions are required to read contacts and SMS messages. These permissions must be declared in the 'AndroidManifest.xml' file.

Code AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.example.contentproviderdemo">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.ContentProviderDemo">
        <activity android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

    <!-- Permissions -->
    <uses-permission android:name="android.permission.READ_CONTACTS" />
    <uses-permission android:name="android.permission.READ_SMS" />
</manifest>
```

Explanation

- '<uses-permission android:name="android.permission.READ_CONTACTS" />' allows the app to read the user's contacts.
- '<uses-permission android:name="android.permission.READ_SMS" />' allows the app to read SMS messages.

Step 3. Define the UI in 'activity_main.xml':

Define the layout of the main activity, including buttons for loading contacts and SMS messages.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
```

```
    <Button
        android:id="@+id/button_load_contacts"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Load Contacts"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp" />

    <Button
        android:id="@+id/button_load_sms"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Load SMSs"
        android:layout_below="@id/button_load_contacts"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="30dp" />

</RelativeLayout>
```

Explanation

- 'RelativeLayout' is used as the parent layout.
- Two buttons are defined: 'button_load_contacts' and 'button_load_sms' for loading contacts and SMS messages, respectively.

Step 4. Implement the Main Activity in 'MainActivity.java':

The main activity handles the user interactions and data loading. It checks for necessary permissions and then loads the data.

Code MainActivity.java

```
package com.example.contentproviderdemo;

import android.Manifest;
import android.annotation.SuppressLint;
import android.app.AlertDialog;
import android.content.ContentResolver;
import android.content.pm.PackageManager;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.ContactsContract;
import android.provider.Telephony;
import android.text.method.ScrollingMovementMethod;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
```

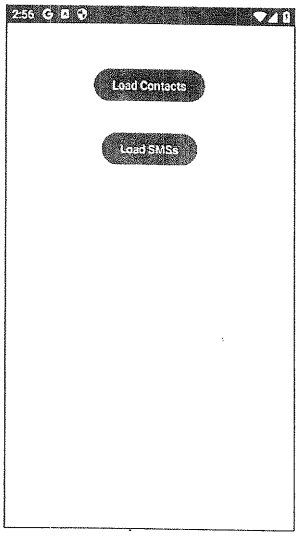
```
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

public class MainActivity extends AppCompatActivity {

    private static final int REQUEST_CODE_READ_CONTACTS = 100;
    private static final int REQUEST_CODE_READ_SMS = 101;
    private TextView textView;

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

        Button buttonLoadContacts = findViewById(R.id.button_load_contacts);
        Button buttonLoadSMS = findViewById(R.id.button_load_sms);

        buttonLoadContacts.setOnClickListener(view -> checkPermissionAndLoad
            (Manifest.permission.READ_CONTACTS, REQUEST_CODE_READ_CONTACTS));
        buttonLoadSMS.setOnClickListener(view -> checkPermissionAndLoad
                    (Manifest.permission.READ_SMS, REQUEST_CODE_READ_SMS));
    }

    private void checkPermissionAndLoad(String permission, int requestCode) {
        if (ContextCompat.checkSelfPermission(this, permission)
                                    != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this, new String[]{permission},
                                                requestCode);
        } else {
            loadData(requestCode);
        }
    }

    private void loadData(int requestCode) {
        switch (requestCode) {
            case REQUEST_CODE_READ_CONTACTS:
                loadContacts();
                break;
            case REQUEST_CODE_READ_SMS:
                loadSMS();
                break;
        }
    }
```

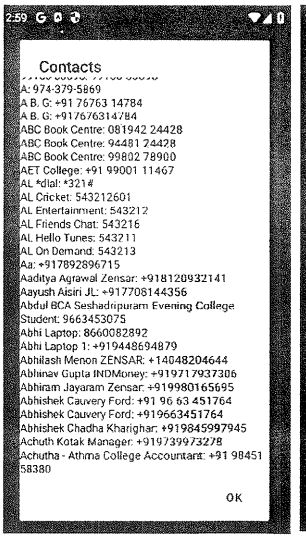
```
private void loadContacts() {
    ContentResolver resolver = getContentResolver();
    Uri uri = ContactsContract.CommonDataKinds.Phone.CONTENT_URI;
    Cursor cursor = resolver.query(uri, null, null, null,
            ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME + " ASC");
    StringBuilder contacts = new StringBuilder();
    while (cursor != null && cursor.moveToNext()) {
        @SuppressLint("Range") String name = cursor.getString(cursor.getColumnIndex
                        (ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME));
        @SuppressLint("Range") String phone = cursor.getString(cursor.getColumnIndex
                        (ContactsContract.CommonDataKinds.Phone.NUMBER));
        contacts.append(name).append(": ").append(phone).append("\n");
    }
    if (cursor != null) {
        cursor.close();
    }
    showDataDialog("Contacts", contacts.toString());
}

private void loadSMS() {
    ContentResolver resolver = getContentResolver();
    Uri uri = Telephony.Sms.CONTENT_URI;
   Cursor cursor = resolver.query(uri, null, null, null, Telephony.Sms.DATE + " DESC");
    StringBuilder smsMessages = new StringBuilder();
    while (cursor != null && cursor.moveToNext()) {
       @SuppressLint("Range") String address = cursor.getString(cursor.getColumnIndex
                                            (Telephony.Sms.ADDRESS));
        @SuppressLint("Range") String body = cursor.getString(cursor.getColumnIndex
                                            (Telephony.Sms.BODY));
        smsMessages.append("From: ").append(address).append("\nMessage: ")
                                    .append(body).append("\n\n");
    }
    if (cursor != null) {
        cursor.close();
    }
    showDataDialog("SMS Messages", smsMessages.toString());
}

private void showDataDialog(String title, String message) {
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setTitle(title);
    textView = new TextView(this);
    textView.setText(message);
```

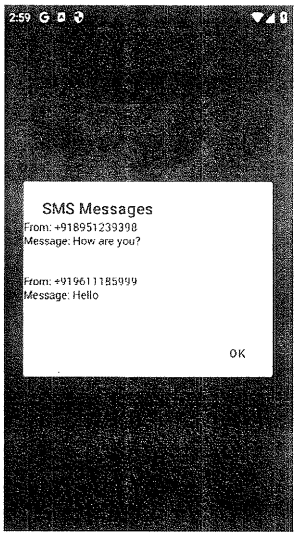
```
        textView.setMovementMethod(new ScrollingMovementMethod());
        builder.setView(textView);
        builder.setPositiveButton("OK", (dialog, which) -> dialog.dismiss());
        builder.show();
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[]
                                    permissions, @NonNull int[] grantResults) {
      super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (grantResults.length > 0 && grantResults[0]
                                == PackageManager.PERMISSION_GRANTED) {
            loadData(requestCode);
        } else {
          Toast.makeText(this, "Permission Denied", Toast.LENGTH_SHORT).show();
        }
    }
}
```

Output

- When the application is run, it will display two buttons: "Load Contacts" and "Load SMSs".
- Clicking either button will check for the necessary permissions, request them if needed, and then load and display the corresponding data in a dialog.







Explanation

- **Permission Handling:** The 'checkPermissionAndLoad' method checks if the necessary permission is granted. If not, it requests the permission.
- **Data Loading:**
  - 'loadContacts' method queries the contacts content provider and builds a string of contact names and phone numbers.
  - 'loadSMS' method queries the SMS content provider and builds a string of SMS addresses and bodies.
- **Displaying Data:** The 'showDataDialog' method displays the loaded data in a dialog with a scrolling text view.
- **Permission Result Handling:** The 'onRequestPermissionsResult' method handles the result of the permission request and proceeds to load the data if permission is granted.

9.10 Creating Your Own Content Provider (Custom Content Provider)

A custom content provider in Android is a component that allows applications to securely share data with other applications. It acts as an intermediary between the data source (such as a SQLite database, file system, or network) and other applications that want to access that data. By implementing a custom content provider, developers can define a structured way for external applications to interact with and manipulate the data stored within their app.

While Android offers several built-in content providers, there are scenarios where an application needs to expose its unique data set that isn't covered by these built-in providers. For example, if an application manages a custom database of user notes, tasks, or other proprietary data, creating a custom content provider helps other applications to access and manipulate this data.

What is Custom Content Provider ?

A custom content provider in Android is a specialized component designed to manage access to a structured set of data within an application. Unlike the built-in content providers provided by Android for common data types like contacts, calendar events, or media files, a custom content provider is created by the application developer to expose the app's specific data to other applications. This allows other applications to query, insert, update, and delete the data in a controlled and secure manner.

Example | **Understanding Custom Content Provider**

When developing an app that stores user information or any other type of data, if there is a need for other apps to access and utilize that data, a custom content provider becomes essential. By creating a custom content provider, developers can define the data schema, implement data access methods, and expose the data to external applications in a secure and controlled manner.

Scenario: App1 Accessing Data from App2 Using a Custom Content Provider

- **App1:**
  - **Purpose:** App1 needs to access specific data stored in App2.
  - **Implementation:** App1 can interact with App2's data through the custom content provider exposed by App2.

- **Permissions:** App1 must have the necessary permissions to access the data provided by App2's content provider.
- **Data Retrieval:** App1 can query, insert, update, or delete data in App2's database by utilizing the content provider's CRUD operations.

✦ **App2:**

- **Custom Content Provider:** App2 implements a custom content provider to manage and expose its data securely.
- **Data Schema:** App2 defines the data schema, including tables, columns, and URIs, within the content provider.
- **Data Access Methods:** App2 implements methods for querying, inserting, updating, and deleting data in a structured way.
- **Security:** App2 enforces permissions and access controls to ensure that only authorized apps can interact with its data.

Interaction Flow:

1. **App1 Requests Data:** App1 sends a query request to the content provider URI provided by App2.
2. **Content Provider Handling:** App2's content provider processes the request, performs the necessary database operations, and returns the requested data to App1.
3. **Data Manipulation:** App1 can manipulate the data retrieved from App2's content provider based on its requirements.
4. **Data Updates:** If App1 needs to update the data in App2's database, it can do so by utilizing the content provider's update methods.

Benefits:

- **Data Sharing:** Enables controlled sharing of data between different apps.
- **Data Consistency:** Ensures that data access and modifications follow a predefined schema.
- **Security:** Enforces access control mechanisms to protect sensitive data.
- **Interoperability:** Promotes interoperability by providing a standardized way for apps to interact with shared data.

In this scenario, App1 leverages the custom content provider implemented by App2 to access and utilize specific data stored within App2's app. By following this approach, developers can establish a secure and structured mechanism for sharing data across different applications while maintaining data integrity and security.

Example **Custom Content Provider Example Scenarios**

1. **Notes Application:** A notes application may want to allow other applications to read, write, and manage the notes. By creating a custom content provider, the notes can be accessed and manipulated using a standardized API.
2. **Task Management App:** A task management app may need to expose its tasks to calendar apps, widgets, or other productivity tools.
3. **Photo Gallery App:** A photo gallery app may want to share selected images with social media apps or editing tools. By implementing a custom content provider, the app can allow external applications to access and display specific photos based on predefined criteria.

4. **Fitness Tracking Application:** A fitness tracking application may need to share workout data with health monitoring apps or fitness communities. Through a custom content provider, the app can provide access to workout logs, progress metrics, and achievements.

Steps to Create a Custom Content Provider

1. **Define the Content Provider Class:**
   - Create a Java class that extends 'ContentProvider' to implement the custom content provider.
   - Override the necessary methods such as 'query()', 'insert()', 'update()', 'delete()', and 'getType()' to handle data operations.
   - The content provider class acts as the interface for interacting with the underlying data source.
2. **Define URIs and MIME Types:**
   - Define content URIs that represent the data resources within the content provider.
   - Content URIs follow the format 'content://authority/path/id' and are used to access specific data or perform CRUD operations.
   - Define MIME types for the data handled by the content provider, specifying the type of data that can be returned.
3. **Implement CRUD Methods:**
   - **query():** Implement the 'query()' method to retrieve data from the content provider based on specified criteria.
   - **insert():** Implement the 'insert()' method to add new data to the content provider.
   - **update():** Implement the 'update()' method to modify existing data in the content provider.
   - **delete():** Implement the 'delete()' method to remove data from the content provider.
   - **getType():** Implement the 'getType()' method to return the MIME type of the data associated with a content URI.
4. **Declare the Content Provider in the Manifest:**

   Register the custom content provider in the 'AndroidManifest.xml' file using the '<provider>' tag.
   - Specify the authority, which is a unique string to identify the content provider.
   - Define the class name of the content provider that extends 'ContentProvider'.
   - Declare the permissions required to access the content provider, such as read or write permissions.
5. **Enforce Permissions:**
   - Specify the necessary permissions in the 'AndroidManifest.xml' file to control access to the content provider.
   - Define permissions like 'READ_PROVIDER' or 'WRITE_PROVIDER' to restrict read and write operations on the data.

Creating a Custom Content Provider

n this section, we will create a custom content provider in Android, which allows other applications o access our app's data, similar to how we access built-in content providers like Contacts or SMS. This example will demonstrate creating a notes application with basic CRUD operations (Create, Read, Update, Delete) using a custom content provider. The application will include a simple UI for nteracting with the data.

Step 1: Project Setup:

- **Create a new project in Android Studio:**
  1. Open Android Studio and select 'Create New Project'.
  2. Choose 'Phone and Tablet' -> 'Empty Views Activity' and click 'Next'.
  3. Set the 'Name' to 'ContentProviderDemo'.
  4. Set the 'Package Name' to 'com.example.contentproviderdemo'.
  5. Set the 'Save location' and 'Language' to 'Java'.
  6. Click 'Finish'.

Step 2: Define the Database Schema:

Define a helper class to manage the SQLite database. This class will handle the creation and upgrading of the database.

Code 'DatabaseHelper.java'

```
package com.example.customcontentprovider;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class DatabaseHelper extends SQLiteOpenHelper {

    private static final String DATABASE_NAME = "notes.db";
    private static final int DATABASE_VERSION = 1;

    private static final String TABLE_CREATE =
            "CREATE TABLE notes (_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                    "title TEXT, " +
                    "content TEXT);";

    public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
```

```
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(TABLE_CREATE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS notes");
        onCreate(db);
    }
}
```

Explanation

- **DATABASE_NAME:** The name of the database file.
- **DATABASE_VERSION:** The version number of the database.
- **TABLE_CREATE:** The SQL statement to create the 'notes' table.
- **onCreate():** Called when the database is created for the first time.
- **onUpgrade():** Called when the database needs to be upgraded.

