Chapter-1
Introduction to Android

Android is an open-source operating system developed by Google, primarily designed for touchscreen mobile devices such as smartphones and tablets. It is based on the Linux kernel and provides a rich set of features for mobile application development.

Key Features of Android

1. **Open-Source**: Android's open-source nature allows developers to modify and distribute it, enabling a vast developer community to contribute to its ecosystem.
2. **Customizable User Interface**: Android supports a variety of customizable UIs, making it versatile for device manufacturers and developers.
3. **Multitasking**: Android supports running multiple applications simultaneously, enhancing user productivity.
4. **Connectivity**: Android supports a wide range of connectivity options, such as Wi-Fi, Bluetooth, NFC, and USB.
5. **Rich Application Framework**: Developers can build innovative apps using tools and APIs provided by Android.
6. **Support for Multimedia**: Android supports a broad range of audio, video, and image formats, making it ideal for multimedia-rich applications.
7. **Regular Updates**: Google regularly updates Android, introducing new features and security enhancements.

Components of Android Architecture

1. **Applications**: These are the user-facing apps installed on the device, like messaging, email, and games.
2. **Application Framework**: Provides APIs for building robust and interactive applications.
3. **Libraries**: Android includes a set of native libraries, such as WebKit for web browsing and OpenGL for graphics rendering.
4. **Android Runtime (ART)**: Manages application execution and optimizes app performance through Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilation.
5. **Linux Kernel**: Acts as the core, handling hardware abstraction, memory management, and security.

Development in Android

- **Programming Languages**: Android applications are primarily developed using Java, Kotlin, or C++.
- **Integrated Development Environment (IDE)**: Android Studio, provided by Google, is the official IDE for Android development.
- **Development Tools**: Includes tools like Gradle for build automation, Android Emulator for testing, and Logcat for debugging.

Advantages of Android

1. **Wide Market Reach**: With Android being the dominant mobile OS globally, applications reach a large audience.
2. **Diverse Device Support**: Android powers devices across a range of price points, making it accessible to various users.
3. **Robust Community Support**: A large developer community ensures constant innovation and support.
4. **Monetization Opportunities**: Developers can monetize apps via Google Play Store, ads, or in-app purchases.

Applications of Android

1. **Mobile Applications**: Games, social networking apps, productivity tools, etc.

2. **IoT and Wearables**: Android is widely used in IoT devices and wearables like smartwatches.
3. **Automotive Systems**: Android Auto offers integration with car infotainment systems.
4. **Smart TVs**: Android TV powers many smart televisions.
5. **Healthcare and Fitness Apps**: Enables health tracking and fitness monitoring applications.

History of Android

The history of Android is an interesting journey of evolution from a small startup idea to the dominant mobile operating system in the world. Below is a detailed timeline of its key milestones:

1. Early Beginnings (2003-2005)

- **2003**: Android Inc. is founded by **Andy Rubin**, **Rich Miner**, **Nick Sears**, and **Chris White**. The original goal of the company was to develop an advanced operating system for digital cameras, but the founders soon realized the broader potential of creating a mobile operating system.
- **2005**: Google acquires Android Inc. for an undisclosed amount. Google saw the potential for Android to become a mobile operating system for smartphones. Rubin, Miner, and the rest of the team join Google, and Android starts becoming more focused on mobile devices.

2. The Development Phase (2005-2007)

- **2007**: The **Open Handset Alliance (OHA)** is formed, a consortium of technology and mobile companies aimed at promoting open standards for mobile devices. Google announces Android as an open-source platform for mobile phones.
  - **Key members of the OHA**: Google, HTC, Intel, Qualcomm, Motorola, and Samsung.
  - Google also announced the launch of the **Android Software Development Kit (SDK)** for developers.
- **2008**: Google announces the first **Android-powered device**: the **HTC Dream (also known as the T-Mobile G1)**. It was the first smartphone running Android OS, featuring a physical keyboard and a touchscreen.

3. Android's Rise (2008-2010)

- **2008-2009**: Android starts gaining traction as more devices adopt it. It is recognized for being open-source, which allows manufacturers to modify it as needed.
- **2009**: Android Market (now **Google Play Store**) is launched, allowing developers to distribute their apps.
- **2010**: Google launches **Android 2.2 (FroYo)**, which brings significant improvements in performance, features, and speed.
- **2010**: Android's market share begins to grow rapidly, and it is recognized as a strong competitor to Apple's iOS, which was dominating the smartphone market.

4. Establishing Dominance (2011-2014)

- **2011**: Android is now a widely adopted platform, with major companies like Samsung, Motorola, and LG releasing Android-powered devices.
  - Google's acquisition of **Motorola Mobility** in 2012 boosts Android's presence in hardware, leading to further growth in the Android ecosystem.
- **2012**: Android surpasses **iOS** in terms of the number of active devices globally. **Android 4.0 (Ice Cream Sandwich)** is released, introducing a major redesign of the interface and unifying the phone and tablet UI.
- **2013**: Android continues to dominate the smartphone market, with versions like **Jelly Bean** and **KitKat** improving performance, user interface, and overall user experience.

- **2014**: **Android 5.0 (Lollipop)** is launched, with major changes to the design language (Material Design), which gave a more polished and modern look to the OS.

5. Android Expands Beyond Smartphones (2014-2020)

- **2014**: Google announces **Android Wear**, a version of Android optimized for smartwatches and wearables.
- **2015**: **Android 6.0 (Marshmallow)** focuses on app permissions, improved battery life, and fingerprint support.
- **2017**: **Android 8.0 (Oreo)** is launched, with features like picture-in-picture mode and adaptive icons. Google also introduces **Android Go**, a lightweight version of Android designed for low-end smartphones.
- **2018**: **Android 9.0 (Pie)** arrives, with focus on improving AI-driven features such as battery optimization, adaptive brightness, and gesture navigation.
- **2020**: **Android 10** introduces a system-wide dark mode and several privacy-focused features. It marks the shift in naming conventions, moving from dessert-based names (e.g., Oreo, Pie) to simple numerical names.

6. Recent Developments (2020-Present)

- **2021**: **Android 12** is released, with
- a major design overhaul called **Material You**, providing users with more customization options for the UI, including dynamic theming based on wallpaper colors.
- **2022**: **Android 13** brings enhanced privacy and security features, more customization, and better support for foldable devices.
- **2023**: **Android 14** was introduced with improvements for larger screen devices, new customization features, and additional privacy and security enhancements.

Impact of Android

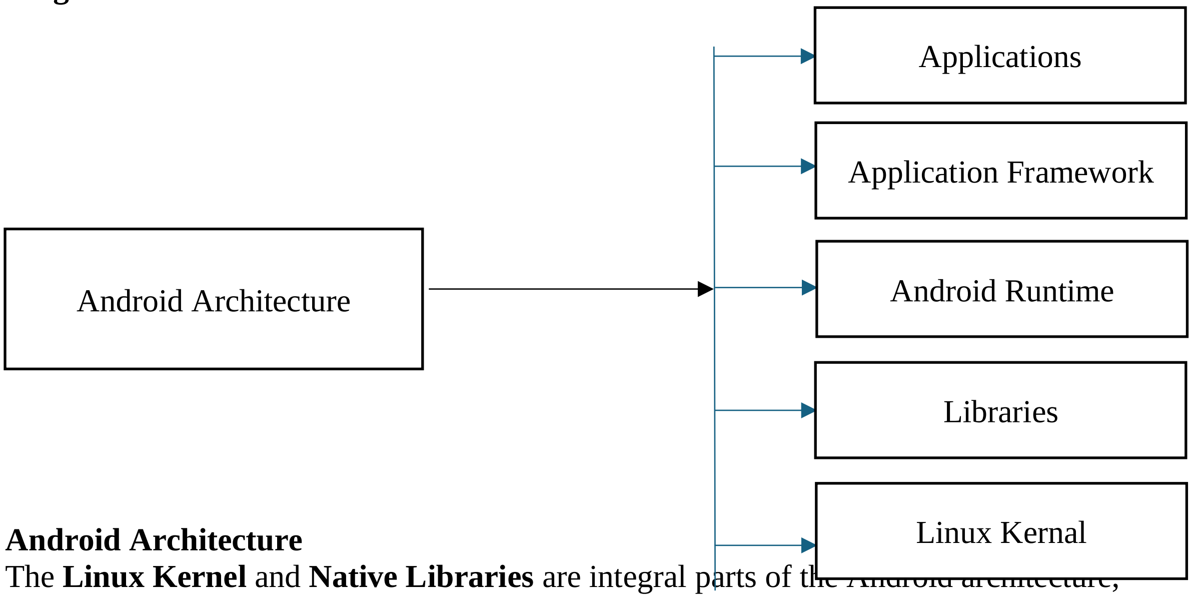
1. **Mobile Device Dominance**: Android has become the most popular mobile operating system worldwide, with over 70% of the global market share, dominating the smartphone and tablet industries.
2. **App Ecosystem**: The Android ecosystem has fostered an immense app marketplace, offering millions of apps through the Google Play Store.
3. **Diversity in Devices**: Android powers a wide range of devices, from budget to flagship phones, tablets, smartwatches, and even automotive systems (Android Auto).
4. **Openness and Customization**: The open-source nature of Android has allowed manufacturers to customize and differentiate their devices, contributing to a rich diversity of Android-powered gadgets.
5. **Monetization for Developers**: Android has provided a platform for developers to create and distribute apps globally, allowing monetization through in-app purchases, ads, and paid apps.

Summary

Year	Event
2003	Android Inc. founded by **Andy Rubin**, **Rich Miner**, **Nick Sears**, and **Chris White**. Initially developed for digital cameras.
2005	Google acquires Android Inc. and its team, shifting focus to mobile OS development.
2007	**Open Handset Alliance (OHA)** is formed. Google announces Android as an open-source mobile OS.
2008	First Android device, **HTC Dream (T-Mobile G1)**, is launched. **Android SDK** is introduced.

