Introduction to Android Operating System

This section covers the foundational aspects of the Android operating system, its history, architecture, and setting up the development environment.

History of Android

- **Initial Development:** Android Inc. was founded in October 2003 by Andy Rubin, Rich Miner, Nick Sears, and Chris White in Palo Alto, California. The initial intention was to develop an advanced operating system for digital cameras, but it later shifted to smartphones to compete with Symbian and Microsoft Windows Mobile.
- **Google Acquisition:** Google acquired Android Inc. in July 2005, marking its entry into the mobile operating system market.
- **Open Handset Alliance (OHA):** In 2007, the OHA was formed, a consortium of tech and mobile companies committed to developing open standards for mobile devices. Android was released as the first product of the OHA.
- **First Android Phone:** The HTC Dream (also known as T-Mobile G1) was the first commercially available smartphone running Android, released in October 2008.
- **Version Naming:** Android versions have historically been named after desserts or sweet treats in alphabetical order (Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycomb, Ice Cream Sandwich, Jelly Bean, KitKat, Lollipop, Marshmallow, Nougat, Oreo, Pie). Since Android 10, Google has moved to numerical naming (Android 10, Android 11, etc.).

Various Versions of Android

Android has undergone numerous iterations, each bringing new features, performance improvements, and API enhancements. Some notable versions include:

- **Android 1.0 (2008):** First public release. Basic features like home screen, notifications, Android Market.
- **Android 1.5 Cupcake (2009):** On-screen keyboard, video recording/playback, copy/paste in browser.
- **Android 2.0/2.1 Eclair (2009):** Multiple accounts, Google Maps Navigation, live wallpapers.
- **Android 2.2 Froyo (2010):** JIT compilation for performance, USB tethering, Wi-Fi hotspot.
- **Android 2.3 Gingerbread (2010):** UI refinements, NFC support, improved gaming performance.
- **Android 3.0 Honeycomb (2011):** Tablet-optimized UI, multi-core processor support.
- **Android 4.0 Ice Cream Sandwich (2011):** Unified UI for phones/tablets, Face Unlock, Android Beam (NFC sharing).
- **Android 4.1-4.3 Jelly Bean (2012):** "Project Butter" for UI smoothness, Google Now, actionable notifications.
- **Android 4.4 KitKat (2013):** Lower memory footprint, immersive mode, print framework.
- **Android 5.0 Lollipop (2014):** Material Design, ART runtime (replacing Dalvik), lock screen notifications.
- **Android 6.0 Marshmallow (2015):** Runtime permissions, Doze mode (battery saving), fingerprint support.
- **Android 7.0 Nougat (2016):** Multi-window support, improved notifications, Data Saver.
- **Android 8.0 Oreo (2017):** Picture-in-picture, notification channels, autofill APIs.
- **Android 9 Pie (2018):** Gesture navigation, Adaptive Battery, App Actions, Slices.
- **Android 10 (2019):** Dark Theme, Gesture Navigation (refined), improved privacy controls.

- **Android 11 (2020):** Conversation notifications, chat bubbles, enhanced privacy controls, device controls.
- **Android 12 (2021):** "Material You" design, redesigned widgets, privacy dashboard.
- **Android 13 (2022):** Themed app icons, per-app language preferences, photo picker.
- **Android 14 (2023):** Predictive back gestures, customizable lock screen, improved privacy.

Setting up Android Application Development Environment

The primary tool for Android development is Android Studio, which comes bundled with necessary SDK components.

Downloading JDK and Android Studio

- **JDK (Java Development Kit):** While Android Studio bundles its own OpenJDK, it's good practice to have a compatible JDK installed separately on your system. Android development often uses JDK 11 or later. You can download it from Oracle or OpenJDK distributions like Adoptium (Temurin).
- **Android Studio:** Download the latest version of Android Studio from the official Android Developers website (developer.android.com/studio). It's available for Windows, macOS, and Linux.