Step 3: Define the Content Provider Class:

Create a custom content provider class that extends 'ContentProvider' and implements the required methods to handle data operations.

Code 'NotesProvider.java'

```
package com.example.customcontentprovider;

import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.net.Uri;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;

public class NotesProvider extends ContentProvider {

    private static final String AUTHORITY = "com.example.customcontentprovider.provider";
    private static final String PATH_NOTES = "notes";
    public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY
                                                    + "/" + PATH_NOTES);
```

```
private static final int NOTES = 1;
private static final int NOTES_ID = 2;
private static final UriMatcher uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);

static {
    uriMatcher.addURI(AUTHORITY, PATH_NOTES, NOTES);
    uriMatcher.addURI(AUTHORITY, PATH_NOTES + "/#", NOTES_ID);
}

private SQLiteDatabase database;

@Override
public boolean onCreate() {
    DatabaseHelper helper = new DatabaseHelper(getContext());
    database = helper.getWritableDatabase();
    return true;
}

@Nullable
@Override
public Cursor query(@NonNull Uri uri, @Nullable String[] projection,
       @Nullable String selection, @Nullable String[] selectionArgs,
                                                @Nullable String sortOrder) {
    switch (uriMatcher.match(uri)) {
        case NOTES:
            return database.query("notes", projection, selection, selectionArgs,
                                  null, null, sortOrder);
        case NOTES_ID:
            selection = "_id=?";
            selectionArgs = new String[]{String.valueOf(ContentUris.
                                                              parseId(uri))};
            return database.query("notes", projection, selection,
                                    selectionArgs, null, null, sortOrder);
        default:
            throw new IllegalArgumentException("Unknown URI: " + uri);
    }
}

@Nullable
@Override
public Uri insert(@NonNull Uri uri, @Nullable ContentValues values) {
    long id = database.insert("notes", null, values);
    return ContentUris.withAppendedId(CONTENT_URI, id);
}
```

```
    @Override
    public int update(@NonNull Uri uri, @Nullable ContentValues values,
                 @Nullable String selection,@Nullable String[] selectionArgs) {
        switch (uriMatcher.match(uri)) {
            case NOTES:
                return database.update("notes", values, selection, selectionArgs);
            case NOTES_ID:
                selection = "_id=?";
                selectionArgs = new String[]{String.valueOf
                                                (ContentUris.parseId(uri))};
                return database.update("notes", values, selection, selectionArgs);
            default:
                throw new IllegalArgumentException("Unknown URI: " + uri);
        }
    }

    @Override
    public int delete(@NonNull Uri uri, @Nullable String selection,
                                      @Nullable String[] selectionArgs) {
        switch (uriMatcher.match(uri)) {
            case NOTES:
                return database.delete("notes", selection, selectionArgs);
            case NOTES_ID:
                selection = "_id=?";
                selectionArgs = new String[]{String.valueOf(ContentUris.parseId(uri))};
                return database.delete("notes", selection, selectionArgs);
            default:
                throw new IllegalArgumentException("Unknown URI: " + uri);
        }
    }

    @Nullable
    @Override
    public String getType(@NonNull Uri uri) {
        switch (uriMatcher.match(uri)) {
            case NOTES:
             return "vnd.android.cursor.dir/vnd.com.example.customcontentprovider.notes";
            case NOTES_ID:
            return "vnd.android.cursor.item/vnd.com.example.customcontentprovider.notes";
            default:
                throw new IllegalArgumentException("Unknown URI: " + uri);
        }
    }
}
```

Explanation

- **AUTHORITY:** A unique string that identifies the content provider.
- **PATH_NOTES:** The path to the notes data.
- **CONTENT_URI:** The base URI for accessing the content provider.
- **uriMatcher:** A utility class to match URIs to integer constants.
- **onCreate():** Initializes the database helper and gets a writable database instance.
- **query():** Handles query requests from clients.
- **insert():** Handles insert requests from clients.
- **update():** Handles update requests from clients.
- **delete():** Handles delete requests from clients.
- **getType():** Returns the MIME type for the data at the given URI.

Step 4: Define the UI in 'activity_main.xml':

Define the layout of the main activity, including buttons for adding, updating, and deleting notes.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <EditText
        android:id="@+id/editTextTitle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Title" />

    <EditText
        android:id="@+id/editTextContent"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Content" />

    <Button
        android:id="@+id/buttonAddNote"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add Note" />
```

```
    <Button
        android:id="@+id/buttonUpdateNote"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Update Note" />

    <Button
        android:id="@+id/buttonDeleteNote"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Delete Note" />

    <ListView
        android:id="@+id/listViewNotes"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:choiceMode="singleChoice" />
</LinearLayout>
```

Explanation

- 'LinearLayout' is used as the parent layout.
- Two 'EditText' fields for entering the title and content of the note.
- Three buttons: 'buttonAddNote', 'buttonUpdateNote', and 'buttonDeleteNote' for adding, updating, and deleting notes, respectively.
- A 'ListView' to display the list of notes.

Step 5: Implement the Main Activity in 'MainActivity.java':

The main activity handles the user interactions and data loading. It checks for necessary permissions and then loads the data.

Code MainActivity.java

```
package com.example.customcontentprovider;

import android.annotation.SuppressLint;
import android.content.ContentResolver;
import android.content.ContentUris;
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
```

```
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.Toast;

import androidx.appcompat.app.AppCompatActivity;

import java.util.ArrayList;

public class MainActivity extends AppCompatActivity {

    private EditText editTextTitle, editTextContent;
    private Button buttonAddNote, buttonUpdateNote, buttonDeleteNote;
    private ListView listViewNotes;

    private ArrayAdapter<String> adapter;
    private ArrayList<String> notesList;
    private ArrayList<Long> notesIds;

    private Uri currentNoteUri = null;

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

        editTextTitle = findViewById(R.id.editTextTitle);
        editTextContent = findViewById(R.id.editTextContent);
        buttonAddNote = findViewById(R.id.buttonAddNote);
        buttonUpdateNote = findViewById(R.id.buttonUpdateNote);
        buttonDeleteNote = findViewById(R.id.buttonDeleteNote);
        listViewNotes = findViewById(R.id.listViewNotes);

        notesList = new ArrayList<>();
        notesIds = new ArrayList<>();
        adapter = new ArrayAdapter<>(this,
          android.R.layout.simple_list_item_single_choice, notesList);
        listViewNotes.setAdapter(adapter);
        buttonAddNote.setOnClickListener(v -> addNote());
        buttonUpdateNote.setOnClickListener(v -> updateNote());
        buttonDeleteNote.setOnClickListener(v -> deleteNote());
```

```
        listViewNotes.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view,
                                                    int position, long id) {
                currentNoteUri = ContentUris.withAppendedId(NotesProvider.
                                    CONTENT_URI, notesIds.get(position));
                loadNoteForEdit(position);
            }
        });

        loadNotes();
    }

    private void addNote() {
        String title = editTextTitle.getText().toString();
        String content = editTextContent.getText().toString();

        ContentValues values = new ContentValues();
        values.put("title", title);
        values.put("content", content);

        ContentResolver resolver = getContentResolver();
        resolver.insert(NotesProvider.CONTENT_URI, values);

        loadNotes();
    }

    private void updateNote() {
        if (currentNoteUri == null) {
            Toast.makeText(this, "Select a note to update", Toast.LENGTH_SHORT).show();
            return;
        }

        String title = editTextTitle.getText().toString();
        String content = editTextContent.getText().toString();

        ContentValues values = new ContentValues();
        values.put("title", title);
        values.put("content", content);

        ContentResolver resolver = getContentResolver();
        resolver.update(currentNoteUri, values, null, null);
```

```
        loadNotes();
    }

    private void deleteNote() {
        if (currentNoteUri == null) {
            Toast.makeText(this, "Select a note to delete",
                                    Toast.LENGTH_SHORT).show();
            return;
        }
        ContentResolver resolver = getContentResolver();
        resolver.delete(currentNoteUri, null, null);
        loadNotes();
    }

    private void loadNotes() {
        notesList.clear();
        notesIds.clear();

        ContentResolver resolver = getContentResolver();
        Cursor cursor = resolver.query(NotesProvider.CONTENT_URI, null, null,
                                        null, null);
        if (cursor != null) {
            while (cursor.moveToNext()) {
                @SuppressLint("Range") String title = cursor.getString
                                        (cursor.getColumnIndex("title"));
                @SuppressLint("Range") long id = cursor.getLong
                                        (cursor.getColumnIndex("_id"));
                notesList.add(title);
                notesIds.add(id);
            }
            cursor.close();
        }

        adapter.notifyDataSetChanged();
        currentNoteUri = null;
        editTextTitle.setText("");
        editTextContent.setText("");
    }

    private void loadNoteForEdit(int position) {
        ContentResolver resolver = getContentResolver();
        Cursor cursor = resolver.query(currentNoteUri, null, null, null, null);
```

```
        if (cursor != null) {
            if (cursor.moveToFirst()) {
                @SuppressLint("Range") String title = cursor.getString
                                        (cursor.getColumnIndex("title"));
                @SuppressLint("Range") String content = cursor.getString
                                        (cursor.getColumnIndex("content"));
                editTextTitle.setText(title);
                editTextContent.setText(content);
            }
            cursor.close();
        }
    }
}
```

Explanation

- **Permission Handling:** The 'checkPermissionAndLoad' method checks if the necessary permission is granted. If not, it requests the permission.
- **Data Loading:**

  'loadContacts' method queries the contacts content provider and builds a string of contact names and phone numbers.

  'loadSMS' method queries the SMS content provider and builds a string of SMS addresses and bodies.
- **Displaying Data:** The 'showDataDialog' method displays the loaded data in a dialog with a scrolling text view.
- **Permission Result Handling:** The 'onRequestPermissionsResult' method handles the result of the permission request and proceeds to load the data if permission is granted.

Step 6: Declare the Content Provider in the Manifest

Register the custom content provider in the 'AndroidManifest.xml' file.

Code AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.customcontentprovider">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.CustomContentProvider">

        <activity android:name=".MainActivity"
            android:exported="true">
```

```
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <provider
            android:name=".NotesProvider"
            android:authorities="com.example.customcontentprovider.provider"
            android:exported="true" />
    </application>

</manifest>
```

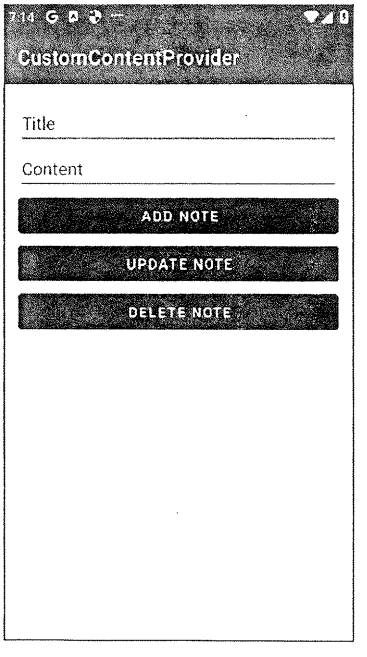
Explanation

- **android:name:** The class name of the content provider.
- **android:authorities:** The authority string to identify the content provider.
- **android:exported:** Whether the content provider can be accessed by other applications.

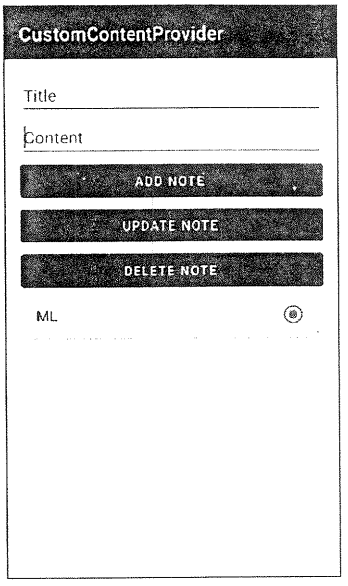
Output : Output and Behavior

When the application is run, it will display two EditText fields and three buttons for managing notes. The user can add, update, and delete notes. Notes are displayed in a ListView, and selecting a note will populate the EditText fields with its details for editing.

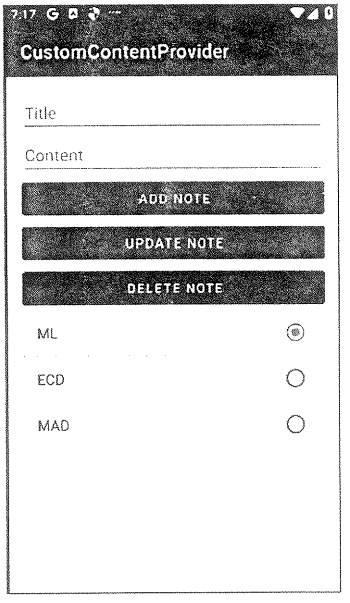
- **Add Note:** Enter a title and content, then click "Add Note" to save the note.
- **Update Note:** Select a note from the ListView, modify the title or content, then click "Update Note" to save the changes.
- **Delete Note:** Select a note from the ListView and click "Delete Note" to remove the note.



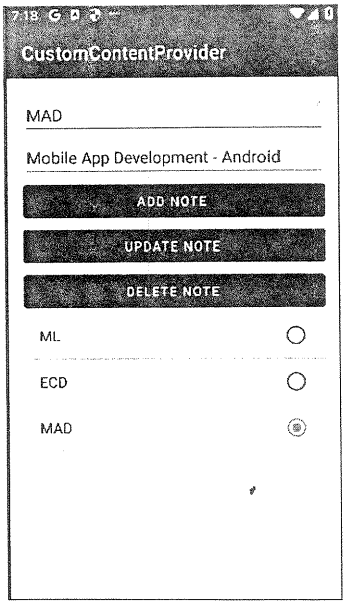
Launch Screen



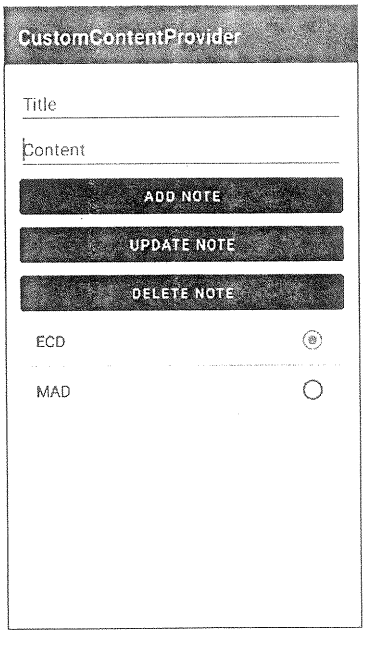
Added a Note (ML as Note Title)



Added a Note (MAD and ECD)



Modified MAD Content from Mobile App Dev to Mobile App Development - Android



After Deleting ML

9.11 Using Custom Content Provider

Once a custom content provider is created, it can be used by other applications to access and manipulate the data provided by it. This enables data sharing and reuse across different applications. In this example, we will use the custom content provider we created in the 'ContentProviderDemo' project to access and display notes in another application.