Year	Event
2009	**Android Market (now Google Play Store)** is launched for app distribution.
2010	**Android 2.2 (FroYo)** released with performance improvements. Android starts gaining momentum.
2011	Android adoption grows with major manufacturers like Samsung, LG, and Motorola. Google acquires **Motorola Mobility**.
2012	Android surpasses iOS in global active device numbers. **Android 4.0 (Ice Cream Sandwich)** is released, unifying phone and tablet interfaces.
2013	**Android 4.3 (Jelly Bean)** and **Android 4.4 (KitKat)** improve system performance and introduce new features.
2014	**Android 5.0 (Lollipop)** released with **Material Design** overhaul. Android begins expanding into wearables with **Android Wear**.
2015	**Android 6.0 (Marshmallow)** introduces app permissions, improved battery life, and fingerprint support.
2017	**Android 8.0 (Oreo)** released, with features like picture-in-picture mode and adaptive icons. **Android Go** is launched for low-end smartphones.
2018	**Android 9.0 (Pie)** adds AI features, improved battery optimization, and gesture navigation.
2020	**Android 10** drops the dessert naming convention and introduces system-wide **dark mode** and enhanced privacy features.
2021	**Android 12** introduces **Material You**, allowing more UI customization and a more personalized experience.
2022	**Android 13** enhances privacy, security, and adds support for foldable devices and tablets.
2023	**Android 14** introduces features for larger screen devices and further improves privacy and security.

Diagram of Android Architecture



Android Architecture

The **Linux Kernel** and **Native Libraries** are integral parts of the Android architecture, forming the core that supports the higher layers like the application framework and user applications. Android Runtime, Application Framework, and Applications in Android Architecture, these layers build upon the foundation provided by the **Linux Kernel** and

Native Libraries, enabling app developers to create and deliver seamless user experiences. Below is a detailed explanation of each:

1. Linux Kernel

The **Linux Kernel** is the foundation of Android's architecture. It acts as an abstraction layer between the hardware and software, handling essential low-level operations.

Key Features of the Linux Kernel:

- **Hardware Abstraction**: Provides an interface for hardware components like memory, processor, and peripherals (e.g., camera, audio).
- **Device Drivers**: Includes drivers for devices like display, Wi-Fi, Bluetooth, and battery management.
- **Security**: Implements security features like user authentication, process isolation, and SELinux (Security-Enhanced Linux) for enforcing security policies.
- **Power Management**: Efficiently manages battery consumption by controlling CPU and device activity.
- **Memory Management**: Allocates and manages memory for applications and processes.

Examples of Kernel Responsibilities:

- Managing communication between apps and the physical hardware.
- Handling file systems, process scheduling, and system resources.

2. Native Libraries

Native libraries in Android are written in C/C++ and provide essential functionalities for various system components and application-level features. These libraries are used by both the Android Runtime and the Application Framework.

Key Native Libraries in Android:

Library	Purpose
WebKit	A web browser engine for rendering web pages.
OpenGL ES	For 2D and 3D graphics rendering, often used in games and multimedia applications.
SQLite	A lightweight database engine for data storage in apps.
SSL/crypto	Provides secure communication through encryption.
Media Framework	Supports audio, video playback, and recording (e.g., MP3, AAC, MPEG-4).
FreeType	Handles font rendering.
Surface Manager	Manages screen composition and handles display layers.
LibC	Provides standard C library functions, such as memory allocation and I/O operations.

Role of Native Libraries:

- They provide the underlying support for multimedia processing, database management, and user interface rendering.
- Native libraries are accessed via the **Application Framework** or directly through the **JNI (Java Native Interface)** for performance-critical tasks.

3. Android Runtime (ART)

The **Android Runtime (ART)** is responsible for executing applications and managing memory, resources, and process lifecycles.

Key Features:

- **Ahead-of-Time (AOT) Compilation**: Converts app bytecode into native machine code during installation, improving app performance.
- **Just-in-Time (JIT) Compilation**: Optimizes code execution at runtime for better resource utilization.
- **Garbage Collection**: Automatically reclaims memory from objects no longer in use, ensuring efficient memory management.
- **Core Libraries**: Provide essential APIs for data structures, file I/O, networking, threading, and other basic functionalities needed by apps.

Role in Android Architecture:

- Executes applications efficiently and ensures a smooth user experience.
- Works closely with the **Linux Kernel** and **Native Libraries** for resource management and hardware interactions.

4. Application Framework

The **Application Framework** provides developers with a set of high-level APIs to build apps without dealing with lower-level system complexities. It acts as the middle layer between the runtime and user applications.

Key Components:

Component	Description
Activity Manager	Manages the lifecycle of apps and activities.
Window Manager	Manages windows and user interface components.
Content Providers	Facilitates sharing of data between applications.
Resource Manager	Handles resources like images, strings, and layouts.
Notification Manager	Manages system and app notifications.
View System	Provides UI elements such as buttons, text fields, and other widgets.
Telephony Manager	Handles phone-related functionalities like calls and SMS.
Location Manager	Provides access to location-based services (GPS, network-based location).
Package Manager	Manages app installation, updates, and permissions.

Role in Android Architecture:

- Offers services and frameworks to simplify app development.
- Enables reusability of components, as apps can share data and functionality through the framework.

5. Applications

The **Applications** layer represents the topmost level in the Android architecture. These are the apps that users interact with daily.

Types of Applications:

Type	Examples
System Apps	Pre-installed apps like Phone, Contacts, Messaging, Camera, and Settings.
User Apps	Third-party apps installed via Google Play Store or other sources (e.g., WhatsApp, Instagram).

Features:

- Built using the **Android SDK** (Java/Kotlin).
- Leverage the **Application Framework** for features like UI design, data storage, and notifications.

- Run within their own **sandboxed environment** to ensure security and isolation.

Role in Android Architecture:

- Serve as the primary interface for user interaction.
- Utilize the underlying layers (Application Framework, Runtime, Native Libraries, and Kernel) for functionality.

Relationship Between the Five Layers of Android Architecture

Layer	Relies On	Provides To	Key Role
Linux Kernel	Hardware	Native Libraries, Android Runtime	Manages hardware resources, device drivers, security, and power management.
Native Libraries	Linux Kernel	Android Runtime, Application Framework	Provides essential services like graphics rendering, database access, and media playback.
Android Runtime (ART)	Linux Kernel, Native Libraries	Application Framework	Executes app code, manages memory, and optimizes app performance through AOT/JIT compilation.
Application Framework	Android Runtime, Native Libraries	Applications	Provides APIs and high-level services for building apps (e.g., UI, notifications, telephony).
Applications	Application Framework	End Users	User-facing software for communication, multimedia, and other functions.

Detailed Flow of Interaction

Interaction Type	From	To	Description
Hardware Communication	Linux Kernel	Hardware	Handles hardware communication through device drivers.
Low-Level Services	Native Libraries	Linux Kernel	Libraries like OpenGL, SQLite, and Media Framework access kernel services for functionality.
Runtime Execution	Android Runtime	Native Libraries, Linux Kernel	ART uses libraries for execution tasks and kernel for hardware abstraction.
High-Level API Access	Application Framework	Android Runtime, Native Libraries	Framework provides APIs for developers to use runtime and libraries for app features.
User Interaction	Applications	Application Framework	Apps interact with the framework to display UI, access services, and perform tasks like database queries.

Exercise

Very Short Questions (1 Mark each)

Multiple-Choice Questions (MCQs)

1. Which layer of Android architecture directly communicates with the hardware?

A. Application Framework
B. Native Libraries
C. Linux Kernel
D. Android Runtime

Answer: C

2. What is the primary purpose of the Android Runtime (ART)?
   A. Rendering graphics
   B. Managing memory and executing applications
   C. Accessing hardware components
   D. Providing APIs to developers

   Answer: B
3. Which of the following is not a part of the Android Native Libraries?
   A. OpenGL
   B. SQLite
   C. Activity Manager
   D. WebKit

   Answer: C
4. What does the Application Framework layer primarily provide?
   A. Low-level hardware interaction
   B. APIs and services for app development
   C. User interface rendering
   D. Device drivers

   Answer: B
5. Which component in the Application Framework manages notifications?
   A. Window Manager
   B. Notification Manager
   C. Activity Manager
   D. Package Manager

   Answer: B

True/False Questions

1. The Linux Kernel is responsible for executing app code. (**False**)
2. The Android Runtime supports both Ahead-of-Time and Just-in-Time compilation. (**True**)
3. The Notification Manager is part of the Android Runtime. (**False**)
4. SQLite is a database engine provided in the Native Libraries layer. (**True**)
5. Applications rely directly on the Linux Kernel for hardware access. (**False**)

Short Questions (2 Marks each)

1. Define the role of the Linux Kernel in Android architecture.
2. What are native libraries in Android, and why are they important?
3. Explain the difference between system apps and user apps.
4. What is the role of the Activity Manager in the Application Framework?
5. Mention two components of the Android Runtime.

Long Questions (5 Marks each)

1. Describe the architecture of Android with a focus on its layered structure.
2. Discuss the functionalities provided by the Native Libraries in Android architecture.
3. Compare the roles of Android Runtime and Application Framework in app development.

4. Explain how the Linux Kernel and Native Libraries interact within the Android architecture.
5. Highlight the key components of the Application Framework and their roles.

Very Long Questions (10 Marks each)

1. Describe in detail the relationship between the five layers of Android architecture.
2. Explain the role of each layer in Android architecture with examples.
3. Illustrate with a diagram the architecture of Android and explain the interaction between layers.

Chapter-2

Android Application Components: Detailed Overview

Android application components are the building blocks of an Android app. Each component plays a unique role and interacts with others to create a functional app. These components are activated using **intents**, which are messages that allow communication between components.