Installing and Configuring Android Studio

1. **Installation:**
   - **Windows:** Run the installer `.exe` file and follow the on-screen prompts.
   - **macOS:** Drag the Android Studio application into your Applications folder.
   - **Linux:** Unpack the `.zip` file and run `studio.sh` from the `bin/` directory.
2. **Initial Setup Wizard:**
   - The first time you launch Android Studio, it runs a setup wizard.
   - **SDK Components:** It will prompt you to download essential SDK components, including:
     - **Android SDK Platform-Tools:** Contains `adb` (Android Debug Bridge), `fastboot`, etc.
     - **Android SDK Build-Tools:** For building Android applications.
     - **Android SDK Platform:** For specific Android versions (APIs). Download the latest stable API level and a few older ones for testing compatibility.
     - **Android Emulator:** For running virtual Android devices.
     - **HAXM/Hypervisor (for Intel CPUs) or WHPX (Windows Hypervisor Platform):** Essential for fast emulator performance. Ensure virtualization is enabled in your computer's BIOS/UEFI.
3. **SDK Manager:** Access the SDK Manager (Tools > SDK Manager in Android Studio) to install additional Android SDK versions, SDK Tools, and System Images for emulators as needed.
4. **AVD Manager:** Access the AVD Manager (Tools > Device Manager or AVD Manager) to create and manage Android Virtual Devices (AVDs) for emulation. Select a device definition (e.g., Pixel 6) and a system image (Android version) to create an emulator.

The Developer Workflow

The typical Android application development workflow involves several iterative steps:

1. **Project Setup:** Create a new project in Android Studio, selecting a template (e.g., Empty Activity).
2. **Coding (Kotlin/Java):** Write your application logic in Kotlin or Java.

3. **UI Design (XML):** Design your user interfaces using XML layout files (`.xml`) in Android Studio's Layout Editor or by writing XML directly.
4. **Build:** Android Studio uses Gradle to compile your code, resources, and dependencies into an APK (Android Package Kit) or AAB (Android App Bundle).
5. **Run/Deploy:**
   - **Emulator:** Deploy the app to an Android Virtual Device (AVD).
   - **Physical Device:** Deploy the app to a connected Android device via USB debugging.
6. **Test:** Run unit tests, integration tests, and UI tests. Manually test on various devices and API levels.
7. **Debug:** Use Android Studio's debugger, logcat, and profilers to identify and fix issues.
8. **Iterate:** Repeat steps 2-7, continuously refining and adding features.
9. **Publish:** Prepare the app for release and upload it to the Google Play Store.

Understanding an Android Studio Project

When you create a new Android project in Android Studio, it generates a standard project structure.

Various folders in Android Studio project

- **`.gradle/`:** Contains Gradle wrapper and caches.
- **`.idea/`:** Android Studio's project files.
- **`app/`:** This is the module for your main application source code.
  - **`build/`:** Generated output directory (APKs, AABs, compiled classes).
  - **`libs/`:** Place any third-party JARs manually if not using Gradle dependencies.
  - **`src/`:** Contains your source code and resources.
    - **`androidTest/`:** Unit tests that run on an Android device or emulator.
    - **`main/`:** Primary source set for your app.
      - **`java/` (or `kotlin/`):** Your Java/Kotlin source code (`.java` or `.kt` files). Organized by package names.
      - **`res/`:** Contains application resources (layouts, drawables, strings, etc.).
      - **`AndroidManifest.xml`:** The manifest file for your app.
    - **`test/`:** Unit tests that run on the local JVM.
  - **`build.gradle` (Module: app):** Gradle build script for the `app` module. Defines app-specific dependencies, signing configs, and build types.
- **`gradle/`:** Contains Gradle wrapper files.
- **`build.gradle` (Project: YourProjectName):** Top-level Gradle build script. Defines project-wide configurations and repositories.
- **`gradle.properties`:** Gradle properties (e.g., memory settings).
- **`settings.gradle`:** Defines which modules are included in the project.

Introduction to `AndroidManifest.xml`

The `AndroidManifest.xml` file is a crucial XML file that defines the fundamental characteristics of your application and declares its essential components to the Android system. Every Android app must have one.

- **Package Name:** Unique identifier for your app (e.g., `com.example.myapp`).
- **Application Components:** Declares all the app's components (Activities, Services, Content Providers, Broadcast Receivers). Without these declarations, the system cannot start them.