We have created a custom content provider for managing notes in the 'ontentProviderDemo' project. Now, we will create a new project called 'MyNotesApp' to use this custom content provider and load all the notes. This demonstrates how one app can access data provided by another app through a content provider.

Basic Steps to Use a Custom Content Provider

1. **Declare Permissions:** The consuming app must declare the necessary permissions to read from or write to the content provider.
2. **Use Content Resolver:** Use 'ContentResolver' to query, insert, update, or delete data from the custom content provider.
3. **Handle Data:** Process the data returned by the content provider as needed, such as displaying it in a user interface.

Step-by-Step Implementation of Using Custom Content Provider

Step 1: Project Setup:

- Create a new project in Android Studio:
  1. Open Android Studio and select 'Create New Project'.
  2. Choose 'Phone and Tablet' -> 'Empty Views Activity' and click 'Next'.
  3. Set the 'Name' to 'MyNotesApp'.
  4. Set the 'Package Name' to 'com.example.mynotesapp'.
  5. Set the 'Save location' and 'Language' to 'Java'.
  6. Click 'Finish'.

Step 2: Declare Permissions in 'AndroidManifest.xml':

Declare the necessary permissions to read data from the custom content provider.

Code AndroidManifest.xml

```
<!-- Permission to read from custom content provider -->
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Explanation

- '<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />': This permission is required to read data from the custom content provider.

Step 3: Define the UI in 'activity_main.xml':

Define the layout of the main activity, including a 'ListView' to display the notes.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <ListView
        android:id="@+id/listViewNotes"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Explanation

- 'LinearLayout' is used as the parent layout.
- A 'ListView' to display the list of notes.

Step 4: Implement the Main Activity in 'MainActivity.java':

The main activity handles loading and displaying notes from the custom content provider.

Code MainActivity.java

```
package com.example.mynotesapp;

import android.annotation.SuppressLint;
import android.content.ContentResolver;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.ListView;

import androidx.appcompat.app.AppCompatActivity;

import java.util.ArrayList;

public class MainActivity extends AppCompatActivity {

    private ListView listViewNotes;
    private ArrayAdapter<String> adapter;
    private ArrayList<String> notesList;
```

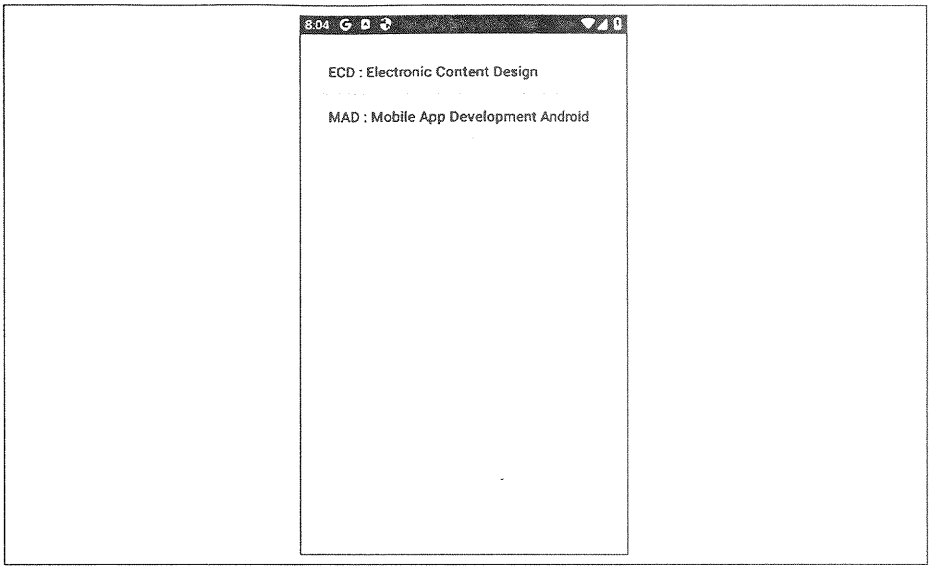
```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        listViewNotes = findViewById(R.id.listViewNotes);
        notesList = new ArrayList<>();
        adapter = new ArrayAdapter<>(this,
                            android.R.layout.simple_list_item_1, notesList);
        listViewNotes.setAdapter(adapter);

        loadNotes();
    }

    private void loadNotes() {
        ContentResolver resolver = getContentResolver();
        Uri uri = Uri.parse
              ("content://com.example.customcontentprovider.provider/notes");
        Cursor cursor = resolver.query(uri, null, null, null, null);

        if (cursor != null) {
            while (cursor.moveToNext()) {
                @SuppressLint("Range") String title = cursor.getString
                                (cursor.getColumnIndex("title"));
                @SuppressLint("Range") String content = cursor.getString
                                (cursor.getColumnIndex("content"));
                notesList.add(title.concat(" : ").concat(content));
            }
            cursor.close();
        }

        adapter.notifyDataSetChanged();
    }
}
```

Explanation

- **Load Notes:**
  - The 'loadNotes' method queries the custom content provider using the 'ContentResolver' and retrieves the list of notes.
  - The retrieved notes are added to the 'notesList' and displayed in the 'ListView' using an 'ArrayAdapter'.

Output

When the 'MyNotesApp' application is run, it will display a list of notes fetched from the custom content provider. This demonstrates how one app can access and display data provided by another app through a content provider.



9.12 Review Questions

Section - A

1. What is a Content Provider in Android?
2. What is the primary purpose of a Content Provider?
3. What is Content Provider URI ? Give an example.
4. Give two examples of Built-in Content Providers.
5. What is Content Resolver?
6. What is Content Values ?
7. What is a Cursor ?
8. Mention the Types of Permissions in Content Providers.
9. What is Custom Content Provider ?
10. Describe the importance of the ContentResolver class.
12. Describe the importance of the ContentValues class.
13. How to query data using a Cursor? Give a example code.
14. Write any two Methods of Content Resolver.
15. Mention two advanatges of Content Provider.

Section - B

1. Explain the features of Content Providers in Android.
2. Mention the Advantages and Disadvantages of Content Providers.
3. What is Content Provider URI ? Explain the Structure of a Content Provider URI with an example.
4. What is Content Resolver? Write the Features of Content Resolver.
5. Explain the Methods of Content Resolver.
6. What are Content Values ? Write the Features and Importance of ContentValues.
7. Explain the Methods of ContentValues.
8. What is a Cursor ? Write the features and Importance of Cursor.
9. Explain how to query database and fetch the records using a Cursor.
10. What is Custom Content Provider ? Write the Steps to Create a Custom Content Provider.

Section - C

1. Explain Content Resolver Methods along with example code.
2. Explain How ContentValues are used in CRUD Operations.
3. Explain Permissions in ContentProviders. How to declare and Request Permissions in Android?
4. Explain the steps in using Android Built-in Content Providers.
5. Explain the process of creating a Custom Content Provider with example code.
6. Explain the steps in using Custom Content Provider.

UNIT - IV

CHAPTER 10

NETWORKING & CREATING SERVICES

CONTENTS

- Introduction
- Reading/Writing Local Data
- Accessing the Internal File System
- Accessing the SD Card
- Preparing App for Publishing
- Consuming Web Services Using HTTP
- Consuming JSON Services
- Creating Your Own Services
- Binding Activities to Services
- Understanding Threading
- Review Questions

10.1 Introduction to Networking

n Chapter 7, we discussed how apps can communicate with the outside world using SMS messaging ınd emails. Another way to connect with the outside world is through the wireless network on an ındroid device.

n this section, we will explore how Android applications interact with the broader digital landscape ria the internet. By understanding the mechanisms behind connecting to web servers using the HTTP ɔrotocol, we will uncover the process through which apps retrieve valuable data from online sources. Γhis knowledge is crucial for developers seeking to create dynamic and data-driven applications that ·etch information from remote servers in real-time.

Ve will also discuss the art of parsing XML files, a structured data format commonly used in web :ervices, to extract specific and relevant information. This skill is essential for handling and processing lata exchanged between Android apps and web servers efficiently. Additionally, we will get into the ·ealm of JSON, a lightweight and versatile data format that has gained popularity for its simplicity ınd ease of use. We can streamline data handling processes and enhance the performance of the ıpplication using JSON when dealing with web-based information.

What are Web Services?

- Web services are software systems designed to facilitate communication and data exchange over the internet. They enable different applications or systems to interact with each other regardless of the platforms or programming languages they are built with. Web services typically use standard protocols and data formats to ensure interoperability and seamless integration between disparate systems.
- Web services can provide various functionalities such as fetching data from databases, performing calculations, processing requests, and more. They are commonly used to enable communication between client applications (such as mobile apps) and server-side systems, allowing for the exchange of information in a structured and standardized manner.

Communicating with Web Services

- Communicating with web services involves sending requests and receiving responses over the internet using predefined protocols and data formats. Clients, such as mobile apps, interact with web services by making HTTP requests to specific endpoints (URLs) provided by the service. The requests typically include parameters or data required for the operation, such as retrieving information or submitting updates.
- Web services process these requests on the server-side, perform the necessary operations, and send back responses containing the requested data or status information. Communication with web services is based on established standards like SOAP (Simple Object Access Protocol) or REST (Representational State Transfer), which define how data is exchanged between clients and servers.

Role of HTTP in Consuming Web Services

- HTTP (Hypertext Transfer Protocol) plays a crucial role in consuming web services by establishing a standardized communication protocol between clients and servers. Through

its request-response model, clients can send HTTP requests to servers, specifying the desired actions or data retrieval operations. The versatility of HTTP methods like GET, POST, PUT, and DELETE allows clients to interact with web services in various ways, such as fetching information, submitting data, updating resources, and deleting content.

- HTTP's stateless nature ensures that each request is independent, enhancing scalability and reliability in consuming web services without the need to maintain session information between requests. The inclusion of header information in HTTP requests enables clients to convey additional details to servers, facilitating secure and efficient data exchange.

Understanding JSON

- JSON (JavaScript Object Notation) serves as a lightweight and versatile data interchange format widely used in modern software development. Its simplicity, human-readable syntax, and compatibility with various programming languages make it an ideal choice for transmitting structured data between different systems. JSON's efficiency in terms of bandwidth usage and parsing speed, along with its support for complex data structures like nested objects and arrays, enables seamless communication in web services and API integrations.
- JSON plays a pivotal role in facilitating data exchange, interoperability, and integration across diverse software systems. Its role extends to enabling efficient communication between client-side and server-side components, supporting complex data structures, and streamlining API interactions. JSON's widespread adoption in web development, mobile applications, and web services underscores its significance in modern software engineering as a versatile and efficient data interchange format.

Important Classes and Methods in Networking

1. 'HttpURLConnection' Class :

HttpURLConnection is a standard class provided by the Java SDK and is widely used in Android for managing HTTP connections. This class offers a flexible and efficient way to communicate with web servers using the HTTP protocol. It allows developers to send requests and receive responses over the web, supporting various HTTP methods such as GET, POST, PUT, DELETE, etc.

Important Methods and Their Usage:

Method & Description	Example
'openConnection()': Opens a connection to the specified URL. This method is used to establish a connection to a URL.	`HttpURLConnection urlConnection = (HttpURLConnection) new URL("http://example.com").openConnection();`
'setRequestMethod(String method)': Sets the request method (e.g., "GET", "POST"). This method defines the HTTP method used for the request.	`urlConnection.setRequestMethod("GET");`
'setDoOutput(boolean doOutput)': Sets whether the connection allows output. This is typically set to 'true' for POST requests.	`urlConnection.setDoOutput(true);`

'setDoInput(boolean doInput)': Sets whether the connection allows input. This should be set to 'true' if the response will be read.	`urlConnection.setDoInput(true);`
'connect()': Opens a communications link to the resource referenced by the URL. This method actually establishes the connection.	`urlConnection.connect();`
'getInputStream()': Returns an input stream that reads from this open connection. This method is used to read the server's response.	`InputStream in = urlConnection.getInputStream();`
'getResponseCode()': Gets the HTTP response code from the server. This method retrieves the HTTP status code, such as 200 for OK or 404 for not found.	`int responseCode = urlConnection.getResponseCode();`
'setRequestProperty(String key, String value)': Sets a general request property. This method is used to set header fields.	`urlConnection.setRequestProperty("Content-Type", "application/json");`
'getErrorStream()': Returns the error stream if the connection failed but the server returned useful information.	`InputStream errorStream = urlConnection.getErrorStream();`
'disconnect()': Indicates that other requests to the server are unlikely in the near future. This method releases resources associated with the connection.	`urlConnection.disconnect();`

2. **AsyncTask Class:**

'AsyncTask' is a class in Android that allows to perform background operations and publish results on the UI thread without having to manipulate threads and handlers. It simplifies the process of managing background operations and updating the UI, making it ideal for tasks that need to run in the background but also need to update the user interface.

Important Methods and Their Usage

Method & Description	Example
'onPreExecute()': Runs on the UI thread before the background task begins. It is used to set up the task, for example, by showing a progress bar.	`@Override protected void onPreExecute() { super.onPreExecute(); // Setup task }`
'doInBackground(Params... params)': Runs on a background thread. This is where the background task is performed.	`@Override protected String doInBackground(String... params) { // Perform background task return "Result"; }`
'onProgressUpdate(Progress... values)': Runs on the UI thread and is used to display progress updates during the background computation.	`@Override protected void onProgressUpdate(Integer... values) { // Update progress }`

'onPostExecute(Result result)': Runs on the UI thread after the background task is completed. It is used to update the UI with the result of the background task.	`@Override protected void onPostExecute(String result) { super.onPostExecute(result); // Update UI with result }`
'execute(Params... params)': Executes the task with the specified parameters.	`new MyAsyncTask().execute("param1", "param2");`

10.2 Consuming Web Services Using Http

Consuming web services refers to the process of utilizing and interacting with services provided by remote servers over the internet. Web services are software systems designed to enable communication and data exchange between different applications or systems over a network, typically using standard protocols like HTTP.

Consuming web services using HTTP involves leveraging the Hypertext Transfer Protocol (HTTP) to establish communication between clients and servers over the internet. HTTP serves as the foundation of the World Wide Web, offering a standardized and versatile approach for exchanging data. When developers consume web services using HTTP, they utilize various HTTP methods such as GET, POST, PUT, and DELETE to perform specific actions like retrieving, submitting, updating, and deleting data through interactions with designated web service endpoints. These methods enable developers to communicate effectively with web services by structuring requests with headers, parameters, and payloads to convey necessary information to the server.

This section will explain how to use HTTP to talk to web servers and download text and files. We will also learn how to read XML files and get the important parts, which is useful for accessing web services. In addition to XML, we will look at JSON (JavaScript Object Notation), a simpler alternative. We'll use tools in the Android SDK to work with JSON data.

10.2.1 Importance of Web Services in Mobile Apps

Web services play a crucial role in mobile app development by enabling seamless communication between mobile applications and external servers or services over the internet. They provide a standardized way for apps to access and exchange data, perform various functionalities, and integrate with external resources. The importacne and benefits of using Webservices in Mobile Apps are listed below.

1. **Data Accessibility:** Web services allow mobile apps to access a wide range of data and services available on the web, such as real-time information, databases, and external APIs. This enables apps to provide up-to-date and relevant content to users without storing all the data locally.
2. **Functionality Expansion:** By leveraging web services, mobile apps can extend their functionalities beyond what is possible with local resources. Apps can integrate with third-party services, utilize advanced algorithms, and access cloud-based resources to enhance user experience and offer innovative features.
3. **Efficiency and Cost-Effectiveness:** Using web services eliminates the need for developers to build and maintain complex functionalities from scratch. By leveraging existing web services, developers can save time, resources, and costs associated with developing and maintaining specific modules or features within the app.

4. **Scalability and Flexibility:** Web services provide scalability by allowing apps to handle a large volume of data and user requests efficiently. Apps can easily scale up or down based on demand without significant changes to the underlying infrastructure. Additionally, web services offer flexibility in integrating new features or updating existing functionalities without major app revisions.
5. **Enhanced User Experience:** Web services play a vital role in enhancing the user experience of mobile apps by providing access to a diverse range of functionalities and content. By integrating with web services, apps can offer personalized and dynamic content, interactive features, and real-time updates to users.

Examples | **Example Scenarios of Using Web Services in Mobile Apps**

1. **Exchange Rates Service:**
   - **Scenario:** A currency converter app is being developed that necessitates real-time exchange rates for various currencies.
   - Instead of developing and maintaining a currency exchange rate module within the app, integration with a third-party web service providing up-to-date exchange rate data is preferred.
   - **Example:** Utilization of a free or paid exchange rate API like Open Exchange Rates or X-Rates API to retrieve accurate and current exchange rate information for seamless currency conversion within the app.
2. **Weather Forecast Service:**
   - **Scenario:** A weather app is envisioned to exhibit current weather conditions and forecasts for different locations.
   - Instead of locally storing weather data, the app can employ a weather web service to procure real-time weather information from external sources.
   - **Example:** Integration with a weather API such as OpenWeatherMap or WeatherAPI to fetch weather data like temperature, humidity, and forecasts for specific locations, delivering users with precise and timely weather updates.
3. **Payment Gateway Service:**
   - **Scenario:** An e-commerce app is under development necessitating secure payment processing for transactions.
   - Instead of constructing a payment gateway from scratch, integration with a dependable payment gateway web service to manage secure payment transactions is favored.
   - **Example:** Adoption of a payment gateway service like PayPal, Stripe, or Braintree to facilitate secure online payments within the app, ensuring a seamless and trustworthy payment experience for users.
4. **Location-Based Service:**
   - **Scenario:** Development of a location-based app that furnishes personalized content or services based on the user's current location.
   - By integrating with a location-based web service, the app can access geolocation data and offer location-specific features to enrich user engagement.
   - **Example:** Utilization of location services like Google Maps API or Foursquare API to retrieve location data, exhibit maps, provide location-based recommendations, and enable geotagging functionalities within the app.

10.2.2 Basic Steps in Consuming Web Services Using Http

To consume web services using HTTP in Android, developers typically utilize the 'HttpURLConnection' class. The basic steps involved in consuming web services using HTTP in an Android application are:

1. **Add Internet Permission:** Ensure that the AndroidManifest.xml file includes the necessary permission to access the internet. Add the following permission within the '<manifest>' tag:

   ```
   <uses-permission android:name="android.permission.INTERNET" />
   ```

2. **Make HTTP Requests:**
   - Create an 'AsyncTask' or use a background thread to perform network operations asynchronously and avoid blocking the main UI thread.
   - Use 'HttpURLConnection' to establish a connection to the web service endpoint and configure the request method (GET, POST, PUT, DELETE).
   - Set request headers, parameters, and the request body as needed.
   - Send the request to the server and receive the response.
3. **Handle Response:**
   - When handling responses from web services in an Android application, developers need to consider various scenarios based on the format of the response data.
   - Process the data returned in the response and update the UI or perform any necessary operations based on the received data.
     - **Text Response:** For plain text responses, assign the content to a TextView for display.
     - **Image Response:** Decode image data and display it in an ImageView.
     - **XML Response:** Parse XML data to extract relevant information from the response.
     - **JSON Response:** Parse JSON data to extract required fields efficiently.
4. **Update UI:** Ensure that any UI updates based on the web service response are performed on the main UI thread using methods like 'runOnUiThread()'.
5. **Error Handling:**
   - Implement error handling mechanisms to manage network errors, timeouts, and exceptions that may occur during the HTTP request.
   - Display appropriate error messages or take corrective actions based on the type of error encountered.

10.2.3 Downloading Text

We will download text content from a specified URL using an 'HttpURLConnection'. The content will be read line by line and then displayed in a 'TextView' on the user interface. This involves setting up network permissions, performing network operations on a background thread using 'AsyncTask', and updating the UI from the background thread.

In this example, we will create an Android application that downloads text from a URL and displays it in a 'TextView'. This demonstrates how to perform network operations in the background and update the UI with the downloaded content.

Step 1: Create a New Android Project:

Create an Android project by selecting "Empty Views Activity", naming the project "TextDownloader", and choosing Java as the language

Step 2: Add Internet Permissions:

Ensure that the AndroidManifest.xml file includes the necessary permission to access the internet. Add the following permission within the '<manifest>' tag:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Step 3: Design the Layout:

Define the user interface in 'activity_main.xml'. This layout includes a 'TextView' to display the downloaded text and a 'Button' to initiate the download.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button_download"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Download Text"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp" />

    <TextView
        android:id="@+id/text_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/button_download"
        android:layout_marginTop="20dp"
        android:padding="16dp"
        android:textSize="16sp" />
</RelativeLayout>
```