1. Activities

*Definition: An **activity** is a component that provides the user interface (UI) of an application. Each screen in an Android app is typically represented by an activity.*

Key Features:

- Represents a single screen with a UI.
- Manages user interactions and navigation between screens.
- Multiple activities can be present in an app, and they interact through intents.

Lifecycle:

1. **onCreate()**: Called when the activity is created. Used for initialization (e.g., setting the layout, initializing variables).
2. **onStart()**: Invoked when the activity becomes visible.
3. **onResume()**: Called when the activity is in the foreground and ready for user interaction.
4. **onPause()**: Triggered when the activity is partially obscured (e.g., another activity appears).
5. **onStop()**: Invoked when the activity is no longer visible.
6. **onDestroy()**: Called before the activity is destroyed.
7. **onRestart()**: Called when the activity is restarted after being stopped.

Example: A login screen is an activity where users enter their credentials. On successful login, the app navigates to another activity, such as a dashboard.

2. Services

*Definition: A **service** is a component that performs long-running operations in the background without a user interface.*

Key Features:

- Ideal for background tasks like downloading files, playing music, or syncing data.
- Operates independently of the user interface.

Types:

1. **Foreground Service**: Displays a persistent notification (e.g., music player).
2. **Background Service**: Runs without user interaction but may be limited by system restrictions (e.g., periodic data sync).
3. **Bound Service**: Allows other components to bind to it and interact (e.g., retrieving data from a remote server).

Lifecycle:

1. **onCreate()**: Called when the service is created.
2. **onStartCommand()**: Executes each time a service is started.
3. **onBind()**: Used for bound services; returns an interface for interaction.
4. **onDestroy()**: Cleans up resources when the service is stopped.

Example: A music player app uses a service to play music in the background while the user navigates through other apps.

3. Broadcast Receivers

*Definition: Broadcast receivers listen for and respond to **system-wide broadcasts** or **app-specific broadcasts**.*

Key Features:

- Passive component that waits for events.
- Useful for handling asynchronous tasks triggered by system events.

Common Broadcasts:

- System-generated: Battery low, network change, device boot.
- App-generated: Custom events broadcasted by apps.

Lifecycle:

- **onReceive()**: The only method invoked when the broadcast is received. Must complete quickly.

Example: A broadcast receiver listens for a "low battery" broadcast and alerts the user to save their work.

4. Content Providers

*Definition: A **content provider** enables apps to share structured data securely.*

Key Features:

- Used for data sharing between applications.
- Implements a standard interface for querying, inserting, updating, and deleting data.

Use Cases:

- Sharing contacts, media files, or app-specific data.
- Providing data to widgets or external apps.

Core Methods:

1. query(): Retrieves data.
2. insert(): Adds new data.
3. update(): Modifies existing data.
4. delete(): Removes data.

Example: The Contacts app provides access to user contacts via a content provider. Other apps, like messaging apps, can query this data.

5. Fragments

*Definition: A **fragment** is a reusable portion of the UI and logic that can be embedded in activities.*

Key Features:

- Modularizes UI for better scalability and adaptability.
- Can be dynamically added or removed during runtime.
- Helps in managing multiple layouts for different screen sizes.

Lifecycle:

1. **onAttach()**: Called when the fragment is associated with an activity.
2. **onCreateView()**: Inflates the fragment's UI.
3. **onResume()**: Fragment becomes visible.
4. **onPause()**: Fragment is partially obscured.
5. **onDestroyView()**: Cleans up resources related to the UI.

Example: A news app uses fragments to display a list of articles on one side and the details of a selected article on the other (for tablets).

Relationship Between Components

- **Activities** are the main entry points for user interaction.
- **Services** handle background tasks initiated by activities or other components.
- **Broadcast Receivers** listen for events that might trigger an activity or service.
- **Content Providers** enable sharing of data between activities or even different apps.
- **Fragments** modularize activities for better adaptability and UI management.

Application Component	Description	Relationship with Other Components
Activity	Represents a single screen with a user interface.	Can start other activities (via Intent). Can interact with services or broadcast receivers.
Service	Runs in the background to perform long-running tasks.	Can interact with activities, content providers, and broadcast receivers for data exchange.
Broadcast Receiver	Listens for system-wide broadcast announcements.	Can start activities or services based on broadcast intents. Can also communicate with content providers.
Content Provider	Manages shared data and provides it to other applications.	Can interact with activities, services, and broadcast receivers to share data across apps.

Application Component	Description	Relationship with Other Components
Intent	An object that represents an application's intention to perform an action.	Used to start activities, services, and send broadcasts. Serves as a means of communication between components.
Fragment	A modular section of an activity's UI.	Tied to an activity. Can interact with the activity and other fragments within the same activity.

Additional Components:

1. **Widgets:**

Widgets in Android are UI components that provide users with a way to interact with the application. They can display information, allow input, and trigger actions.

1. TextView

- **Description**: Displays text to the user.
- **Properties**:
  - Can display static or dynamic text.
  - Supports various formatting options, such as text size, style, color, and alignment.
  - Supports HTML formatting with the use of Html.fromHtml() method.
  - Text can be dynamically set using setText() method.
- **Common Usage**: Display labels, messages, instructions, or results.

Example: TextView textView = findViewById(R.id.textView); textView.setText("Hello, World!");

2. Button

- **Description**: A clickable element that triggers actions.
- **Properties**:
  - Can be styled to look like a physical button.
  - Can trigger actions via an onClickListener.
  - Can display text, images, or both.
- **Common Usage**: Use for actions such as submitting forms, opening new screens, or executing commands.

  Example:

```
Button button = findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
      // Action on button click
```

```
    }
});
```

3. EditText

- **Description**: A text input field.
- **Properties**:
  - Can accept single-line or multi-line input.
  - Supports various input types like text, password, email, number, etc.
  - Can display hints, request focus, and handle validation.
- **Common Usage**: Used for capturing user input, such as usernames, passwords, and messages.

  Example: EditText editText = findViewById(R.id.editText); String input = editText.getText().toString();

4. ImageView

- **Description**: Displays an image.
- **Properties**:
  - Supports images from resources, URLs, or files.
  - Can scale images (e.g., centerCrop, fitCenter).
  - Can apply various transformations like rotations or borders.
- **Common Usage**: Display images such as icons, logos, and photos.

  Example: ImageView imageView = findViewById(R.id.imageView); imageView.setImageResource(R.drawable.icon);

5. CheckBox

- **Description**: A toggle button that can be checked or unchecked.
- **Properties**:
  - Supports single or multiple selections.
  - Has isChecked() and setChecked() methods to get/set its state.
  - Supports text and can be styled with custom colors or drawable resources.
- **Common Usage**: Used for forms or settings where users need to select one or more options.

  Example:

  ```
  CheckBox checkBox = findViewById(R.id.checkBox);
  checkBox.setChecked(true);
  ```

6. RadioButton

- **Description**: A button that behaves like a checkbox, but only one can be selected from a group.
- **Properties**:
  - Can only be part of a RadioGroup to allow exclusive selection.
  - isChecked() and setChecked() methods allow getting and setting its state.

- **Common Usage**: Used in forms where users need to select a single option from a group.

  Example:

  ```
  RadioButton radioButton = findViewById(R.id.radioButton);
  radioButton.setChecked(true);
  ```

7. Switch

- **Description**: A Material Design toggle switch.
- **Properties**:
  - Allows binary choices (on/off) with visual feedback.
  - Can be customized with custom thumb and track drawable resources.
  - Listens for state change using setOnCheckedChangeListener.
- **Common Usage**: Used in settings screens to toggle between two options, such as turning a feature on or off.

  Example:

  ```
  Switch switch = findViewById(R.id.switch);
  switch.setChecked(true);
  switch.setOnCheckedChangeListener(new
  CompoundButton.OnCheckedChangeListener() {
     @Override
     public void onCheckedChanged(CompoundButton buttonView, boolean
  isChecked) {
         // Handle state change
     }
  });
  ```

8. SeekBar

- **Description**: A slider widget to select a value from a range.
- **Properties**:
  - Can be horizontal or vertical.
  - Displays a progress level and allows users to select a value within a specified range.
  - Can have tick marks, progress, and a thumb (handle).
  - Supports listeners for onProgressChanged.
- **Common Usage**: Commonly used for volume controls, brightness adjustment, or any continuous value selection.

  Example:

  ```
  SeekBar seekBar = findViewById(R.id.seekBar);
  seekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener()
  {
     @Override
  ```

```
    public void onProgressChanged(SeekBar seekBar, int progress, boolean
fromUser) {
        // Handle progress change
    }
    // Other methods omitted for brevity
});
```

9. ProgressBar

- **Description**: Displays progress of a task.
- **Properties**:
  - Can be determinate or indeterminate.
  - Used to show how much of a task has been completed.
  - Supports custom styles and progress levels.
- **Common Usage**: Used when performing long-running tasks like downloading files or loading content.

  Example:

```
ProgressBar progressBar = findViewById(R.id.progressBar);
progressBar.setVisibility(View.VISIBLE); // Show progress
progressBar.setProgress(50); // Update progress
```

10. Spinner

- **Description**: A dropdown menu allowing users to select one item from a list.
- **Properties**:
  - Displays a list of items in a drop-down.
  - Can be customized with adapters to provide dynamic data.
- **Common Usage**: Used when you want users to choose from a predefined set of options (e.g., country selection, categories).

  Example:

```
Spinner spinner = findViewById(R.id.spinner);
ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
android.R.layout.simple_spinner_dropdown_item, items);
spinner.setAdapter(adapter);
```

11. AutoCompleteTextView

- **Description**: A text input widget with suggestions.
- **Properties**:
  - Displays suggestions based on the user's input.
  - Useful for search bars or input fields where predefined suggestions are necessary.
- **Common Usage**: Auto-suggest feature in search, tag input, or location entry.