- **Permissions:** Declares permissions the app requires to access protected parts of the system or other apps (e.g., `android.permission.CAMERA`, `android.permission.INTERNET`).
- **Hardware and Software Features:** Declares hardware or software features the app requires (e.g., `android.hardware.camera`, `android.hardware.location.gps`).
- **Minimum API Level:** `minSdkVersion` - the lowest Android version on which the app can run.
- **Target API Level:** `targetSdkVersion` - the API level your app is designed to run on.
- **Metadata:** Other information like app icon, label, theme.

Example Snippet:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">

    <!-- Declare permissions -->
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.CAMERA" />

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

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

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

        <!-- Declare Broadcast Receivers -->
        <receiver android:name=".MyBroadcastReceiver" android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
            </intent-filter>
        </receiver>

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

Introduction to `res` folder

The `res` (resources) folder contains all the non-code resources for your application, organized into various subdirectories based on type and configuration (e.g., screen density, language).

- **`drawable/`:** Image files (`.png`, `.jpg`, `.gif`, `.xml` for shapes/layers) that can be drawn on the screen.
  - `drawable-hdpi`, `drawable-mdpi`, `drawable-xhdpi`, `drawable-xxhdpi`, `drawable-xxxhdpi`: For different screen pixel densities.
- **`layout/`:** XML files that define the user interface layouts of your Activities or Fragments (`.xml`).
- **`mipmap/`:** Launcher icons (`.png`). Optimized for app launchers, also supports density-specific folders.
- **`values/`:** XML files containing simple values:
  - `colors.xml`: Custom colors.
  - `strings.xml`: Text strings. Essential for internationalization.
  - `styles.xml`: Themes and styles for your UI.
  - `dimens.xml`: Dimension values (e.g., `dp`, `sp`).
- **`menu/`:** XML files defining menus (e.g., options menu, context menu).
- **`xml/`:** Arbitrary XML files (e.g., preferences, search configurations, remote configurations).
- **`raw/`:** Arbitrary raw files (e.g., audio, video).

Gradle

Gradle is the build automation system used by Android Studio. It's a powerful and flexible tool for managing dependencies, compiling code, packaging, and automating various aspects of the build process.

- **Build Scripts (`.gradle` files):** Written in Groovy (default) or Kotlin DSL.
- **Domain Specific Language (DSL):** Provides a concise and expressive way to describe builds.
- **Plugins:** Extend Gradle's functionality (e.g., Android Gradle Plugin for Android-specific tasks).
- **Dependency Management:** Automatically downloads and manages libraries from repositories.

Various aspects of `build.gradle` file

There are typically two `build.gradle` files in an Android Studio project:

1. **Project-level `build.gradle` (Top-level):**

   - Defines **repositories** for plugins and dependencies (e.g., `google()`, `mavenCentral()`).

   - Declares **dependencies for Gradle plugins** (e.g., Android Gradle Plugin, Kotlin Gradle Plugin).

   - Example:

   ```
   // Top-level build file where you can add configuration options common
   to all sub-projects/modules.
   plugins {
       id 'com.android.application' version '8.0.0' apply false
   ```

```
    id 'com.android.library' version '8.0.0' apply false
    id 'org.jetbrains.kotlin.android' version '1.8.0' apply false
}
buildscript {
    repositories {
        google()
        mavenCentral()
    }
}

allprojects {
    repositories {
        google()
        mavenCentral()
    }
}
```

2. **Module-level `build.gradle` (`app/build.gradle`):**

   - **`plugins`:** Applies the necessary plugins for this module (e.g., `com.android.application` for an app, `org.jetbrains.kotlin.android` if using Kotlin).
   - **`android` block:**
     - **`namespace`:** The package name for your app.
     - **`compileSdk`:** The API level Gradle should use to compile your app.
     - **`defaultConfig`:**
       - `applicationId`: Unique ID for your app on Google Play.
       - `minSdk`: Minimum API level your app supports.
       - `targetSdk`: API level your app is tested against.
       - `versionCode`: Internal version number (incremental).
       - `versionName`: Public-facing version string.
       - `testInstrumentationRunner`: Test runner for Android tests.
     - **`buildTypes`:** Configures build variants (e.g., `debug`, `release`).
       - `minifyEnabled`: Whether to enable code shrinking (ProGuard/R8).
       - `signingConfig`: For signing the APK/AAB.
     - **`packagingOptions`:** Configure how files are packaged.
     - **`kotlinOptions`:** Kotlin-specific compiler options.
   - **`dependencies` block:** Declares all libraries your app depends on.
     - `implementation`: Adds dependency to the compile classpath and packaging. (Recommended for most cases).
     - `testImplementation`: For local JVM tests.
     - `androidTestImplementation`: For instrumented Android tests.
     - `debugImplementation`, `releaseImplementation`: For specific build types.
   - Example:

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
}

android {
    namespace 'com.example.myapp'
    compileSdk 34 // Example: Android 14

    defaultConfig {
        applicationId "com.example.myapp"
        minSdk 24 // Example: Android 7.0 Nougat
        targetSdk 34
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner
"androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-
optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = '1.8'
    }
    buildFeatures {
        viewBinding true // Enable View Binding for cleaner UI code
    }
}

dependencies {
    // AndroidX libraries
    implementation 'androidx.core:core-ktx:1.9.0'
    implementation 'androidx.appcompat:appcompat:1.6.1'
    implementation 'com.google.android.material:material:1.10.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'

    // Kotlin coroutines
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-
android:1.7.1'

    // Testing
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.5'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
}
```

Introduction to Kotlin Programming Language

Kotlin is a modern, statically typed programming language developed by JetBrains. It runs on the Java Virtual Machine (JVM) and is fully interoperable with Java code. In 2019, Google announced Kotlin as the preferred language for Android app development.

Advantages of Kotlin over Java

- **Conciseness:** Kotlin requires less boilerplate code compared to Java, leading to more readable and maintainable code.
- **Null Safety:** Built-in features to help eliminate `NullPointerException`s, a common source of bugs in Java.
- **Interoperability with Java:** Seamlessly call Java code from Kotlin and vice-versa, allowing gradual adoption in existing Java projects.
- **Coroutines:** First-class support for asynchronous programming, making it easier to write non-blocking code for UI responsiveness and background tasks.
- **Extension Functions:** Ability to add new functions to existing classes without modifying their source code.
- **Data Classes:** Simplified creation of classes primarily used to hold data, automatically generating `equals()`, `hashCode()`, `toString()`, and `copy()`.
- **Smart Casts:** Automatically casts variables to a specific type after a type check (e.g., `if (obj is String) { println(obj.length) }`).
- **Functional Programming Support:** Features like higher-order functions, lambda expressions.
- **Community and Tooling:** Strong community support and excellent tooling integration in Android Studio.

Basic Kotlin Syntax

- **Function Declaration:** Uses `fun` keyword.

```kotlin
fun greet(name: String): String {
    return "Hello, $name!"
}
// Single-expression function
fun add(a: Int, b: Int) = a + b
```

- **Variable Declaration:** Uses `var` (mutable) and `val` (immutable, like `final` in Java).

```kotlin
val PI = 3.14159 // Constant
var counter = 0 // Variable
```

- **Classes:**

  ```
  class Person(val name: String, var age: Int) {
      fun sayHello() {
          println("Hello, my name is $name.")
      }
  }
  val person = Person("Alice", 30)
  person.sayHello()
  ```

- **Comments:** `//` for single-line, `/* ... */` for multi-line.

Coding Conventions

Kotlin has official coding conventions recommended by JetBrains and Google, promoting consistent and readable code:

- **Naming:** CamelCase for classes, functions, variables. PascalCase for class names. `SCREAMING_SNAKE_CASE` for top-level constants.
- **Formatting:** Indent 4 spaces, spaces around operators, etc.
- **Empty Lines:** Use judiciously for readability.
- **Ordering of Members:** Consistent ordering (properties, constructors, init blocks, functions).

Basic Types, Packages, Control Flow

- **Basic Types:** `Int`, `Double`, `Float`, `Long`, `Short`, `Byte`, `Boolean`, `Char`, `String`. All are objects.

- **Packages:** Declared with `package` keyword at the top of the file. Used to organize code.

  ```
  package com.example.myapp.utilities

  // ... code ...
  ```

- **Control Flow:**

  - `if`/`else if`/`else`: Can be used as an expression (returns a value).
  - `when`: Replaces `switch` statement, more powerful (can match types, ranges, conditions).
  - `for`: Iterates over ranges, collections, arrays.
  - `while`/`do-while`: Standard loops.
  - `break`/`continue`: Control loop execution.
  - `return`: Exit a function.

Java Interop (Calling Java code from Kotlin and vice versa)

Kotlin is designed for 100% interoperability with Java.

- **Calling Java from Kotlin:** You can directly use Java classes, methods, and fields in Kotlin code.