Step 4: Implement the MainActivity:

In 'MainActivity.java', implement the logic to download the text using 'AsyncTask' and display it in the 'TextView'.

Code MainActivity.java

```
package com.example.textdownloader;

import android.os.AsyncTask;
import android.os.Bundle;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private Button buttonDownload;

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

        textView = findViewById(R.id.text_view);
        buttonDownload = findViewById(R.id.button_download);

        buttonDownload.setOnClickListener(v -> new DownloadTextTask().execute(
                "https://example-files.online-convert.com/document/txt/example.txt"));
    }

    private class DownloadTextTask extends AsyncTask<String, Void, String> {

        @Override
        protected String doInBackground(String... urls) {
            String urlString = urls[0];
            StringBuilder result = new StringBuilder();
            try {
                    URL url = new URL(urlString);
                    HttpURLConnection urlConnection = (HttpURLConnection)
                                                        url.openConnection();
                    BufferedReader reader = new BufferedReader(new InputStreamReader
                                            (urlConnection.getInputStream()));
```

```
                String line;
                while ((line = reader.readLine()) != null) {
                    result.append(line).append("\n");
                }

                reader.close();
                urlConnection.disconnect();
            } catch (Exception e) {
                e.printStackTrace();
                return "Error: " + e.getMessage();
            }

            return result.toString();
        }

        @Override
        protected void onPostExecute(String result) {
            if (result.startsWith("Error:")) {
                Toast.makeText(MainActivity.this, result, Toast.LENGTH_SHORT).show();
            } else {
                textView.setText(result);
            }
        }
    }
}
```

Explanation

1. **Setup and Layout:**

   The layout file ('activity_main.xml') contains a 'Button' with the ID 'button_download' and a 'TextView' with the ID 'text_view'. The 'Button' is used to trigger the download process, and the 'TextView' displays the downloaded text content.

2. **MainActivity:**

   - In the 'MainActivity' class, references to the 'Button' and 'TextView' are obtained using 'findViewById()'.
   - An 'OnClickListener' is set on the 'Button' to start the 'AsyncTask' for downloading the text from a specified URL when clicked.
   - The URL 'https://example-files.online-convert.com/document/txt/example.txt' is used as an example. This URL points to a text file that we want to download.

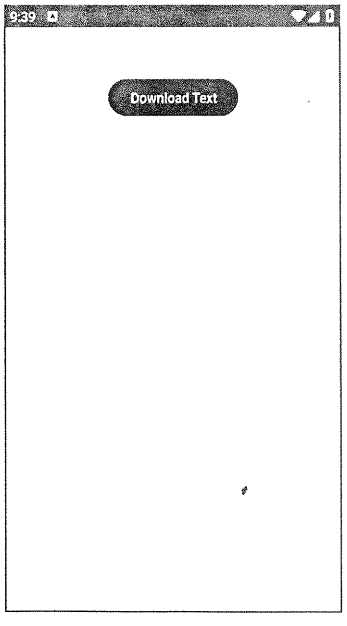
3. **AsyncTask for Downloading Text:**

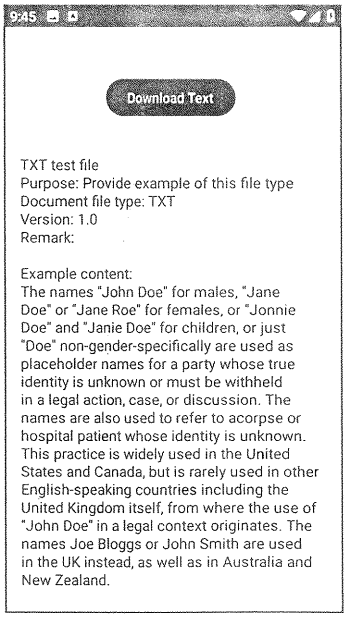
   The 'DownloadTextTask' is an inner class that extends 'AsyncTask'. It performs network operations on a background thread to prevent blocking the UI thread.

- **doInBackground():**
  - This method runs on a background thread and takes a URL string as a parameter.
  - It creates a 'URL' object from the given URL string and opens an 'HttpURLConnection' to connect to the URL.
  - A 'BufferedReader' is used to read the text content line by line from the URL's input stream.
  - The text content is appended to a 'StringBuilder'.
  - If an exception occurs (e.g., network error), it is caught, and an error message is returned.
  - The downloaded text is returned as a string.
- **onPostExecute():**
  - This method runs on the UI thread after the background operation is completed.
  - It checks if the result string starts with "Error:". If so, a toast message is shown to display the error.
  - If no error occurred, the downloaded text is set to the 'TextView' to display it to the user.

Output

When the app is run, pressing the "Download Text" button will start the download process. Once the text is downloaded, it will be displayed in the 'TextView'. If there is an error during the download, an error message will be shown using a 'Toast'.





).2.4 Downloading Image

this section, we will create an Android application that downloads an image from a specified ɪL and displays it in an 'ImageView'. This process involves setting up network permissions, ·rforming network operations on a background thread using 'AsyncTask', and updating the UI ɔm the background thread. This example demonstrates how to perform network operations in the .ckground and update the UI with the downloaded content, which is a common requirement in any Android applications.

ep 1: Create a New Android Project:

Create an Android project by selecting "Empty Views Activity", naming the project "ImageDownloader", and choosing Java as the language.

ep 2: Add Internet Permissions:

Ensure that the 'AndroidManifest.xml' file includes the necessary permission to access the internet. Add the following permission within the '<manifest>' tag:

```
<uses-permission android:name="android.permission.INTERNET" />
```

ep 3: Design the Layout:

Define the user interface in 'activity_main.xml'. This layout includes an 'ImageView' to display the downloaded image and a 'Button' to initiate the download.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button_download"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Download Image"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp" />

    <ImageView
        android:id="@+id/image_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/button_download"
        android:layout_marginTop="20dp"
        android:padding="16dp"
        android:scaleType="centerCrop" />
</RelativeLayout>
```

Step 4: Implement the MainActivity:

In 'MainActivity.java', implement the logic to download the image using 'AsyncTask' and display it in the 'ImageView'.

Code MainActivity.java

```
package com.example.imagedownloader;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import android.os.Bundle;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class MainActivity extends AppCompatActivity {

    private ImageView imageView;
    private Button buttonDownload;

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

        imageView = findViewById(R.id.image_view);
        buttonDownload = findViewById(R.id.button_download);

        buttonDownload.setOnClickListener(v -> new DownloadImageTask().
                    execute("https://www.w3schools.com/w3images/fjords.jpg"));
    }

    private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {

        @Override
        protected Bitmap doInBackground(String... urls) {
            String urlString = urls[0];
            Bitmap bitmap = null;

            try {
                URL url = new URL(urlString);
                HttpURLConnection urlConnection = (HttpURLConnection)
                                                  url.openConnection();
```