  Example:

```
AutoCompleteTextView autoCompleteTextView =
findViewById(R.id.autoCompleteTextView);
ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
android.R.layout.simple_dropdown_item_1line, suggestions);
autoCompleteTextView.setAdapter(adapter);
```

12. RatingBar

- **Description**: Allows users to rate something, typically using stars.
- **Properties**:
    - Provides a visual representation of ratings (usually stars or other symbols).
    - Can be customized to show different symbols and ratings.
- **Common Usage**: Used in review or feedback systems to rate products, services, or media.

    Example:

```
RatingBar ratingBar = findViewById(R.id.ratingBar);
ratingBar.setRating(3); // Set a 3-star rating
```

13. CalendarView

- **Description**: A widget to display a calendar and select a date.
- **Properties**:
    - Allows users to navigate through months and select specific dates.
    - Can be customized to highlight certain dates.
- **Common Usage**: Used for scheduling, date pickers, and event management.

    Example:

```
CalendarView calendarView = findViewById(R.id.calendarView);
calendarView.setOnDateChangeListener(new
CalendarView.OnDateChangeListener() {
   @Override
   public void onSelectedDayChange(CalendarView view, int year, int month, int
dayOfMonth) {
       // Handle date selection
   }
});
```

14. DatePicker

- **Description**: Allows users to select a date.
- **Properties**:
    - Shows a date picker dialog where users can select the day, month, and year.
- **Common Usage**: Used for forms where users need to select a specific date.

    Example:

```
DatePicker datePicker = findViewById(R.id.datePicker);
int day = datePicker.getDayOfMonth();
int month = datePicker.getMonth();
int year = datePicker.getYear();
```

15. TimePicker

- **Description**: Allows users to select a time.
- **Properties**:
  - Displays a time picker dialog for users to choose the hour and minute.
- **Common Usage**: Used for setting event times, alarms, or reminders.

  Example:

```
TimePicker timePicker = findViewById(R.id.timePicker);
int hour = timePicker.getCurrentHour();
int minute = timePicker.getCurrentMinute();
```

These widgets form the backbone of user interface design in Android applications, helping to build interactive, dynamic, and responsive UIs.

2. Layouts

In Android, **Layouts** are used to define the structure and arrangement of user interface elements (UI components) on the screen. They serve as containers for UI elements such as buttons, text fields, images, etc. Layouts help organize the visual structure and ensure a responsive, user-friendly interface. There are various types of layouts, each with unique properties and behavior.

1. LinearLayout

- **Description**: A layout that arranges its children in a single direction, either vertically or horizontally.
- **Properties**:
  - orientation: Defines whether the children are laid out horizontally or vertically.
  - weight: Allows distributing space proportionally among child views. A view with weight will take up more space relative to others.
  - gravity: Aligns the children inside the layout (e.g., center, left, right).
  - layout_width and layout_height: Can be defined as match_parent, wrap_content, or a specific value.
- **Common Usage**:When you need to align elements sequentially (either vertically or horizontally).

  Example:

```
<LinearLayout
   android:orientation="vertical"
```

```
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <Button android:text="Button 1" />
    <Button android:text="Button 2" />
</LinearLayout>
```

2. RelativeLayout

- **Description**: A layout where child elements are positioned relative to each other or the parent container.
- **Properties**:
  - Allows positioning views relative to others (e.g., below, alignParentRight, centerInParent).
  - Can be used for complex layouts that require precise control over view positioning.
- **Common Usage**: When you want to align views relative to other views or to the parent layout.

  Example:

```
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/button1"
        android:text="Button 1"
        android:layout_alignParentTop="true"/>
    <Button
        android:text="Button 2"
        android:layout_below="@id/button1"/>
</RelativeLayout>
```

3. ConstraintLayout

- **Description**: A more flexible and efficient layout that allows you to create complex layouts with flat view hierarchies using constraints.
- **Properties**:
  - Views are positioned relative to each other using constraints like startToStartOf, endToEndOf, topToTopOf, etc.
  - It has better performance than RelativeLayout because it avoids deep nested views.
- **Common Usage**: For designing complex UI where views need to be aligned in various ways (e.g., constraints on all four sides).

  Example:

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/button1"
        android:text="Button 1"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"/>
    <Button
        android:id="@+id/button2"
        android:text="Button 2"
        app:layout_constraintTop_toBottomOf="@id/button1"
        app:layout_constraintStart_toStartOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

4. FrameLayout

- **Description**: A layout designed to block out an area on the screen to display a single item.
- **Properties**:
  - Views are stacked on top of each other; the last added view is displayed on top.
  - Useful for showing a single view or fragment in a particular area of the screen.
- **Common Usage**: When you want to show one view at a time or a single item like an image or a fragment.

  Example:

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ImageView android:src="@drawable/image" />
</FrameLayout>
```

5. TableLayout

- **Description**: A layout that arranges its children into rows and columns, like an HTML table.
- **Properties**:
  - The layout contains rows (TableRow) and each row contains individual views.
  - It automatically arranges the children into columns based on the TableRow arrangement.
- **Common Usage**: Ideal for displaying data in a tabular format.

  Example:

```
<TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <TableRow>
        <TextView android:text="Row 1, Column 1" />
        <TextView android:text="Row 1, Column 2" />
    </TableRow>
    <TableRow>
        <TextView android:text="Row 2, Column 1" />
        <TextView android:text="Row 2, Column 2" />
    </TableRow>
</TableLayout>
```

6. GridLayout

- **Description**: A layout that arranges its children into a grid of rows and columns, similar to TableLayout, but more flexible.
- **Properties**:
  - Allows more control over positioning with specific row and column indices.
  - Can define both row and column span for a child view.
- **Common Usage**: Ideal for situations where you need to arrange items in a grid format (e.g., calculator layout, dashboard).

  Example:

```
<GridLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:columnCount="2">
    <Button android:text="Button 1" />
    <Button android:text="Button 2" />
</GridLayout>
```

7. ScrollView

- **Description**: A layout that allows scrolling when the content exceeds the screen size.
- **Properties**:
  - Can contain only one direct child, but that child can contain other views.
  - Used when content might overflow vertically.
- **Common Usage**: For forms or any UI where the content is dynamic or might exceed the screen size.

  Example:

```
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <Button android:text="Button 1" />
        <Button android:text="Button 2" />
    </LinearLayout>
```

```
</ScrollView>
```

8. HorizontalScrollView

- **Description**: A layout that allows horizontal scrolling of its content when the content exceeds the screen width.
- **Properties**:
    - Similar to ScrollView, but allows horizontal scrolling.
    - Can contain only one direct child, which is typically a horizontally arranged layout (e.g., LinearLayout).
- **Common Usage**: For horizontal lists, image galleries, or other elements that may exceed the screen width.

  Example:

```
<HorizontalScrollView
   android:layout_width="match_parent"
   android:layout_height="wrap_content">
   <LinearLayout
      android:orientation="horizontal"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content">
      <Button android:text="Button 1" />
      <Button android:text="Button 2" />
   </LinearLayout>
</HorizontalScrollView>
```

9. CoordinatorLayout

- **Description**: A super-powered FrameLayout designed to coordinate interactions between its child views.
- **Properties**:
    - It provides advanced features like app bars, floating action buttons, and more.
    - It works with other Material Design components such as AppBarLayout and FloatingActionButton to achieve complex behaviors like scrolling or collapsing toolbars.
- **Common Usage**: For building dynamic, interactive layouts that require scrolling, complex behaviors, and transitions.

  Example:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <AppBarLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content">
      <Toolbar android:title="Coordinator Layout" />
```

```
    </AppBarLayout>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me" />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

10. ViewGroup

- **Description**: The base class for all layout containers that manage and position child views.
- **Properties**:
  - A ViewGroup is a container that holds other views (or view groups) and can arrange them in any form.
  - Examples of ViewGroup include LinearLayout, RelativeLayout, FrameLayout, etc.
- **Common Usage**: To manage the arrangement and interaction of multiple child views.

How to Choose the Right Layout:

- Use **LinearLayout** when views need to be arranged in a simple sequence.
- Use **RelativeLayout** for more complex, flexible positioning of elements.
- Use **ConstraintLayout** for a flat view hierarchy with flexible positioning.
- Use **FrameLayout** for single content display or simple overlay views.
- Use **GridLayout** for organized grid-based arrangements.
- Use **ScrollView** or **HorizontalScrollView** when content might overflow.
- Use **CoordinatorLayout** for advanced interactions like floating action buttons and toolbars.

These layouts offer a variety of ways to organize the UI components of an Android application, and the choice of layout depends on the complexity of the design and the desired behavior.

Chapter-3
Android Development Tools: Detailed Overview

1. **Android Studio:** Android Studio is the official integrated development environment (IDE) for Android application development. It is based on IntelliJ IDEA and provides all the tools required to develop Android apps, including a code editor, debugging tools, performance analyzers, and more.

Key Features:

- **Code Editor**: Offers syntax highlighting, code completion, and refactoring features for Java, Kotlin, XML, and other languages.
- **Layout Editor**: A drag-and-drop interface to design app UIs visually.
- **Built-in Emulator**: A virtual device for testing your app without needing a physical device.
- **Git Integration**: Provides version control tools for managing project code.
- **Profiling Tools**: Used to measure CPU, memory, and network usage of your app.

Example: You can use Android Studio to write code for a *Weather App*, designing the layout for the home screen, writing Kotlin code for fetching data from an API, and using the built-in emulator to test the app on various devices.

2. Java Development Kit (JDK): The JDK is a software development kit used to compile and run Java applications. It includes libraries, the Java runtime environment (JRE), and the Java compiler needed for compiling Java code into bytecode.

Key Features:

- **Compiler**: Translates Java code into bytecode that the Java Virtual Machine (JVM) can execute.
- **JRE**: Provides the environment for running Java applications.
- **Libraries**: Includes standard Java libraries to manage tasks such as data handling, networking, and UI creation.

Example: If you're building an Android app in Java, the JDK will be required to compile the Java files into an APK that can be installed and run on Android devices.

3. Gradle: Gradle is an open-source build automation tool used by Android Studio to manage project dependencies, build configurations, and automate the build process. It handles everything from compiling source code to packaging the app.

Key Features:

- **Dependency Management**: Gradle can automatically download and manage libraries like Retrofit, Gson, etc.
- **Build Variants**: It allows you to define different build configurations, such as debug and release.
- **Customizable**: You can customize the build process with your scripts to perform tasks such as testing and deployment.

Example: You can configure Gradle to include dependencies like Firebase for authentication or Glide for image loading. When you click "Build" in Android Studio, Gradle compiles the code, runs unit tests, and creates the APK for deployment.

4. Android Virtual Device (AVD): An AVD is an emulator configuration that mimics the hardware and software features of an Android device. It allows developers to run their apps in a virtual environment for testing without needing a physical device.

Key Features:

- **Customizable Device Profiles**: You can define various screen sizes, Android versions, and other hardware specifications.

- **Snapshot Support**: Allows you to save the current state of the emulator and quickly resume testing from where you left off.
- **Multiple Devices**: Test your app on different Android versions, screen sizes, and device types (e.g., phone, tablet).

Example: Testing a *News Reader App* on various devices and Android versions using AVD ensures that your app will run smoothly on different phones and tablets, even if you don't own those devices.

5. Git: Git is a distributed version control system used to track changes in source code during software development. It allows teams of developers to collaborate by providing features like branching, merging, and committing code changes.

Key Features:

- **Branching**: Developers can create branches to work on features or fixes without affecting the main codebase.
- **Commit History**: Git keeps a detailed history of code changes, making it easy to track modifications and revert to previous versions.
- **Collaboration**: Multiple developers can work on the same project and merge their changes.

Example: If you're working with a team to develop a *Recipe App*, Git allows you to collaboratively make changes, manage branches for different features (e.g., "UI changes" or "API integration"), and easily merge those changes into the main branch.

6. Firebase: Firebase is a suite of cloud-based services that provides backend solutions for mobile apps, including real-time databases, authentication, and cloud messaging.

Key Features:

- **Firebase Realtime Database**: A cloud-based NoSQL database that allows you to store and sync data in real-time.
- **Authentication**: Easy-to-integrate sign-in services, including email/password, Google, Facebook, etc.
- **Cloud Firestore**: A flexible, scalable database for storing and syncing data.
- **Crashlytics**: Real-time crash reporting and analytics.

Example: In a *Chat App*, Firebase can handle user authentication, storing messages, and updating chat data in real-time across devices. You can also use Firebase Cloud Messaging (FCM) to send push notifications to users.

7. XML (Extensible Markup Language): XML is used in Android for defining the layout and structure of the app's user interface. Every UI component in Android, such as buttons, text fields, and images, is defined using XML.

Key Features:

- **Declarative UI**: XML allows you to declaratively define the UI structure of your app.
- **Layout Management**: You can define complex layouts, such as linear or relative layouts, to arrange UI components.
- **Styling**: XML is also used to define styles and themes for your app's UI elements.

Example: You can create an XML layout file for a login screen with EditText views for the username and password, and a Button for submitting the login information.

8. Kotlin: Kotlin is a modern, statically typed programming language that runs on the Java Virtual Machine (JVM). It is officially supported for Android development and is designed to be more concise and expressive than Java.

Key Features:

- **Concise Syntax**: Kotlin reduces boilerplate code, making development faster.

- **Interoperability**: Kotlin is fully interoperable with Java, so you can mix and match Kotlin and Java code within the same project.
- **Null Safety**: Kotlin helps prevent null pointer exceptions by incorporating null safety features directly into the language.

Example: Instead of writing lengthy Java code, you can use Kotlin to create a simple data class to hold user information, or use features like coroutines for handling background tasks.

9. Android Debug Bridge (ADB): ADB is a command-line tool that allows developers to communicate with Android devices for debugging purposes. It can be used to install and manage apps, retrieve logs, and run various commands on the device.

Key Features:

- **Device Management**: Install, uninstall, or transfer files to/from a connected device.
- **Debugging**: View system logs (logcat) to debug app behavior.
- **Shell Commands**: Run various system-level commands directly on the Android device.

Example: You can use ADB to install an APK on your device, view logs to diagnose issues, or run shell commands to emulate user input for automated testing.

10. ProGuard: ProGuard is a code shrinker, optimizer, and obfuscator used in Android app development. It helps reduce the size of the app and obscure code to protect intellectual property.

Key Features:

- **Code Shrinking**: Removes unused code, making the app smaller and faster.
- **Obfuscation**: Renames classes, methods, and variables to make the code harder to reverse-engineer.
- **Optimization**: Optimizes bytecode to improve performance.

Example: In a commercial game app, you can use ProGuard to reduce the APK size and protect your game logic and assets from being easily reverse-engineered.

11. SQLite: SQLite is a self-contained, serverless SQL database engine used to store app data locally on Android devices.

Key Features:

- **Local Database**: SQLite allows your app to store data persistently without requiring an internet connection.
- **Lightweight**: It is fast, small, and requires minimal setup.
- **SQL Queries**: You can perform complex queries on data using SQL syntax.

Example: You could use SQLite in a *Notes App* to store user notes locally on the device. The app could save, update, and delete notes by interacting with an SQLite database.

12. Postman: Postman is an API testing tool used to make requests to web services, examine responses, and debug APIs. It is often used for testing the backend services that your app will communicate with.

Key Features:

- **Request Creation**: Send HTTP requests (GET, POST, PUT, DELETE) to test API endpoints.
- **Response Analysis**: Inspect responses, including headers, status codes, and body data.
- **Automated Testing**: Automate API tests with collections and monitor API performance.

Example: You can use Postman to test the API of an e-commerce app that retrieves product details from the server. You can check the responses, such as status codes and JSON data, before integrating them into your app.

13. LeakCanary: LeakCanary is a memory leak detection library for Android that helps identify and fix memory leaks in an app.
Key Features:

- **Automatic Leak Detection**: Identifies memory leaks when objects are not garbage collected.
- **Leak Analysis**: Provides detailed reports and stack traces to help you locate and fix memory leaks.
- **Real-Time Monitoring**: It runs in the background while you test your app to detect leaks.

Example: If your *Photo Gallery App* is using large images and you suspect a memory leak, LeakCanary can help you identify where the app is holding onto memory unnecessarily.

14. HAXM (Intel Hardware Accelerated Execution Manager): HAXM is an Intel driver that enables hardware acceleration for Android emulators, speeding up emulator performance on Intel CPUs.
Key Features:

- **Hardware Acceleration**: Improves the performance of Android emulators by using Intel's hardware virtualization features.
- **Faster Emulation**: Reduces the time it takes to load and interact with an emulator.

Example: By enabling HAXM, you can run an Android emulator for a *Music Streaming App* much faster, making testing and development more efficient.

15. Crashlytics: Crashlytics is a real-time crash reporting tool provided by Firebase. It helps monitor the stability of your app by reporting crashes as they occur in real-time.
Key Features:

- **Real-Time Crash Reports**: Automatically reports when an app crashes along with detailed stack traces.
- **Crash Insights**: Provides insights into the causes of crashes, including device information and system state.
- **Performance Monitoring**: Tracks app performance and helps identify slow operations or bottlenecks.

Example: After launching a new version of a *Fitness Tracker App*, you can use Crashlytics to monitor and quickly respond to any crashes experienced by users.

Components of an Android application

An Android application is composed of several key components that work together to provide functionality and interactivity. These components include activities, services, content providers, broadcast receivers, and other resources.

1. Activity

Definition: An Activity represents a single screen in an Android application. It serves as the entry point for user interactions and typically contains the user interface (UI).
Key Features:

- An activity is where users interact with your app (e.g., a login screen, main screen, settings screen).
- Each activity can contain one or more UI elements (buttons, text fields, etc.).
- Activities are managed in a stack-like structure called the **Activity Stack**. When you navigate between screens, the previous activity goes into the stack.

Example: In a *Weather App*, you might have an activity for showing the current weather, another for displaying the forecast, and another for settings.
Code Example:

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

2. Service

Definition: A Service is a component that runs in the background to perform long-running tasks, like playing music, downloading files, or processing data, without interacting directly with the user interface.

Key Features:

- Services run independently of user interactions.
- They can continue to run even if the activity that started them is destroyed (e.g., downloading a file in the background).
- Services can be either **started** (using startService()) or **bound** (using bindService()).

Example: In a *Music App*, a service may play music in the background even when the user navigates to other screens.

Code Example:

```
class MusicService : Service() {
    override fun onBind(intent: Intent?): IBinder? {
        return null // Not a bound service
    }
    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        // Play music
        return START_STICKY
    }
}
```

3. Broadcast Receiver

Definition: A BroadcastReceiver listens for and responds to global system-wide broadcast announcements (e.g., device power-up, network changes, or app-specific notifications).

Key Features:

- Broadcast receivers don't have a user interface. They are used to handle asynchronous events.
- You can register broadcast receivers in two ways: either **statically** (via the AndroidManifest.xml) or **dynamically** (via code).

Example: A BroadcastReceiver could listen for a change in network connectivity or when the battery is low, and trigger appropriate actions like showing a notification or pausing the music.

Code Example:

```
class BatteryReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        val level = intent.getIntExtra(BatteryManager.EXTRA_LEVEL, -1)
        // Handle battery level change
    }
}
```

4. Content Provider

Definition: A ContentProvider manages access to a structured set of data. It provides a mechanism for sharing data between different applications.

Key Features:

- Content providers abstract the data storage and allow applications to access shared data in a secure manner.
- They work with URIs (Uniform Resource Identifiers) to provide access to data stored in databases, files, or even over the network.

Example: The **Contacts** app uses a content provider to share contact data with other apps (like messaging or email apps).

Code Example:

```
val cursor = contentResolver.query(
    ContactsContract.Contacts.CONTENT_URI,
    null, null, null, null
)
```

5. Intent

- **Definition**: An Intent is a messaging object used to request an action or communication between components in an Android app.
- **Key Features**:
    - **Explicit Intent**: Used to start a specific activity or service (e.g., starting a specific screen in your app).
    - **Implicit Intent**: Used to request an action without specifying the exact component (e.g., opening a webpage or sharing content).
- **Example**: You can use an implicit intent to open a web page in a browser:

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("http://www.example.com"))
startActivity(intent)
```

6. Fragments

- **Definition**: A Fragment represents a portion of the user interface or behavior that can be placed inside an Activity. Fragments are often used to create dynamic and flexible UIs, especially on tablets where multiple fragments can be displayed at once.
- **Key Features**:
    - A fragment has its own lifecycle, and can be added, removed, or replaced dynamically during the activity lifecycle.
    - They are modular, and the same fragment can be reused in different activities.
- **Example**: In a *News App*, one fragment might display the list of articles, and another fragment might show detailed information about a specific article.
- **Code Example**:

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

7. UI Components

- **Definition**: UI components in Android are various elements used to design the user interface, such as buttons, text fields, and images.
- **Key Features**:
    - UI components are defined in XML files and are typically placed within activities or fragments.
    - Android provides a wide variety of views like TextView, Button, ImageView, RecyclerView, etc.