```
                InputStream inputStream = urlConnection.getInputStream();
                bitmap = BitmapFactory.decodeStream(inputStream);
                inputStream.close();
                urlConnection.disconnect();
            } catch (Exception e) {
                e.printStackTrace();
                return null;
            }

            return bitmap;
        }

        @Override
        protected void onPostExecute(Bitmap result) {
            if (result == null) {
                Toast.makeText(MainActivity.this, "Error downloading image",
                                                    Toast.LENGTH_SHORT).show();
            } else {
                imageView.setImageBitmap(result);
            }
        }
    }
}
```

Explanation

1. **Setup and Layout:**

   The layout file ('activity_main.xml') contains a 'Button' with the ID 'button_download' and an 'ImageView' with the ID 'image_view'. The 'Button' is used to trigger the download process, and the 'ImageView' displays the downloaded image.

2. **MainActivity:**

   - In the 'MainActivity' class, references to the 'Button' and 'ImageView' are obtained using 'findViewById()'.
   - An 'OnClickListener' is set on the 'Button' to start the 'AsyncTask' for downloading the image from a specified URL when clicked.
   - The URL 'https://www.w3schools.com/w3images/fjords.jpg' is used as an example. This URL points to an image that we want to download.

3. **AsyncTask for Downloading Image:**

   The 'DownloadImageTask' is an inner class that extends 'AsyncTask'. It performs network operations on a background thread to prevent blocking the UI thread.

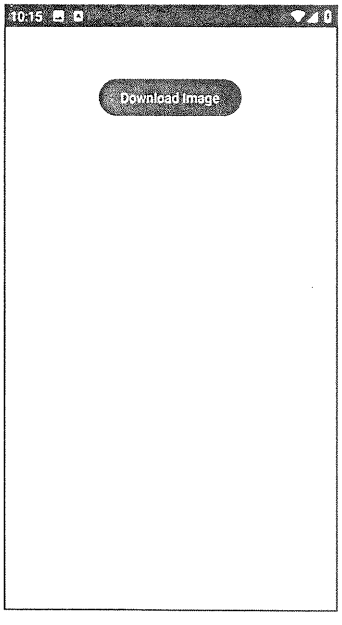
   - **doInBackground():**
     - This method runs on a background thread and takes a URL string as a parameter.
     - It creates a 'URL' object from the given URL string and opens an 'HttpURLConnection' to connect to the URL.

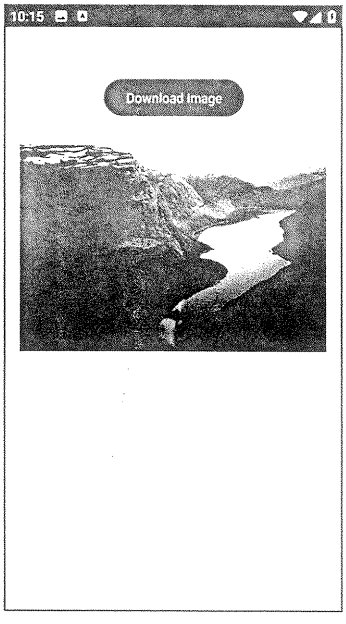
- An 'InputStream' is used to read the image data from the URL's input stream.
- The image data is decoded into a 'Bitmap' using 'BitmapFactory.decodeStream()'.
- If an exception occurs (e.g., network error), it is caught, and 'null' is returned.
- The downloaded 'Bitmap' is returned.

- **onPostExecute():**
  - This method runs on the UI thread after the background operation is completed.
  - It checks if the result is 'null'. If so, a toast message is shown to display an error.
  - If no error occurred, the downloaded 'Bitmap' is set to the 'ImageView' to display the image.

Output

When the app is run, pressing the "Download Image" button will start the download process. Once the image is downloaded, it will be displayed in the 'ImageView'. If there is an error during the download, an error message will be shown using a 'Toast'.





10.2.5 Consuming and Parsing XML Web Services in Android

In our previous examples, we have seen how to download and handle the response of text and images. Now, we will focus on a common use case where the web service response is in XML format. Parsing XML is essential for many Android applications that interact with web services. In this section, we will create an Android application that consumes a web service returning XML data and displays specific information.

What is XML Webservices?

:ML (eXtensible Markup Language) web services are a type of web service that uses XML as the ɔrmat for transmitting data between a server and a client. XML is a flexible, structured data format hat is both human-readable and machine-readable. It is widely used in web applications for data xchange due to its platform and language independence.

Characteristics of XML Web Services

1. **Platform Independence:** XML web services can be consumed by clients on different platforms, making them highly interoperable.
2. **Language Independence:** XML can be generated and parsed by many programming languages, allowing for easy integration across diverse systems.
3. **Structured Data:** XML provides a structured way to represent data using tags, making it easy to understand and manipulate.
4. **Self-describing Data:** XML includes metadata, which describes the structure and meaning of the data, enhancing readability and usability.
5. **Extensible:** XML allows for custom tags and attributes, enabling it to be extended to meet various needs without breaking existing functionality.

Common Uses of XML Web Services

1. **Data Exchange:** XML web services are commonly used to exchange data between a client (such as a web browser or mobile app) and a server.
2. **APIs:** Many web APIs use XML as the format for request and response payloads, especially in legacy systems.
3. **Configuration Files:** XML is often used for configuration files in applications because it is easy to read and modify.
4. **Data Storage:** Some databases and data storage systems use XML to store and retrieve data.

How XML Web Services Work

1. **Client Request:** The client (e.g., a web browser or mobile app) sends an HTTP request to the server, often using protocols like SOAP (Simple Object Access Protocol) which commonly employs XML.
2. **Server Response:** The server processes the request and sends back an HTTP response containing XML data.
3. **Data Parsing:** The client parses the XML data and uses it to update the user interface or perform other operations.

Benefits of Using XML Web Services

+ **Interoperability:** XML web services can be accessed by clients written in different programming languages and running on different platforms.

- **Standardization:** XML is a well-established standard for data exchange, supported by many tools and libraries.
- **Flexibility:** XML's extensible nature allows for the creation of complex data structures that can be tailored to specific needs.

Example

We will use the freely accessible URL `**https://www.floatrates.com/daily/usd.xml**` which provides exchange rate information. We will parse this XML data and extract the information for the "INR" (Indian Rupee) currency.

Examples Sample of the XML data from `https://www.floatrates.com/daily/usd.xml`

```
<item>
    <title>1 USD = 83.57469608 INR</title>
    <link>http://www.floatrates.com/usd/inr/</link>
    <description>1 U.S. Dollar = 83.57469608 Indian Rupee</description>
    <pubDate>Fri, 21 Jun 2024 23:56:55 GMT</pubDate>
    <baseCurrency>USD</baseCurrency>
    <baseName>U.S. Dollar</baseName>
    <targetCurrency>INR</targetCurrency>
    <targetName>Indian Rupee</targetName>
    <exchangeRate>83.57469608</exchangeRate>
    <inverseRate>0.01196534</inverseRate>
    <inverseDescription>1 Indian Rupee = 0.01196534 U.S. Dollar</inverseDescription>
</item>
<item>
    <title>1 USD = 159.00288601 JPY</title>
    <link>http://www.floatrates.com/usd/jpy/</link>
    <description>1 U.S. Dollar = 159.00288601 Japanese Yen</description>
    <pubDate>Fri, 21 Jun 2024 23:56:55 GMT</pubDate>
    <baseCurrency>USD</baseCurrency>
    <baseName>U.S. Dollar</baseName>
    <targetCurrency>JPY</targetCurrency>
    <targetName>Japanese Yen</targetName>
    <exchangeRate>159.00288601</exchangeRate>
    <inverseRate>0.00628919</inverseRate>
    <inverseDescription>1 Japanese Yen = 0.00628919 U.S. Dollar</inverseDescription>
</item>
```

We aim to create an Android app that fetches the XML data from the URL, parses the data, and extracts information specifically for the "INR" currency. The extracted data will then be displayed in a `TextView`.

How XML Parsing Works?

XML parsing involves reading XML data, understanding its structure, and converting it into a format that can be easily manipulated by a program. In Android, XML parsing is typically used to process

data from web services or local XML files. There are several approaches to XML parsing, each with its own advantages and disadvantage.

Types of XML Parsers :

1. **DOM (Document Object Model):**
   - Loads the entire XML document into memory and represents it as a tree structure of nodes.
   - Allows for easy and random access to any part of the document.
   - Suitable for small to medium-sized XML documents.
   - **Example libraries:** `javax.xml.parsers.DocumentBuilderFactory`, `org.w3c.dom.Document`, etc.
2. **SAX (Simple API for XML):**
   - Parses the XML document sequentially and triggers events (like start and end of an element) as it encounters them.
   - Does not load the entire document into memory, making it suitable for large XML files.
   - More complex to use compared to DOM because it requires handling various events.
   - **Example libraries:** `org.xml.sax`, `org.xml.sax.helpers.DefaultHandler`, etc.
3. **StAX (Streaming API for XML):**
   - Provides a cursor-based API for parsing XML documents.
   - Allows for both reading and writing XML documents.
   - Efficient and suitable for large documents.
   - **Example libraries:** `javax.xml.stream.XMLInputFactory`, `javax.xml.stream.XMLStreamReader`, etc.

For simplicity, we will focus on the DOM parser in this example. The DOM parser is easy to use and understand, making it suitable for beginners and small to medium-sized XML files.

Key Methods of DOM Parser:

1. **DocumentBuilderFactory :** This factory class provides a way to configure and obtain a `DocumentBuilder` instance. It defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.

   Usage: `DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();`
2. **DocumentBuilder:** This class defines the API to obtain `Document` objects from XML documents. Using a `DocumentBuilder`, you can parse an XML document and produce a DOM `Document` object.

   Usage: `DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();`
3. **Document:** Represents the entire XML document. It is the root of the document tree and provides methods to access and manipulate the XML data.

   Usage: `Document doc = dBuilder.parse(inputStream);`

4. **NodeList:** Represents an ordered collection of nodes. This interface provides the abstraction of an ordered collection of nodes, without defining or constraining how this collection is implemented.

   Usage: `NodeList nList = doc.getElementsByTagName("item");`

5. **Element:** Represents an element in the XML document. It inherits from `Node` and provides additional methods to access and manipulate the attributes and content of the element.

   Usage: `Element element = (Element) node;`

6. **Node:** Represents a single node in the document tree. Nodes can be of different types, such as elements, text nodes, comments, etc. The `Node` interface provides methods to access and manipulate the node and its children.

   Usage: `Node node = nList.item(i);`

How DOM Parsing Works?

The DOM parser reads an XML document and converts it into a tree structure where each node represents a part of the document (such as elements, attributes, text, etc.). This tree structure allows for easy navigation and manipulation of the XML data.

Key Steps in DOM Parsing:

1. **Initialize the Parser:**
   - Create an instance of `DocumentBuilderFactory`.
   - Obtain a `DocumentBuilder` instance from the factory.
2. **Parse the XML Document:**
   - Use the `DocumentBuilder` to parse the XML document and generate a `Document` object.
   - The `Document` object represents the entire XML document as a tree of nodes.
3. **Navigate the Document:**
   - Use methods provided by the `Document` and `Element` classes to navigate and manipulate the nodes in the document tree.
   - Access specific elements and attributes using methods like `getElementsByTagName`, `getAttribute`, etc.

Step-by-Step Implementation of Consuming XML Web Services

Step 1: Create a New Android Project:

Create an Android project by selecting "Empty Views Activity", naming the project "XMLDownloader", and choosing Java as the language.

Step 2: Add Internet Permissions:

Ensure that the 'AndroidManifest.xml' file includes the necessary permission to access the internet. Add the following permission within the '<manifest>' tag:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Step 3: Design the Layout:

Define the user interface in 'activity_main.xml'. This layout includes a 'TextView' to display the parsed XML data and a 'Button' to initiate the download.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button_download"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Download XML"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp" />

    <TextView
        android:id="@+id/text_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/button_download"
        android:layout_marginTop="20dp"
        android:padding="16dp"
        android:textSize="16sp" />
</RelativeLayout>
'''
```

Step 4: Implement the MainActivity:

In 'MainActivity.java', implement the logic to download and parse the XML data using 'AsyncTask', filter for the target currency "INR", and display the relevant information in the 'TextView'.

Code MainActivity.java

```
package com.example.xmldownloader;
import android.os.AsyncTask;
import android.os.Bundle;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
```

```
import org.w3c.dom.NodeList;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private Button buttonDownload;

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

        textView = findViewById(R.id.text_view);
        buttonDownload = findViewById(R.id.button_download);

        buttonDownload.setOnClickListener(v -> new DownloadXMLTask().
                        execute("https://www.floatrates.com/daily/usd.xml"));
    }

    private class DownloadXMLTask extends AsyncTask<String, Void, String> {

        @Override
        protected String doInBackground(String... urls) {
            String urlString = urls[0];
            StringBuilder result = new StringBuilder();
            try {
                URL url = new URL(urlString);
                HttpURLConnection urlConnection = (HttpURLConnection)
                                                    url.openConnection();
                InputStream inputStream = urlConnection.getInputStream();

                // Parse the XML data
                DocumentBuilderFactory dbFactory = DocumentBuilderFactory.
                                                        newInstance();
                DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
                Document doc = dBuilder.parse(inputStream);
                doc.getDocumentElement().normalize();
```

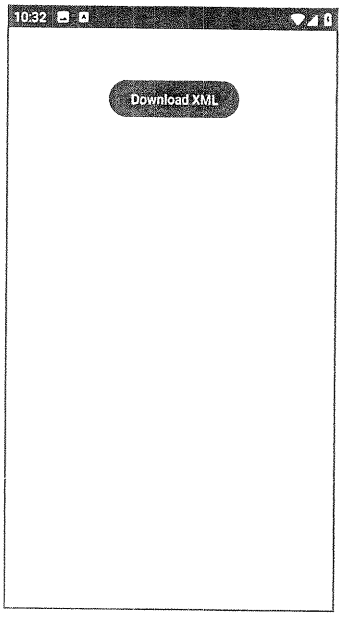
```
                NodeList nList = doc.getElementsByTagName("item");
                for (int i = 0; i < nList.getLength(); i++) {
                    Node node = nList.item(i);
                    if (node.getNodeType() == Node.ELEMENT_NODE) {
                        Element element = (Element) node;
                        String targetCurrency = getValue("targetCurrency", element);
                        if ("INR".equals(targetCurrency)) {
                            result.append("Title: ").append
                                (getValue("title", element)).append("\n");
                            result.append("Description: ").append
                                (getValue("description",element)).append("\n");
                            result.append("Exchange Rate: ").append
                             (getValue("exchangeRate", element)).append("\n");
                            result.append("Inverse Rate: ").append
                             (getValue("inverseRate", element)).append("\n");
                            break;
                        }
                    }
                }
                inputStream.close();
                urlConnection.disconnect();
            } catch (Exception e) {
                e.printStackTrace();
                return "Error: " + e.getMessage();
            }

            return result.toString();
        }

        private String getValue(String tag, Element element) {
            NodeList nodeList = element.getElementsByTagName(tag)
                                                    .item(0).getChildNodes();
            Node node = nodeList.item(0);
            return node.getNodeValue();
        }

        @Override
        protected void onPostExecute(String result) {
            if (result.startsWith("Error:")) {
              Toast.makeText(MainActivity.this, result, Toast.LENGTH_SHORT).show();
            } else {
                textView.setText(result);
            }
        }
    }
}
```

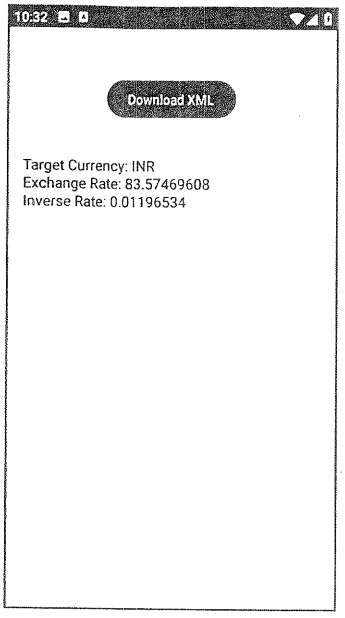
Explanation

1. **Setup and Layout:** The layout file ('activity_main.xml') contains a 'Button' with the ID 'button_download' and a 'TextView' with the ID 'text_view'. The 'Button' is used to trigger the download process, and the 'TextView' displays the parsed XML data.
2. **MainActivity:**
   - In the 'MainActivity' class, references to the 'Button' and 'TextView' are obtained using 'findViewById()'.
   - An 'OnClickListener' is set on the 'Button' to start the 'AsyncTask' for downloading the XML data from a specified URL when clicked.
   - The URL 'https://www.floatrates.com/daily/usd.xml' is used as an example. This URL returns XML data containing currency exchange rates.
3. **AsyncTask for Downloading and Parsing XML:** The 'DownloadXMLTask' is an inner class that extends 'AsyncTask'. It performs network operations on a background thread to prevent blocking the UI thread.
   - **doInBackground():**
     - This method runs on a background thread and takes a URL string as a parameter.
     - It creates a 'URL' object from the given URL string and opens an 'HttpURLConnection' to connect to the URL.
     - An 'InputStream' is used to read the XML data from the URL's input stream.
     - The XML data is parsed using the 'DocumentBuilderFactory' and 'DocumentBuilder' classes.
     - The parsed XML data is then processed to extract elements where the 'targetCurrency' is "INR". The data is appended to a 'StringBuilder'.
     - If an exception occurs (e.g., network error), it is caught, and an error message is returned.
     - The parsed XML data for "INR" is returned as a string.
   - **getValue():**
     - This method takes a tag name and an 'Element' object as parameters.
     - It retrieves the value of the specified tag from the given 'Element' object.
   - **onPostExecute():**
     - This method runs on the UI thread after the background operation is completed.
     - It checks if the result string starts with "Error:". If so, a toast message is shown to display the error.
     - If no error occurred, the parsed XML data is set to the 'TextView' to display it to the user.

Output

When the app is run, pressing the "Download XML" button will start the download and parsing process. Once the XML data is downloaded and parsed, only the information related to the "INR" currency will be displayed in the 'TextView'. If there is an error during the download, an error message will be shown using a 'Toast'.





10.2.6 Consuming JSON Web Services in Android

JSON (JavaScript Object Notation) web services are a type of web service that exchanges data in JSON format. JSON is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is widely used in web applications to transmit data between a server and a client.

Characteristics of JSON Web Services

- **Lightweight:** JSON is text-based and much lighter than XML, making it ideal for transmitting data over the internet.
- **Easy to Read and Write:** JSON syntax is easy to understand and use, making it a popular choice for data interchange.
- **Language Independent:** JSON is language agnostic and can be used with most programming languages, including JavaScript, Python, Java, and many others.
- **Self-describing:** JSON data is self-descriptive, making it easier to understand and use.

Structure of JSON

JSON data is represented as key-value pairs. It can contain multiple nested objects and arrays.

- **Objects:** Represented by curly braces `{}` and contain key-value pairs.
- **Arrays:** Represented by square brackets `[]` and contain a list of values.

Example JSON Data

```
{
  "weather": [
    {
      "id": 800,
      "main": "Clear",
      "description": "clear sky",
      "icon": "01d"
    }
  ],
  "main": {
    "temp": 300.15,
    "pressure": 1013,
    "humidity": 50,
    "temp_min": 300.15,
    "temp_max": 300.15
  },
  "wind": {
    "speed": 4.1,
    "deg": 80
  },
  "name": "Bangalore"
}
```

Explanation

- The JSON data consists of various elements that provide detailed weather information. The weather array includes elements describing weather conditions such as ID, main parameters (like Rain or Snow), a description, and an icon ID.
- The main object contains key weather data like temperature (in Kelvin), atmospheric pressure (in hPa), humidity (in percentage), and minimum and maximum temperatures (in Kelvin).
- The wind object provides wind data, including speed and direction (in degrees). The name field specifies the name of the city. This structured format allows easy extraction and use of specific weather details from the JSON data.

Common Uses of JSON Web Services

1. **Data Exchange:** JSON web services are commonly used to exchange data between a client (such as a web browser or mobile app) and a server.
2. **APIs:** Many web APIs use JSON as the format for request and response payloads.
3. **Configuration Files:** JSON is often used for configuration files because it is easy to read and modify.
4. **Data Storage:** Some databases, like MongoDB, use JSON-like formats for storing data.

Examples of Free JSON Web Services

1. **OpenWeatherMap:** Provides weather data for cities worldwide.
   - **URL:** `https://api.openweathermap.org/data/2.5/weather?q=Bangalore&appid=YOUR_API_KEY`
   - **Data:** Current weather conditions, temperature, humidity, etc.
2. **CoinGecko API:** Provides cryptocurrency prices and market data.
   - **URL:** `https://api.coingecko.com/api/v3/simple/price?ids=bitcoin&vs_currencies=usd`
   - **Data:** Current price of Bitcoin in USD.
3. **JSONPlaceholder:** Provides fake online REST API for testing and prototyping.
   - **URL:** `https://jsonplaceholder.typicode.com/posts`
   - **Data:** List of sample posts, user information, comments, etc.

How JSON Web Services Work

1. **Client Request:** The client (e.g., a web browser or mobile app) sends an HTTP request to the server.
2. **Server Response:** The server processes the request and sends back an HTTP response containing JSON data.
3. **Data Parsing:** The client parses the JSON data and uses it to update the user interface or perform other operations.

Benefits of Using JSON Web Services

+ **Efficiency:** JSON's lightweight nature reduces the amount of data transmitted, improving performance and efficiency.
+ **Ease of Use:** JSON's simplicity and readability make it easy for developers to work with.
+ **Compatibility:** JSON's language-independent format ensures compatibility across different programming languages and platforms.

Parsing JSON

arsing JSON in Android can be achieved using various methods, such as the `org.json` package or ıird-party libraries like `Gson` and `Jackson`. For simplicity, the `org.json` package will be used in ıis example.

ey Classes and Methods

1. **JSONObject:**

   Description: Represents a JSON object, which is a collection of key-value pairs. It is used to parse and retrieve data from JSON strings where each key is a string, and each value can be a string, number, boolean, JSONObject, or JSONArray.

   Key Methods:
   - **`getString(String key)`:** Retrieves a string value associated with a key.
   - **`getInt(String key)`:** Retrieves an integer value associated with a key.

- **`getDouble(String key)`:** Retrieves a double value associated with a key.
- **`getJSONObject(String key)`:** Retrieves a nested JSONObject associated with a key.
- **`getJSONArray(String key)`:** Retrieves a JSONArray associated with a key.
- **`optString(String key)`:** Retrieves a string value associated with a key, or returns an empty string if the key doesn't exist.

2. **JSONArray:**

   Description: Represents a JSON array, which is an ordered collection of values. It is used to parse and retrieve data from JSON strings where the array elements can be any type, including JSONObjects and other JSONArrays.

   Key Methods:

   - **`getJSONObject(int index)`:** Retrieves a JSONObject at the specified index.
   - **`getString(int index)`:** Retrieves a string value at the specified index.
   - **`getInt(int index)`:** Retrieves an integer value at the specified index.
   - **`getDouble(int index)`:** Retrieves a double value at the specified index.
   - **`length()`:** Returns the number of elements in the JSONArray.
   - **`optJSONObject(int index)`:** Retrieves a JSONObject at the specified index, or returns null if the index is out of range.

3. **JSONTokener:**

   Description: A utility class used to parse a JSON string into a JSONObject or JSONArray. It processes the text from which the JSON data is extracted and creates appropriate JSON structures.

   Key Methods:

   - **`JSONTokener(String jsonString)`:** Constructor that initializes the tokener with the JSON string to be parsed.
   - **`nextValue()`:** Parses the next value from the JSON string, which can be a JSONObject, JSONArray, String, Boolean, Integer, Long, Double, or null.

Step-by-Step Implementation of Consuming JSON Web Services

Step 1: Create a New Android Project:

Create an Android project by selecting "Empty Views Activity", naming the project "JSONDownloader", and choosing Java as the language.

Step 2: Add Internet Permissions:

Ensure that the `AndroidManifest.xml` file includes the necessary permission to access the internet. Add the following permission within the `<manifest>` tag:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Step 3: Design the Layout:

Define the user interface in `activity_main.xml`. This layout includes a `TextView` to display the parsed JSON data and a `Button` to initiate the download.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button_download"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Download JSON"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp" />

    <TextView
        android:id="@+id/text_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/button_download"
        android:layout_marginTop="20dp"
        android:padding="16dp"
        android:textSize="16sp" />
</RelativeLayout>
```

tep 4: Get an API Key:

To use the OpenWeatherMap API, sign up and get an API key:

1. Go to the [OpenWeatherMap Sign Up Page](https://home.openweathermap.org/users/sign_up).
2. Create an account by providing an email and creating a password.
3. Once the email is verified and logged in, navigate to the API keys section.
4. Generate a new API key and copy it for use in the application.

tep 5: Implement the MainActivity:

In `MainActivity.java`, implement the logic to download and parse the JSON data using `AsyncTask` and display the relevant information in the `TextView`.

Code MainActivity.java

```
package com.example.jsondownloader;
import android.os.AsyncTask;
import android.os.Bundle;
```

```
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
import androidx.appcompat.app.AppCompatActivity;
import org.json.JSONArray;
import org.json.JSONObject;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class MainActivity extends AppCompatActivity {

    private TextView textView;
    private Button buttonDownload;

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

        textView = findViewById(R.id.text_view);
        buttonDownload = findViewById(R.id.button_download);

        buttonDownload.setOnClickListener(v -> new DownloadJSONTask()
         .execute("https://api.openweathermap.org/data/2.5/
                  weather?q=Bangalore&appid=YOUR_API_KEY"));
    }
    private class DownloadJSONTask extends AsyncTask<String, Void, String> {
        @Override
        protected String doInBackground(String... urls) {
            String urlString = urls[0];
            StringBuilder result = new StringBuilder();
            try {
                    URL url = new URL(urlString);
                    HttpURLConnection urlConnection = (HttpURLConnection)
                                                      url.openConnection();
                    BufferedReader reader = new BufferedReader(new
                          InputStreamReader(urlConnection.getInputStream()));
                    String line;
                    while ((line = reader.readLine()) != null) {
                       result.append(line);
                    }
```

```
                reader.close();
                urlConnection.disconnect();
            } catch (Exception e) {
                e.printStackTrace();
                return "Error: " + e.getMessage();
            }

            return result.toString();
        }

        @Override
        protected void onPostExecute(String result) {

            if (result.startsWith("Error:")) {
                Toast.makeText(MainActivity.this, result,
                                                    Toast.LENGTH_SHORT).show();
            } else {
                try {
                    JSONObject jsonObject = new JSONObject(result);
                    JSONArray weatherArray = jsonObject.getJSONArray("weather");
                    JSONObject weather = weatherArray.getJSONObject(0);
                    String description = weather.getString("description");

                    JSONObject main = jsonObject.getJSONObject("main");
                    double temperatureKelvin = main.getDouble("temp");
                    double temperatureCelsius = temperatureKelvin - 273.15;

                    String cityName = jsonObject.getString("name");

                    String displayText = "City: " + cityName + "\n" +
                    "Temperature: " + String.format("%.2f", temperatureCelsius)
                    + "°C\n" + "Weather: " + description;