- **Example**: In a *ToDo List App*, you might use a RecyclerView to display a list of tasks and EditText to allow the user to add new tasks.
- **Code Example**:

```
<Button
   android:id="@+id/addTaskButton"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Add Task"/>
```

8. AndroidManifest.xml

- **Definition**: The AndroidManifest.xml file is the essential configuration file that declares the components of the app and system features it requires.
- **Key Features**:
  - It defines activities, services, content providers, and broadcast receivers in your app.
  - It also declares permissions, such as access to the internet, camera, or location services.
- **Example**: The manifest file may include permissions for accessing the camera or internet:
- `<uses-permission android:name="android.permission.CAMERA"/>`
- `<uses-permission android:name="android.permission.INTERNET"/>`

9. Resources

- **Definition**: Resources in Android include all the non-code files, such as images, strings, layouts, styles, and raw files that are used by the app.
- **Key Features**:
  - **Drawable**: For images and graphics (e.g., icons).
  - **String**: For storing text, which can be localized.
  - **Layout**: For defining the UI of activities and fragments.
- **Example**: Storing text for different languages in res/values/strings.xml:
- `<string name="app_name">My First App</string>`

An Android application is made up of multiple components that interact with each other. These components allow the app to perform specific tasks, manage data, and provide a user interface. Here's a summary of the main components:

1. **Activity**: UI screen and user interaction.
2. **Service**: Background tasks.
3. **Broadcast Receiver**: Handles system or app-wide events.
4. **Content Provider**: Manages shared data.
5. **Intent**: Facilitates communication between components.
6. **Fragment**: Modular UI components.
7. **UI Components**: Views and layouts for interaction.
8. **AndroidManifest.xml**: Configuration file for declaring components and permissions.
9. **Resources**: Non-code assets like images and strings.

Create your first Android application

To create your first Android application, follow these steps using **Android Studio**, the official Integrated Development Environment (IDE) for Android development.

Step 1: Install Android Studio

1. **Download Android Studio** from the official site https://developer.android.com/studio

2. **Install** Android Studio by following the on-screen instructions for your operating system (Windows, macOS, or Linux).
3. Launch **Android Studio** after installation.

Step 2: Start a New Project

1. Open **Android Studio**.
2. Click on **Start a new Android Studio project**.
3. Choose a **Project Template**: Select **Empty Activity** (this will create a basic app with one screen).
4. Click **Next**.

Step 3: Configure Your Project

1. **Name Your Project**: For example, name it "MyFirstApp".
2. **Package Name**: Choose a unique package name (e.g., com.example.myfirstapp).
3. **Save Location**: Choose the directory where you want your project to be saved.
4. **Language**: Select **Kotlin** (recommended) or **Java**.
5. **Minimum API Level**: Choose the **API level** (API 21 or higher is recommended).
6. Click **Finish**.

Step 4: Understand the Project Structure

Your project will have the following main components:

- **MainActivity.kt** (or MainActivity.java for Java): This is where the logic for the main screen of your app resides.
- **activity_main.xml**: This is the layout file where you define the UI components (buttons, text fields, etc.).
- **Gradle**: Used for building the app and managing dependencies.

Step 5: Modify the Layout (XML)

1. Open the activity_main.xml file located in the res/layout folder.
2. Replace the existing XML code with the following to add a simple button and a text view:

```
<?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/helloTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, Android!"
        android:textSize="24sp"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="200dp"/>

    <Button
        android:id="@+id/helloButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me"
        android:layout_below="@id/helloTextView"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="30dp"/>
```

```
</RelativeLayout>
```
This XML code creates a TextView and a Button. The TextView displays a welcome message, and the Button allows user interaction.

Step 6: Add Logic to the Button (Kotlin)

1. Open the MainActivity.kt (or MainActivity.java) file.
2. Inside the onCreate() method, add logic to make the button change the text when clicked:

```kotlin
package com.example.myfirstapp

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

class MainActivity : AppCompatActivity() {

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

        val helloTextView = findViewById<TextView>(R.id.helloTextView)
        val helloButton = findViewById<Button>(R.id.helloButton)

        helloButton.setOnClickListener {
            helloTextView.text = "You clicked the button!"
        }
    }
}
```
Here's what the code does:

- **findViewById()**: Gets references to the TextView and Button.
- **setOnClickListener()**: Sets an action for the button when it is clicked. When clicked, it changes the text of the TextView.

Step 7: Run Your Application

1. **Connect a Device** or use the **AVD (Android Virtual Device)**:
   - You can either connect a physical Android device or use the AVD to run the app in an emulator.
   - To use the AVD, click on the **AVD Manager** in Android Studio and create an emulator for testing.
2. **Run the App**:
   - Click the green **Run** button (or press Shift + F10).
   - Choose the device/emulator you want to run the app on.

Step 8: See the Results

Once the app is installed on the device or emulator:

- You will see a screen with the message "Hello, Android!" and a **Click Me** button.
- When you click the button, the message will change to "You clicked the button!".

Notifications

In Android, **notifications** are messages that pop up outside of your app's UI to inform users of events or updates, such as new messages, emails, or background tasks. Notifications are typically shown in the system's status bar and can also appear as a popup or in the notification shade.

1. Types of Notifications

Android supports different types of notifications:

- **Basic Notifications**: A simple notification with text, icon, and a click action.
- **Big Text Notifications**: Notifications with a larger body of text (useful for news updates, messages, etc.).
- **Big Picture Notifications**: Notifications that display an image along with text.
- **Inbox Style Notifications**: Display a list of messages or items in the notification itself.
- **Messaging Style Notifications**: For real-time messaging apps to display a conversation.
- **Custom Notifications**: Fully customizable notifications for complex use cases.

2. Basic Structure of a Notification

A basic notification typically has the following components:

- **Title**: The heading or brief description of the notification.
- **Text**: A longer message or detail about the notification.
- **Icon**: A small image that represents the app or type of notification.
- **Action**: Buttons or other interactive elements that the user can click.
- **Priority**: Defines the importance of the notification, determining how it appears (e.g., high priority for urgent messages).

3. Basic Notification in Java

In Android, you can create and display notifications using the NotificationManager class and NotificationCompat.Builder for backward compatibility.

Code Example: Basic Notification in Java

```
import android.app.Notification;
import android.app.NotificationChannel;
import android.app.NotificationManager;
import android.content.Context;
import android.os.Build;
import androidx.core.app.NotificationCompat;

public class MainActivity extends AppCompatActivity {

    private static final String CHANNEL_ID = "default_channel";

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

        // Create Notification Channel (required for Android 8.0 and higher)
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            NotificationChannel channel = new NotificationChannel(
                    CHANNEL_ID,
                    "Default Channel",
                    NotificationManager.IMPORTANCE_DEFAULT
            );
            NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
```

```
            notificationManager.createNotificationChannel(channel);
        }

        // Build the notification
        Notification notification = new NotificationCompat.Builder(this, CHANNEL_ID)
                .setSmallIcon(R.drawable.ic_notification)  // Icon for the notification
                .setContentTitle("Hello, User!")
                .setContentText("This is your first notification.")
                .setPriority(NotificationCompat.PRIORITY_DEFAULT)
                .setAutoCancel(true)  // Automatically remove the notification when tapped
                .build();

        // Show the notification
        NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
        notificationManager.notify(1, notification);  // '1' is the notification ID
    }
}
```

Explanation:

- **NotificationChannel**: A channel groups notifications in Android 8.0 (Oreo) and higher. Channels define the importance of the notification (e.g., sound, vibration).
- **NotificationCompat.Builder**: A builder used to construct the notification with various options (like title, text, icon, etc.).
- **notify()**: Displays the notification with a unique ID (1 in this case).

2. Adding Action Buttons to Notification in Java

You can add action buttons to the notification in Java, allowing users to perform specific tasks like replying to a message or opening an activity.