                    textView.setText(displayText);
                } catch (Exception e) {
                    e.printStackTrace();
                   Toast.makeText(MainActivity.this, "Error parsing JSON data",
                                                    Toast.LENGTH_SHORT).show();
                }
            }
        }
    }
}
```

Explanation

1. **Setup and Layout:**

   The layout file (`activity_main.xml`) contains a `Button` with the ID `button_download` and a `TextView` with the ID `text_view`. The `Button` is used to trigger the download process, and the `TextView` displays the parsed JSON data.

2. **MainActivity:**

   - In the `MainActivity` class, references to the `Button` and `TextView` are obtained using `findViewById()`.
   - An `OnClickListener` is set on the `Button` to start the `AsyncTask` for downloading the JSON data from a specified URL when clicked.
   - The URL `https://api.openweathermap.org/data/2.5/weather?q=Bangalore&appid=YOUR_API_KEY` is used as an example. This URL returns JSON data containing weather information for Bangalore.

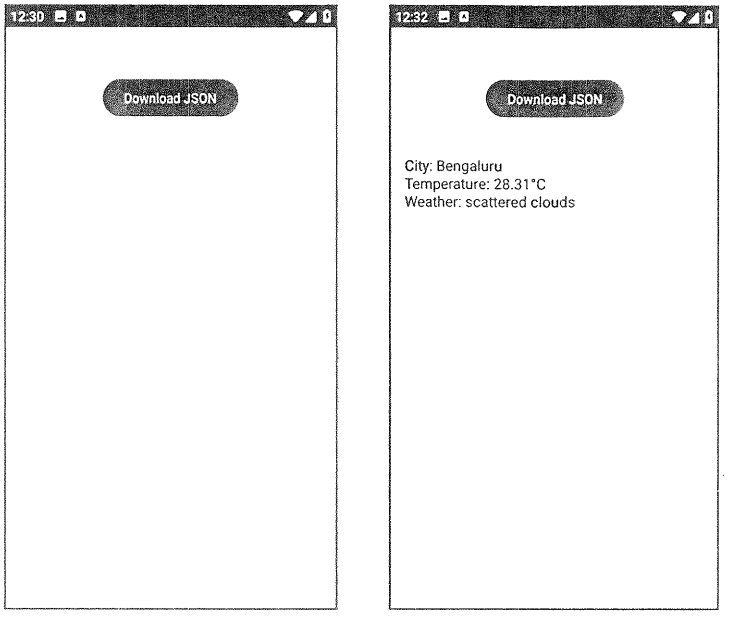
3. **AsyncTask for Downloading and Parsing JSON:**

   The `DownloadJSONTask` is an inner class that extends `AsyncTask`. It performs network operations on a background thread to prevent blocking the UI thread.

   - **doInBackground():**
     - This method runs on a background thread and takes a URL string as a parameter.
     - It creates a `URL` object from the given URL string and opens an `HttpURLConnection` to connect to the URL.
     - A `BufferedReader` is used to read the JSON data from the URL's input stream line by line.
     - The JSON data is appended to a `StringBuilder`.
     - If an exception occurs (e.g., network error), it is caught, and an error message is returned.
     - The downloaded JSON data is returned as a string.
   - **onPostExecute():**
     - This method runs on the UI thread after the background operation is completed.
     - It checks if the result string starts with "Error:". If so, a toast message is shown to display the error.
     - If no error occurred, the JSON data is parsed using the `JSONObject` and `JSONArray` classes to extract specific information (e.g., weather description, temperature in Celsius, city name).
     - The extracted data is formatted into a string and set to the `TextView` to display it to the user.

Output

When the app is run, pressing the "Download JSON" button will start the download and parsing process. Once the JSON data is downloaded and parsed, the weather information for Bangalore, including the temperature in Celsius, will be displayed in the `TextView`. If there is an error during the download or parsing process, an error message will be shown using a `Toast`.



10.3 Creating Your Own Services and Binding Activities to Service

Services in Android are components that facilitate background processing without requiring a user interface. They allow applications to perform long-running operations while the user is not interacting with the app. Services can run indefinitely in the background and can also be bound to activities to enable communication between the activity and the service. Services play a crucial role in enhancing the functionality, performance, and user experience of mobile applications.

Importance of Creating Services

1. **Background Processing:** Services enable apps to execute tasks in the background, such as downloading files, processing data, or handling network requests, without interrupting the user experience.
2. **Improved Performance:** By offloading intensive tasks to services, apps can maintain responsiveness and performance to ensure a smooth user interaction even during resource-intensive operations.
3. **Enhanced User Experience:** Services contribute to a seamless user experience by allowing apps to run background tasks efficiently, provide timely notifications, and deliver real-time updates without disrupting the user's workflow.
4. **Optimized Battery Life:** Well-designed services help optimize battery consumption by efficiently managing resources and scheduling tasks to ensure that background operations do not drain the device's battery excessively.

Example	Example Scenarios
	1. Music Streaming App: A music streaming app utilizes a background service to play music continuously, even when the app is running in the background or the device is locked. This ensures uninterrupted music playback for users. 2. Messaging App: A messaging app employs a service to receive incoming messages in real-time, update message notifications, and synchronize chat data with the server, providing users with instant message delivery and notifications. 3. Location Tracking App: A location tracking app uses a background service to track the user's location periodically, update location data, and trigger location-based alerts or notifications, enhancing the app's geolocation features.

How Services Work?

1. **Service Creation:** Developers define and implement services by extending the Service class in Android. Services are registered in the app manifest file and can be started, stopped, or bound to activities as needed.
2. **Background Execution:** Services run independently of the app's user interface thread, allowing them to perform tasks in the background without blocking the main UI thread. This ensures that the app remains responsive and functional while the service executes its operations.
3. **Communication:** Services can communicate with other app components, such as activities, broadcast receivers, or content providers, using intents, callbacks, or event listeners. This enables seamless data exchange and coordination between different parts of the app.
4. **Lifecycle Management:** Services have their own lifecycle methods, such as onCreate(), onStartCommand(), and onDestroy(), to manage initialization, task execution, and cleanup. Developers must handle service lifecycle events to ensure proper operation and resource management.

Step by Step Implementation of Creating a Service and Binding Activity to Service

Services** in Android are components that facilitate background processing without requiring a user interface. They allow applications to perform long-running operations while the user is not interacting with the app. Services can run indefinitely in the background and can also be bound to activities to enable communication between the activity and the service.

Imagine developing an application that counts numbers in the background. The user should be able to start, stop, and reset the counter, and see the current count on the main activity.

The goal is to create an Android app that:

1. Starts a service to count numbers in the background.
2. Allows the main activity to bind to this service to control and monitor the counter.
3. Provides a user interface to start, stop, and reset the counter.

Step 1: Create a New Android Project

Create an Android project by selecting "Empty Activity", naming the project "CounterServiceDemo", and choosing Java as the language.

Step 2: Define the User Interface

Define the user interface in `activity_main.xml`:

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/start_service_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start Service"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp" />

    <Button
        android:id="@+id/bind_service_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Bind Service"
        android:layout_below="@id/start_service_button"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp" />

    <Button
        android:id="@+id/unbind_service_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Unbind Service"
        android:layout_below="@id/bind_service_button"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp" />

    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start"
        android:layout_below="@id/unbind_service_button"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp" />
```

```
<Button
        android:id="@+id/stop_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Stop"
        android:layout_below="@id/start_button"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp" />

    <Button
        android:id="@+id/reset_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Reset"
        android:layout_below="@id/stop_button"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp" />

    <TextView
        android:id="@+id/count_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/reset_button"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp"
        android:textSize="16sp" />
</RelativeLayout>
```

Step 3: Define the Counter Service

Create a new service class `CounterService`:

Code CounterService.java

```
package com.example.counterserviceexample;

import android.app.Service;
import android.content.Intent;
import android.os.Binder;
import android.os.Handler;
import android.os.IBinder;
import android.os.Looper;
import android.widget.Toast;
```

```
public class CounterService extends Service {
    private final IBinder binder = new LocalBinder();
    private int count = 0;
    public boolean isCounting = false;
    private Handler handler = new Handler(Looper.getMainLooper());
    private Runnable counterRunnable = new Runnable() {
        @Override
        public void run() {
            if (isCounting) {
                count++;
               handler.postDelayed(this, 1000); // Increment count every second
            }
        }
    };

    public class LocalBinder extends Binder {
        CounterService getService() {
            return CounterService.this;
        }
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        startCounting();
        return START_STICKY;
    }

    @Override
    public IBinder onBind(Intent intent) {
        return binder;
    }

    @Override
    public void onDestroy() {
        stopCounting();
        super.onDestroy();
    }

   public void startCounting() {
        if (!isCounting) {
            isCounting = true;
            handler.post(counterRunnable);
        }
    }
```

```
public void startCounting() {
        if (!isCounting) {
            isCounting = true;
            handler.post(counterRunnable);
        }
    }

    public void stopCounting() {
        isCounting = false;
        handler.removeCallbacks(counterRunnable);
    }

    public void resetCount() {
        count = 0;
    }

    public int getCount() {
        return count;
    }
}
```

Step 4: Declare the Service in the Manifest

Add the service to the `AndroidManifest.xml` file (inside application tag)

```
<service android:name=".CounterService" />
```

Step 5: Create the Main Activity to Bind to the Service

Update the `MainActivity` to bind to the service and interact with it:

Code MainActivity.java

```
package com.example.counterserserviceexample;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.Handler;
import android.os.IBinder;
import android.os.Looper;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private CounterService counterService;
    private boolean isBound = false;
    private TextView countTextView;
```

```
private ServiceConnection serviceConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        CounterService.LocalBinder binder = (CounterService.LocalBinder) service;
        counterService = binder.getService();
        isBound = true;
        updateCount();
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
        isBound = false;
    }
};
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    countTextView = findViewById(R.id.count_text_view);

    findViewById(R.id.start_service_button).setOnClickListener(v -> {
       Intent intent = new Intent(MainActivity.this, CounterService.class);
        startService(intent);
    });

    findViewById(R.id.bind_service_button).setOnClickListener(v -> {
       Intent intent = new Intent(MainActivity.this, CounterService.class);
        bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE);
    });

    findViewById(R.id.unbind_service_button).setOnClickListener(v -> {
        if (isBound) {
            unbindService(serviceConnection);
            isBound = false;
        }
    });

     findViewById(R.id.start_button).setOnClickListener(v -> {
        if (isBound) {
            counterService.startCounting();
            updateCountPeriodically();
        }
    });
```

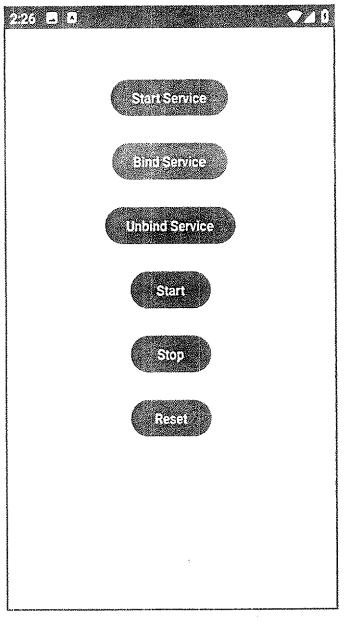
```
        findViewById(R.id.stop_button).setOnClickListener(v -> {
            if (isBound) {
                counterService.stopCounting();
            }
        });

        findViewById(R.id.reset_button).setOnClickListener(v -> {
            if (isBound) {
                counterService.resetCount();
                updateCount();
            }
        });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (isBound) {
            unbindService(serviceConnection);
            isBound = false;
        }
    }

    private void updateCount() {
        if (isBound) {
            countTextView.setText(String.valueOf(counterService.getCount()));
        }
    }

    private void updateCountPeriodically() {
        Handler uiHandler = new Handler(Looper.getMainLooper());
        Runnable updateTask = new Runnable() {
            @Override
            public void run() {
                if (isBound && counterService != null
                                                && counterService.isCounting) {
                    updateCount();
                    uiHandler.postDelayed(this, 1000);
                }
            }
        };
        uiHandler.post(updateTask);
    }
}
```

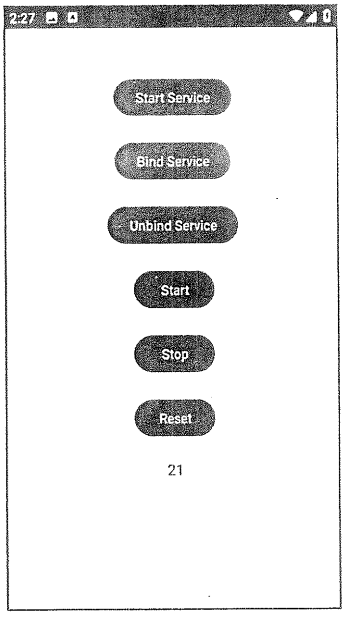
Explanation :

1. **CounterService.java :** This service increments a counter every second when counting is started. It uses a `Handler` to post a `Runnable` that increments the counter.
2. **MainActivity.java :** The activity binds to the service, allowing the user to start, stop, and reset the counter. The `updateCountPeriodically()` method ensures that the counter value in the `TextView` is updated every second while counting.
3. **activity_main.xml :** The layout file defines buttons for starting the service, binding the service, unbinding the service, starting the counter, stopping the counter, and resetting the counter, as well as a `TextView` to display the current count.

Output

When the user opens the app and presses "Start Service", the service is started, and counting begins n the background. By pressing "Bind Service", the activity binds to the service, allowing interaction with it. When the "Start" button is pressed, the counter starts incrementing, and the TextView updates every second to reflect the current count. Pressing "Stop" pauses the counter, keeping the TextView static. The "Reset" button resets the counter to zero, updating the TextView accordingly. Unbinding the service with the "Unbind Service" button detaches the activity from the service.





1. **Starting the Service:**
   - When the user presses the "Start Service" button, the `CounterService` is started by calling `startService(intent)` in the `MainActivity`.
   - The `onStartCommand()` method of the `CounterService` is called, which starts the counting process by posting the `counterRunnable` to the handler.

2. **Binding the Service:**
   - When the user presses the "Bind Service" button, the `bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE)` method is called.
   - The `onServiceConnected()` method of the `ServiceConnection` is triggered, providing the main activity with an instance of the `CounterService`. This allows the activity to call methods on the service to control the counter.
3. **Controlling the Counter:**
   - **Start Counting :** The `startCounting()` method of the `CounterService` sets `isCounting` to true and posts the `counterRunnable` to the handler, which increments the counter every second.
   - **Stop Counting :** The `stopCounting()` method sets `isCounting` to false and removes the `counterRunnable` from the handler, pausing the counting process.
   - **Reset Counter :** The `resetCount()` method resets the counter to zero. The `updateCount()` method in the `MainActivity` updates the `TextView` to reflect the current count.
4. **Unbinding the Service :**
   - When the user presses the "UnbindService" button, the `unbindService(serviceConnection)` method is called, which triggers the `onServiceDisconnected()` method of the `ServiceConnection`, setting `isBound` to false.

10.4 Understanding Threads

A thread in programming is the smallest unit of processing that can be performed in an OS. It represents a separate path of execution for a segment of code. In Android, the main thread (also known as the UI thread) is responsible for handling user interface (UI) updates and user interactions. To ensure smooth and responsive user experiences, it is crucial to offload long-running operations, such as network requests, file I/O, and complex computations, from the main thread to background threads. This helps prevent the UI from becoming unresponsive and avoids the dreaded "Application Not Responding" (ANR) errors.

Why Threads Are Important

1. **Responsiveness :** Offloading heavy operations to background threads keeps the UI thread free to handle user interactions, resulting in a more responsive application.
2. **Parallelism :** By using multiple threads, tasks can be executed concurrently, improving the overall performance and efficiency of the application.
3. **Avoiding ANR :** Performing long-running tasks on the main thread can cause the application to become unresponsive, leading to ANR errors. Using background threads helps avoid this issue.

Key Concepts and Classes

1. **Thread :** The basic unit of execution in Java. A `Thread` represents a single path of execution in a program.
2. **Handler :** A `Handler` allows to send and process `Message` and `Runnable` objects associated

with a thread's `MessageQueue`.

3. **Looper :** A `Looper` runs a message loop for a thread. It processes messages and runnables from the message queue.
4. **AsyncTask :** A helper class for performing background operations and publishing results on the UI thread without directly handling threads or handlers.
5. **ExecutorService :** A higher-level replacement for working with individual threads. It provides a pool of threads and various scheduling capabilities.

Step-by-Step Implementation of Using a Background Thread to Perform a Task

Let's create an example to demonstrate how to use a background thread to perform a long-running task (e.g., downloading a file) and update the UI upon completion.

Step 1: Create a New Android Project

Create an Android project by selecting "Empty Activity", naming the project "ThreadExample", and choosing Java as the language.

Step 2: Define the User Interface

Define the user interface in `activity_main.xml`:

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start Task"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp" />

    <TextView
        android:id="@+id/status_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/start_button"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="20dp"
        android:text="Status: Idle"
        android:textSize="16sp" />
</RelativeLayout>
```

Step 3: Implement the Main Activity

Update the `MainActivity` to handle the button click, start a background thread, and update the UI:

Code MainActivity.java

```
package com.example.threadexample;
import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.widget.Button;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

    private TextView statusTextView;
    private Handler uiHandler = new Handler(Looper.getMainLooper());

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

        statusTextView = findViewById(R.id.status_text_view);

        Button startButton = findViewById(R.id.start_button);
        startButton.setOnClickListener(v -> {
            statusTextView.setText("Status: Running");
            new Thread(new Task()).start();
        });
    }

    private class Task implements Runnable {
        @Override
        public void run() {
            // Simulate a long-running task
            try {
                Thread.sleep(5000); // Simulate a 5-second task
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // Update the UI on the main thread
            uiHandler.post(() -> statusTextView.setText("Status: Completed"));
        }
    }
}
```

Explanation

1. **activity_main.xml** : Defines a simple user interface with a `Button` to start the task and a `TextView` to display the status.
2. **MainActivity.java :**
   - **'Handler uiHandler`:** A `Handler` associated with the main thread's `Looper`. It is used to post tasks back to the main thread.
   - **`onCreate()`:** Sets up the button click listener to start the background thread.
   - **`Task` class:** Implements `Runnable` and simulates a long-running task. After the task is completed, it posts a runnable to the `Handler` to update the UI on the main thread.

Example User Interaction

1. **Start Task :** The user opens the app and presses the "Start Task" button. The status text changes to "Status: Running".
2. **Background Task :** The app starts a new background thread that simulates a 5-second task.
3. **Task Completion :** Once the task is completed, the status text changes to "Status: Completed".

.0.5 Local Data Storage in Android

.ocal data storage in Android allows applications to persist data across sessions, providing a seamless nd consistent user experience. Android offers several methods for storing data locally, each suitable or different types of data and use cases. The primary methods include using Shared Preferences, the nternal file system, and external storage such as an SD card.

0.5.1 Reading and Writing Local Data Using Shared Preferences

hared Preferences is a simple key-value storage mechanism provided by Android for saving small mounts of primitive data. It is ideal for storing user preferences, application settings, and other mall data that needs to persist across user sessions. Shared Preferences is private to the application nd is deleted when the app is uninstalled. This method is efficient and easy to implement, making it popular choice for many apps.

tefer Chapter 5, Section 5.9.1 Saving and Loading User Preference for Detailed Explanation and Examples.

0.5.2 Accessing Internal File System

'he internal file system in Android provides a secure and private storage area for each application. 'iles stored in the internal storage are private to the app and are deleted when the app is uninstalled. 'his makes it a suitable place for storing sensitive data, cache files, and any other data that should not e accessible by other apps or users. Internal storage does not require special permissions, and it is ccessible using standard file I/O operations.

Refer Chapter 5, 5.9.2 Persisting Data to File for Detailed Explanation and Examples.

10.5.3 Accessing SD Card

External storage, such as an SD card, provides shared storage that can be used to store data accessible to other apps and the user. This type of storage is useful for saving files that the user might want to access with other apps, such as photos, videos, and documents. However, files in external storage are less secure since they can be accessed by other apps and the user. To read from and write to the SD card, an app must request the appropriate permissions in the AndroidManifest.xml file. External storage persists beyond the app's lifecycle unless manually deleted by the user or the app.

Refer Chapter 5, 5.9.2 Persisting Data to File for Detailed Explanation and Examples.

10.6 Preparing App for Publishing

Refer Chapter 2, Section 2.9 Publishing your Android Application for Detailed Explanation.

10.7 Deploying APK Files

Refer Chapter 2, Section 2.9 Publishing your Android Application for Detailed Explanation.

10.8 Uploading in Market

Refer Chapter 2, Section 2.9 Publishing your Android Application for Detailed Explanation.

10.9 Review Questions

Section - A

1. Define web services.
2. What is the role of HTTP in consuming webservices?
3. What is JSON and why is it popular for data interchange?
4. Why is the `HttpURLConnection` class used in Android?
5. Why is the `AsyncTask` class used in Android?
6. Mention two benefits of using web services in mobile apps.
7. What do you mean by XML Web Services.
8. What do you mean by JSON Web Services.
9. Write any two Common Uses of XML Web Services.
10. How XML Web Services Work?
11. Write any two Common Uses of JSON Web Services.
12. How JSON Web Services Work?
13. Mention two benefits of using JSON web services.
14. What is UI thread?
15. Why Threads Are Important?

16. What is the primary role of services in Android?
17. Name two lifecycle methods of an Android service.
18. Why is it important to use background threads in Android?

Section - B

1. Explain the use of HttpURLConnection Class and Mention Key Methods.
2. Explain the use of AsyncTask Class and Mention Key Methods of AsyncTask Class.
3. Explain the Importance of Web Services in Mobile Apps.
4. Outline the basic steps involved in consuming web services using HTTP in an Android application.
5. What is XML Webservices? Mention the Characteristics of XML Web Services.
6. How XML Parsing Works? Explain Key Methods of DOM Parser.
7. How DOM Parsing Works?
8. What is JSON Webservices? Mention the Characteristics of JSON Web Services.
9. Explain the Structure of JSON with an example.
10. How Parsing in JSON Work? Explain Key Classes and Methods.
11. Explain the Importance of Creating Services. Give an example of a service.
12. How Services Work in Android?
13. Explain the process of binding an activity to a service in Android.

Section - C

1. Provide a detailed explanation of how to download and display text content from a web service.
2. Provide a detailed explanation of how to download and display image content from a web service.
3. Explain the Step-by-Step Implementation of Consuming XML Web Services.
4. Explain the Step-by-Step Implementation of Consuming JSON Web Services
5. Explain the Step by Step Implementation of Creating a Service and Binding Activity to Service.
6. Explain the Step-by-Step Implementation of Using a Background Thread to Perform a Task.

APPENDIX

A

LAB PROGRAMS

Program 1	Creating "Hello world" Application.
Refer Chapter 2, Section : 2.6 - Your First Android Application - Hello World	

Program 2	Creating an Application that Displays Message based on the Screen Orientation.
Refer Chapter 4, Section : 4.5.3 - Detecting Orientation Changes	

Program 3	Create an application to develop Login window using UI controls.
Refer Chapter 5, Section : 5.10.1 - Developing a Login Window Using UI Controls	

Program 4	Create an application to implement new activity using explicit intent and implicit intent and content provider.
Refer Chapter 3: Section : 3.7 Implementing a New Activity Using Explicit Intent, Implicit Intent and Content Provider	

Program 5	Create an application that displays custom designed Opening Screen.
Refer Chapter 5, Section : 5.10.4 - Creating an Application with a Custom Designed Opening Screen	

Program 6	Create an UI with all views.
Refer Chapter 5, Section : 5.10.3 - Creating an UI with All Views	

Program 7	Create menu in Application.
Refer Chapter 4, Section : 4.6.2 - Adding Action Items to the Action Bar	

Program 8	Read / write the Local data.
Refer Chapter 5: Section : 5.9.1 - Saving and Loading User Preferences (Go to Example Implementaion)	

rogram 9	Create / Read / Write data with database (SQLite).
fer Chapter 8: Section : 8.3 - Hello, Database o to Step-by-Step Implementation of Creating and Using a Database in Android)	

ogram 10	Create an application to send SMS and receive SMS.
fer Chapter 7: Section : 7.1.2 - Step-by-Step Implementation of Sending SMS	

ogram 11	Create an application to send an E-mail.
fer Chapter 7: Section : 7.2.3 - Step-by-Step Implementation of Sending an Emai	

ogram 12	Display Map based on the Current/given location.
fer Chapter 7: Section : 7.3.2 - Getting Location Data o to Step-by-Step Implementation of Getting Location Data Using LocationManager)	

ogram 13	Create a sample application with login module (check user name and password) On successful login change Textview "Login Successful". On login fail alert using Toast "login fail".
fer Chapter 5, Section : 5.10.2 - Developing Login Window and Validating User Credentials	

ogram 14	Learn to Deploy Android Applications.
fer Chapter 2: Section : 2.9 Publishing your Android Application.	

APPENDIX

MODEL QUESTION PAPERS B

Model Question Paper - 1

Time : 2 ½ Hours | Max. Marks : 60

Instructions : Answer All Sections

Section-A

I. Answer any Four questions. Each question carries Two marks (4 X 2 = 8)

1. What is Mobile Application?
2. What is Android SDK?
3. Define View in Android.
4. What is a WebView in Android?
5. What does "designing by declaration" refer to in Android development?
6. What is a Content Provider in Android?

Section-B

II. Answer any Four question. Each question carries Five marks (4 x 5 = 20)

7. Explain the Brief History of Mobile Technologies.
8. What are Intent Filters? Explain the components of Intent Filter
9. Explain DatePicker along with its attributes and features. Give an example.
10. How does a ScrollView help in handling different screen sizes in Android? Explain with an example.
11. How can an application pre-fill email details like recipient, subject, and body using an Intent? Provide an example
12. Explain the process of binding an activity to a service in Android.

Section-C

III. Answer any Four questions. Each question carries Eight marks (4 X 8 = 32)

13. a) What is an Android Studio? Write its features.
    b) What is the use of Android Manifest File? Explain the components (or) structure of Android Manifest File.
14. a) How to create a style and theme in android? Explain with an example.
    b) Discuss the process of adding fragments dynamically in an Android application.
15. List and briefly describe common views used in Android applications with their XML definitions.
16. a) Explain Anchoring Views to Adapt Display Orientation
    b) What Happens to an Activity's State When Orientation Changes?
17. Explain the complete workflow of creating, reading, updating, and deleting data in a SQLite database with code examples.
18. Explain the Step-by-Step Implementation of Consuming XML Web Services.

◎◎◎◎

Model Question Paper - 2

Time : 2 ½ Hours

Max. Marks : 60

Instructions : Answer All Sections

Section-A

I. Answer any Four questions. Each question carries Two marks **(4 X 2 = 8)**

1. What is Android Emulator?
2. Define a Fragment in Android.
3. What is a ViewGroup in Android?
4. What is a RecyclerView in Android?
5. What is Content Resolver?
6. What is SQLite?

Section-B

II. Answer any Four question. Each question carries Five marks **(4 x 5 = 20)**

7. Explain Key Mobile Application Services.
8. How to Use ProgressBar in Android?
9. Explain TimePicker along with its attributes and features. Give an example
10. How can a custom theme be created and applied to an activity in Android?
11. Explain the basic steps to send an SMS using `SmsManager` in an Android application.
12. Outline the basic steps involved in consuming web services using HTTP in an Android application.

Section-C

III. Answer any Four questions. Each question carries Eight marks **(4 X 8 = 32)**

13. Explain the Steps in Publishing Android Application.
14. a) Explain the Features and Functionalities of an Activity
    b) How to Create an Intent? Explain with example.
15. a) Explain RecyclerView along with its attributes and features. Give an example
    b) Explain DialogFragment along with its attributes and features. Give an example.
16. a) Explain Resizing and Repositioning Views to Adapt Display Orientation
    b) Explain the process of creating a view programmatically in Android.
17. Describe the step-by-step implementation of data binding in an Android application
18. Explain the Step-by-Step Implementation of Consuming JSON Web Services

◎◎◎◎

Model Question Paper - 3

Time : 2 ½ Hours Max. Marks : 60

Instructions : Answer All Sections

Section-A

I. Answer any Four questions. Each question carries Two marks (4 X 2 = 8)

1. What is an APK?
2. What is an Activity in Android?
3. What is a 'TextView' in Android?
4. What is a ListView in Android?
5. What is Data Binding in Android?
6. What do you mean by XML Web Services.

Section-B

II. Answer any Four question. Each question carries Five marks (4 x 5 = 20)

7. Explain Mobile Operating Systems.
8. Write the Steps to Debug an Android Application in Android Studio.
9. Explain NumberPicker along with its attributes and features. Give an example
10. Outline the steps to add an options menu in the Sudoku app, including changing the theme and defining the menu in XML
11. Explain the process of monitoring a location using the `LocationManager` class in Android
12. Explain how to retrieve data from a SQLite database using the query method

Section-C

III. Answer any Four questions. Each question carries Eight marks (4 X 8 = 32)

13. Explain Different Mobile Technologies.
14. a) Explain the types of Intents with example code.
    b) What is Dialog in Android? Explain the Types of Dialogs.
15. Discuss how to create a preference management system using Shared Preferences. Provide a detailed code example.
16. a) Explain Creating Separate Layouts for Different Orientations. How it works?
    b) Explain Constraint Layout along with its features, attributes, advantages and disadvantage
17. Explain the steps in using Custom Content Provider.
18. Provide a detailed explanation of how to integrate Google Maps into an Android application

◎◎◎◎

Model Question Paper - 4

Time : 2 ½ Hours Max. Marks : 60

Instructions : Answer All Sections

Section-A

I. Answer any Four questions. Each question carries Two marks (4 X 2 = 8)

1. What is Android Virtual Device (AVD)?
2. What is an Intent in Android ?
3. What is an Action Bar in Android?
4. What is an AutoCompleteTextView in Android?
5. What is Content Provider URI ? Give an example.
6. Why is the `HttpURLConnection` class used in Android?

Section-B

II. Answer any Four question. Each question carries Five marks (4 x 5 = 20)

7. What is Android? Explain the Key Features of Android OS.
8. Explain how the intent object is used to invoke built-in applications in Android.
9. Explain WebView along with its attributes and features. Give an example
10. Provide a detailed explanation on the implementation of an About Box in the Sudoku app.
11. Explain how to query database and fetch the records using a Cursor.
12. Explain the process of creating a Custom Content Provider with example code

Section-C

III. Answer any Four questions. Each question carries Eight marks (4 X 8 = 32)

13. Explain the Android Application Components
14. Describe the complete lifecycle of an Android activity and the purpose of each lifecycle method.
15. Discuss how to implement file reading and writing operations in both internal and external storage. Provide a detailed code example
16. a) Explain Linear Layout along with its features, attributes, advantages and disadvantages
    b) Discuss in detail the different types of UI notifications that can be listened to at the activity level in Android.
17. Provide a detailed example of implementing live data binding in an Android application.
18. Describe the process of creating and managing a proximity alert in an Android application.

◎◎◎◎