Code Example: Adding Action Buttons to Notification

```
import android.app.PendingIntent;
import android.content.Intent;
import androidx.core.app.NotificationCompat;

public class MainActivity extends AppCompatActivity {

    private static final String CHANNEL_ID = "default_channel";

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

        // Create Notification Channel
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            NotificationChannel channel = new NotificationChannel(
                    CHANNEL_ID,
                    "Default Channel",
                    NotificationManager.IMPORTANCE_DEFAULT
            );
            NotificationManager notificationManager = (NotificationManager) 
getSystemService(Context.NOTIFICATION_SERVICE);
            notificationManager.createNotificationChannel(channel);
        }

        // Create an Intent to be fired when the action button is clicked
        Intent replyIntent = new Intent(this, ReplyActivity.class);
        PendingIntent replyPendingIntent = PendingIntent.getActivity(this, 0, replyIntent, 
PendingIntent.FLAG_UPDATE_CURRENT);

        // Create the action
        NotificationCompat.Action action = new NotificationCompat.Action.Builder(
                R.drawable.ic_reply, "Reply", replyPendingIntent)
                .build();

        // Build the notification
        Notification notification = new NotificationCompat.Builder(this, CHANNEL_ID)
                .setSmallIcon(R.drawable.ic_notification)
                .setContentTitle("New Message")
                .setContentText("You have a new message.")
                .addAction(action)  // Add action to the notification
                .setPriority(NotificationCompat.PRIORITY_DEFAULT)
                .setAutoCancel(true)
                .build();

        // Show the notification
        NotificationManager notificationManager = (NotificationManager) 
getSystemService(Context.NOTIFICATION_SERVICE);
        notificationManager.notify(1, notification);  // '1' is the notification ID
    }
}
```

In this example:

- **PendingIntent**: It wraps an intent that will be fired when the action button is pressed (e.g., opening a new activity).
- **addAction()**: Adds an action button to the notification.

3. Big Text Notification in Java

Big text notifications allow you to display a longer body of text in the notification.

Code Example: Big Text Notification

```
import androidx.core.app.NotificationCompat;

public class MainActivity extends AppCompatActivity {

    private static final String CHANNEL_ID = "default_channel";

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

        // Create Notification Channel
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            NotificationChannel channel = new NotificationChannel(
                    CHANNEL_ID,
                    "Default Channel",
                    NotificationManager.IMPORTANCE_DEFAULT
            );
            NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
            notificationManager.createNotificationChannel(channel);
        }

        // Create BigTextStyle for longer text
        NotificationCompat.BigTextStyle bigTextStyle = new
NotificationCompat.BigTextStyle()
                .bigText("This is a long body of text that will be shown in a bigger format in the
notification.");

        // Build the notification
        Notification notification = new NotificationCompat.Builder(this, CHANNEL_ID)
                .setSmallIcon(R.drawable.ic_notification)
                .setContentTitle("Big Text Notification")
                .setStyle(bigTextStyle)  // Apply the BigTextStyle
                .setPriority(NotificationCompat.PRIORITY_DEFAULT)
                .setAutoCancel(true)
                .build();

        // Show the notification
        NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
        notificationManager.notify(1, notification);  // '1' is the notification ID
    }
}
```

4. Open an Activity when Notification is Clicked in Java

You can specify what happens when the user clicks on a notification, such as opening a specific activity.

Code Example: Open an Activity When Notification is Clicked

```
import android.app.PendingIntent;
import android.content.Intent;
import androidx.core.app.NotificationCompat;

public class MainActivity extends AppCompatActivity {

    private static final String CHANNEL_ID = "default_channel";

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

        // Create Notification Channel
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            NotificationChannel channel = new NotificationChannel(
                    CHANNEL_ID,
                    "Default Channel",
                    NotificationManager.IMPORTANCE_DEFAULT
            );
            NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
            notificationManager.createNotificationChannel(channel);
        }

        // Create an Intent to open the MainActivity
        Intent intent = new Intent(this, MainActivity.class);
        PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent,
PendingIntent.FLAG_UPDATE_CURRENT);

        // Build the notification
        Notification notification = new NotificationCompat.Builder(this, CHANNEL_ID)
                .setSmallIcon(R.drawable.ic_notification)
                .setContentTitle("Notification Click")
                .setContentText("Tap to open the main activity")
                .setContentIntent(pendingIntent)  // Open MainActivity when clicked
                .setAutoCancel(true)
                .build();

        // Show the notification
        NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
        notificationManager.notify(1, notification);  // '1' is the notification ID
    }
}
```
5. Handling Background Notifications (Firebase Cloud Messaging - FCM) in Java
For sending push notifications through **Firebase Cloud Messaging (FCM)**, you would use Firebase services. Here's a quick guide:

1. **Add Firebase dependencies** to your build.gradle file.
2. **Configure Firebase** in your AndroidManifest.xml.
3. **Set up FCM** on your Firebase console.
4. **Send and receive notifications**.

This would involve additional setup, such as using FirebaseMessagingService to receive notifications and push them to the system notification tray.

Exercise

Very Short Questions (1 Mark each)
Multiple-Choice Questions (MCQs)

1. **Which class is used to create and manage notifications in Android?**
   a) NotificationManager
   b) NotificationCompat.Builder
   c) NotificationChannel
   d) All of the above
2. **What is the purpose of a NotificationChannel in Android?**

a) To display notifications
b) To categorize and manage notification behaviors (e.g., sound, vibration)
c) To open an activity when the notification is clicked
d) None of the above

3. **Which method is used to show a notification in Android?**
   a) notify()
   b) show()
   c) create()
   d) setNotification()
4. **In which scenario would you use PendingIntent?**
   a) To display notifications
   b) To send notifications to other apps
   c) To wrap an Intent and pass it to another component
   d) None of the above
5. **What does setAutoCancel(true) do in a notification?**
   a) Keeps the notification visible even after the user taps it
   b) Automatically cancels the notification once it is clicked
   c) Allows the notification to cancel on its own
   d) None of the above

True/False Questions

1. Notifications in Android are only visible when the app is open.
2. A **NotificationChannel** is required for Android versions lower than 8.0.
3. The **BigTextStyle** in Android is used to display a large body of text in a notification.
4. Action buttons can be added to notifications to allow users to take specific actions like opening an activity.
5. Notifications are not visible in the status bar unless they have a higher priority.

Very Short Questions (1 Mark each)

1. What is a notification in Android?
2. Which method is used to create a **NotificationChannel**?
3. What is the role of **NotificationManager** in Android?
4. Which class is used to set the title and text of a notification?
5. What does **setSmallIcon()** do in a notification?

Short Questions (2 Mark each)

1. What is the purpose of a notification channel in Android 8.0 and above?
2. Explain the use of **PendingIntent** in notifications.
3. What is the difference between **setPriority()** and **setAutoCancel()** in a notification?
4. What are the various styles available for notifications in Android? Provide an example.
5. How can you add multiple actions to a notification?

long Questions (5 Mark each)

1. Explain how to create and display a basic notification in Android using Java.
2. Describe the process of adding action buttons to a notification in Android.
3. How does the **BigTextStyle** enhance the presentation of a notification? Provide an example of usage.
4. What is the role of **ContentIntent** in a notification, and how does it help in managing user interaction?
5. Describe the steps to implement a Firebase Cloud Messaging (FCM) notification in an Android app.

Very long Questions (10 Mark each)

1. Describe in detail the components of a notification in Android. Include all the options for customizing notifications, such as text, icons, action buttons, styles, etc.
2. Explain the notification management system in Android, including notification channels, their importance, and how to handle notifications in different versions of Android.
3. Create an Android application in Java that implements push notifications using Firebase Cloud Messaging (FCM). Include all necessary code and setup steps for both the client-side (Android) and the server-side (Firebase Console).
4. Discuss the importance and implementation of background notifications in Android. How can they be used to update the user about events even when the app is in the background?
5. Compare and contrast the different notification styles in Android, such as **BigTextStyle**, **BigPictureStyle**, and **InboxStyle**. Provide examples and use cases for each style.

Answer Key

MCQs:

1. d) All of the above
2. b) To categorize and manage notification behaviors (e.g., sound, vibration)
3. a) notify()
4. c) To wrap an Intent and pass it to another component
5. b) Automatically cancels the notification once it is clicked

True/False:

1. False
2. False
3. True
4. True
5. True

Introduction to Activities in Android

What is an Activity?

An **Activity** in Android represents a single screen in an application where users interact with the app. It acts as the entry point for user actions and typically contains a user interface (UI). Activities handle user interactions and transitions between different screens.

For example, in a messaging app:

- The **main activity** displays the list of conversations.
- A **chat activity** shows individual messages.
- A **settings activity** allows users to customize preferences.

Each screen (activity) works independently but can communicate with others using **Intents**.

Key Features of an Activity

1. **User Interface (UI) Management** – Defines how the screen looks and responds to user input.
2. **Lifecycle Management** – Android automatically handles different states (e.g., running, paused, stopped).
3. **Navigation** – Activities allow seamless transitions between different screens.
4. **Multiple Activities** – Apps can consist of multiple activities that interact with each other.
5. **Event Handling** – Activities process user inputs like button clicks, swipes, and gestures.

Activity Lifecycle

Android manages activity states using predefined lifecycle methods. The key lifecycle methods are:

Method	**Description**
onCreate()	Called when the activity is created. Used for initialization.
onStart()	Called when the activity becomes visible to the user.
onResume()	Called when the activity is in the foreground and interactive.

Method	Description
onPause()	Called when the activity is partially hidden (e.g., a new activity is opening).
onStop()	Called when the activity is no longer visible.
onDestroy()	Called before the activity is destroyed.
onRestart()	Called when the activity is restarted after being stopped.

Creating a Simple Activity

An activity consists of:

1. **Java/Kotlin Code** – Defines behavior.
2. **XML Layout** – Defines UI elements.

Java Example: MainActivity.java

```
package com.example.myapp;

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); // Link XML layout
    }
}
```

XML 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">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to My App!" />

</LinearLayout>
```

Navigating Between Activities

To move from one activity to another, use **Intents**.

Example: Open SecondActivity from MainActivity

1. Create a Second Activity

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

2. Use Intent to Navigate

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

3. Declare Activity in AndroidManifest.xml

```
<activity android:name=".SecondActivity" />
```

Types of Activities in Android

Android provides different types of activities, each serving a unique purpose in an application. These activities help in designing structured and user-friendly apps by managing user interactions effectively.

1. Main (Launcher) Activity

- The entry point of an application, launched when the app is opened.
- Defined in AndroidManifest.xml with <intent-filter> as MAIN and LAUNCHER.
- Example: **Home screen, splash screen**.

Example: Defining MainActivity in Manifest

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

2. Full-Screen Activity

- Used for immersive experiences like games, videos, or splash screens.
- Hides the status and navigation bars for a full-screen experience.

Example: Full-Screen Mode in Activity

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
            WindowManager.LayoutParams.FLAG_FULLSCREEN);
    setContentView(R.layout.activity_fullscreen);
}
```

3. List Activity

- Displays a list of items using ListView or RecyclerView.
- Used for showing dynamic data like contacts, messages, or product lists.

Example: ListActivity with Simple ListView

```
public class MyListActivity extends ListActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      String[] items = {"Item 1", "Item 2", "Item 3"};
      ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
android.R.layout.simple_list_item_1, items);
      setListAdapter(adapter);
   }
}
```

4. Fragment Activity

- Supports multiple UI components using **fragments**.
- Used in **tablet layouts, dynamic UI, and multi-pane apps**.
- Requires FragmentActivity or AppCompatActivity.

Example: Using FragmentActivity

```
public class MyFragmentActivity extends FragmentActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_fragment);
   }
}
```

5. Dialog Activity

- Used for **pop-ups, alerts, or quick actions** without taking full screen.
- Typically used for **confirmations, user inputs, or warnings**.

Example: Creating a DialogActivity

```
public class MyDialogActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_dialog);
      setFinishOnTouchOutside(false); // Prevent dismissal on outside touch
   }
}
```

Declaring Dialog Theme in Manifest

```
<activity android:name=".MyDialogActivity"
```

```
    android:theme="@android:style/Theme.Dialog"/>
```

6. Tabbed Activity

- Uses **tabs** to switch between multiple screens in the same activity.
- Implemented using TabLayout and ViewPager2.
- Example: **WhatsApp (Chats, Status, Calls sections)**.

Example: Using Tabbed Activity with ViewPager2

```
ViewPager2 viewPager = findViewById(R.id.view_pager);
TabLayout tabLayout = findViewById(R.id.tab_layout);
new TabLayoutMediator(tabLayout, viewPager, (tab, position) -> {
    tab.setText("Tab " + (position + 1));
}).attach();
```

7. Scrolling Activity

- Used for displaying **long content** (e.g., blog articles, news feeds).
- Utilizes ScrollView for smooth scrolling.

Example: Using ScrollView in XML

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

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Long content goes here..." />
</ScrollView>
```

8. Settings Activity

- Displays **preferences and configurations**.
- Uses PreferenceActivity or PreferenceFragmentCompat.
- Example: **App settings, notification preferences**.

Example: Creating a Settings Activity

```
public class MySettingsActivity extends PreferenceActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

Example: Preferences XML File (res/xml/preferences.xml)

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <CheckBoxPreference
        android:key="notifications"
        android:title="Enable Notifications"
        android:defaultValue="true" />
</PreferenceScreen>
```

9. Login Activity

- Used for **authentication** (email/password, Google Sign-In, etc.).
- Redirects users to the main activity after login.

Example: Login Activity UI (activity_login.xml)

```
<EditText
    android:id="@+id/username"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Enter Username"/>

<Button
    android:id="@+id/loginButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Login"/>
```

Handling Login in Java

```
Button loginButton = findViewById(R.id.loginButton);
loginButton.setOnClickListener(v -> {
    Intent intent = new Intent(LoginActivity.this, MainActivity.class);
    startActivity(intent);
});
```

10. WebView Activity

- Displays web content inside an app using WebView.
- Example: **In-app browsers, documentation viewers**.

Example: WebView Activity

```
WebView webView = findViewById(R.id.webview);
webView.getSettings().setJavaScriptEnabled(true);
webView.loadUrl("https://www.google.com");
```

XML Layout (activity_webview.xml)

```
<WebView
    android:id="@+id/webview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Activity Type	Usage Example
Main Activity	Home screen, splash screen
Full-Screen Activity	Games, videos, immersive experiences
List Activity	Contacts list, messages, product catalog

Activity Type	Usage Example
Fragment Activity	Dynamic UIs, multi-pane layouts
Dialog Activity	Pop-ups, confirmation messages
Tabbed Activity	WhatsApp (Chats, Status, Calls)
Scrolling Activity	Long articles, news feeds
Settings Activity	App preferences, notification settings
Login Activity	User authentication (Google, email/password)
WebView Activity	In-app browser, web content

Activity Life Cycle in Android

The **Activity Life Cycle** in Android describes how an activity transitions through different states from its creation to its destruction. Android manages these states using lifecycle callback methods, ensuring efficient resource usage and smooth user experience.

Activity Lifecycle States and Callbacks

An activity goes through the following lifecycle states:

1. Created → onCreate()

When is it called?

- When the activity is first created.
- Used for initializing UI components and logic.

Common Usage:

- Set up layout using setContentView().
- Initialize variables, listeners, and background processes.
- Restore saved state (if applicable).

Example:

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

2. Started → onStart()

When is it called?

- Right after onCreate().

- When the activity becomes visible but is not yet interactive.

Common Usage:

- Register listeners (e.g., GPS, sensors, UI updates).
- Prepare resources needed for user interaction.

Example:

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

3. Resumed (Foreground) → onResume()

When is it called?

- After onStart().
- When the activity is in the foreground and interactive.

Common Usage:

- Start animations, UI updates, or live data fetching.
- Resume video playback or real-time updates.

Example:

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

4. Paused (Partially Hidden) → onPause()

When is it called?

- When another activity comes partially on top (e.g., a dialog or new screen appears).

Common Usage:

- Pause animations, videos, or background music.
- Save unsaved changes in case the app gets killed.

Example:

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

5. Stopped (Completely Hidden) → onStop()

When is it called?

- When the activity is **completely** hidden (e.g., switched to another app).

Common Usage:

- Release large resources (e.g., database connections, network calls).
- Save user data to avoid losing progress.

Example:

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

6. Restarting → onRestart()

When is it called?

- If the activity was stopped and is **coming back**.

Common Usage:

- Refresh UI or reload data if needed.

Example:

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

7. Destroyed → onDestroy()

When is it called?

- When the activity is being **permanently removed** from memory.
- Occurs when the user closes the app or the system needs memory.

Common Usage:

- Release all resources to prevent memory leaks.
- Save user progress in persistent storage (e.g., SharedPreferences).

Example:

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

Activity Lifecycle Flow Diagram

```
        onCreate()
           ↓
        onStart()
           ↓
        onResume()  → (Activity is in the foreground)
           ↓
   (Another activity appears)
           ↓
        onPause()
           ↓
   (App moves to background)
           ↓
        onStop()
           ↓
   (User returns to the app)
           ↓
        onRestart()
           ↓
        onStart()
           ↓
        onResume()
           ↓
   (User closes the app)
           ↓
        onDestroy()
```

Practical Example: Lifecycle Logging

This example logs each lifecycle callback, making it easy to understand the activity flow.

```
package com.example.myapp;

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

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "Activity Lifecycle";

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

    @Override
    protected void onStart() {
        super.onStart();
        Log.d(TAG, "onStart() called");
    }

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

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

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

    @Override
    protected void onRestart() {
        super.onRestart();
        Log.d(TAG, "onRestart() called");
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.d(TAG, "onDestroy() called");
    }
}
```

Key Takeaways

- onCreate() → **Initialize UI, setup resources**.
- onStart() → **Prepare UI, register listeners**.
- onResume() → **Activity is in the foreground** (User interacting).
- onPause() → **Release resources, pause animations** (Partially hidden).

- onStop() → **Save data, release heavy resources** (Completely hidden).
- onRestart() → **Reload UI if needed** (Coming back to the foreground).
- onDestroy() → **Cleanup resources before activity is destroyed**.

Introduction to Intents in Android

What is an Intent?

An **Intent** in Android is a messaging object that facilitates communication between components such as **activities, services, and broadcast receivers**. It is used to **start new activities, pass data, start background services, or trigger system-wide actions**.

Intents allow Android apps to perform inter-component communication (ICC), both within an app and between different apps.

Types of Intents

Intents are classified into **two main types**:

1. Explicit Intent

- Used to **start a specific activity or service** within the same application.
- The target component (Activity, Service) is **directly mentioned**.

Example: Starting a New Activity (Explicit Intent)

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

Example: Passing Data between Activities

```
Intent intent = new Intent(CurrentActivity.this, NewActivity.class);
intent.putExtra("username", "JohnDoe"); // Passing data
startActivity(intent);
```

Receiving Data in New Activity

```
String username = getIntent().getStringExtra("username");
```

2. Implicit Intent

- 📌 Used when we **do not specify** the exact component to handle the action.
- Instead, we define an **action**, and Android decides which app can handle it.

  Example use cases:

- Open a web page
- Share text
- Make a phone call

- Open Google Maps

Example: Opening a Web Page

```
Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse("https://www.google.com"));
startActivity(intent);
```

Example: Sending an Email

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.setType("message/rfc822");
intent.putExtra(Intent.EXTRA_EMAIL, new String[]{"example@gmail.com"});
intent.putExtra(Intent.EXTRA_SUBJECT, "Hello!");
intent.putExtra(Intent.EXTRA_TEXT, "This is a test email.");
startActivity(Intent.createChooser(intent, "Send Email"));
```

Example: Making a Phone Call

```
Intent intent = new Intent(Intent.ACTION_DIAL);
intent.setData(Uri.parse("tel:+1234567890"));
startActivity(intent);
```

Structure of an Intent

An **Intent** object consists of:

- **Component Name** (only for explicit intents)
- **Action** (Defines what the intent will do)
- **Data** (URI for data to be processed)
- **Category** (Additional information about the action)
- **Extras** (Key-value pairs to send additional data)

◈ Example: Creating an Intent with Action and Data

```
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("geo:37.7749,-122.4194")); // Open Google Maps with coordinates
startActivity(intent);
```