```
// Kotlin code
val arrayList = java.util.ArrayList<String>() // Using Java's ArrayList
arrayList.add("Hello")
val size = arrayList.size
```

- **Calling Kotlin from Java:** Kotlin code can be seamlessly called from Java.
  - Kotlin functions map to Java methods.
  - Kotlin properties map to Java getters/setters (and fields if `lateinit` or `const`).
  - Kotlin top-level functions and properties are compiled as static members of a class named `FileNameKt` (by default).

  ```
  // Java code
  // Assuming a Kotlin file named 'MyKotlinFile.kt' with a top-level function
  'greet'
  String message = MyKotlinFileKt.greet("World");
  System.out.println(message);
  ```

  - `@JvmStatic` can be used to make a Kotlin function a static method of a class.
  - `@JvmOverloads` can be used to generate overloads for functions with default parameter values.

Extension Functions

Allow you to add new functions to an existing class without modifying its source code.

- **Syntax:** `fun ClassName.functionName(params): ReturnType { ... }`
- **Use Cases:** Adding utility methods to standard library classes, making code more readable.

  ```
  fun String.removeSpaces(): String {
      return this.replace(" ", "")
  }

  val text = "Hello World"
  val noSpaces = text.removeSpaces() // Calls the extension function
  println(noSpaces) // "HelloWorld"
  ```

Coroutines

Kotlin Coroutines provide a way to write asynchronous, non-blocking code that is sequential and easy to reason about. They are lightweight threads that do not map directly to OS threads.

- **Purpose:** Simplify concurrent programming, especially for long-running operations like network requests or database access, keeping the UI responsive.
- **`suspend` keyword:** Marks a function that can be paused and resumed later. Can only be called from other `suspend` functions or a coroutine builder.
- **`launch` / `async`:** Coroutine builders to start new coroutines.

- **`withContext`:** To switch execution context (e.g., from a background thread to the main thread for UI updates).
- **Dispatchers:** Determine which thread(s) a coroutine runs on (e.g., `Dispatchers.Main`, `Dispatchers.IO`, `Dispatchers.Default`).

Example:

```
import kotlinx.coroutines.*

fun main() = runBlocking { // This creates a coroutine scope
    println("Start")

    launch(Dispatchers.IO) { // Start a coroutine on the IO thread
        delay(1000) // Simulate a network request
        println("Data fetched")
        withContext(Dispatchers.Main) { // Switch to main thread for UI update (in
Android)
            println("UI updated with data")
        }
    }

    println("End (main thread continues)")
}
// Output:
// Start
// End (main thread continues)
// Data fetched
// UI updated with data
```

Null Safety, Smart Casts, Properties

- **Null Safety:**
  - **Nullable Types:** Declared with `?` (e.g., `String?`, `Int?`). You must explicitly handle `nil` values (e.g., with `if (variable != null)`, `?.` safe call, `?:` Elvis operator).
  - **Non-nullable Types:** Declared without `?` (e.g., `String`, `Int`). Compiler ensures they are never `null`.
  - **`!!` (Null Assertion Operator):** Forces a nullable type to be treated as non-nullable. Crashes if the value is `null`. Use with extreme caution.
  - **`lateinit`:** Used for non-nullable variables that will be initialized later, typically in `onCreate` for Android components. Avoids `NullPointerException` if accessed before initialization.
- **Smart Casts:** Kotlin's compiler automatically casts a variable to a specific type within a block of code after a type check (`is` operator) or a null check (`!= null`).

```
fun printLength(obj: Any) {
    if (obj is String) { // obj is smart-cast to String here
        println("String length: ${obj.length}")
    }
}
```

- **Properties:** Kotlin distinguishes between mutable (`var`) and immutable (`val`) properties. They are accessed directly, but compiler generates getters/setters behind the scenes.
    - **Custom Getters/Setters:** You can define custom logic for property access.
    - **Backing Fields:** Implicit backing fields are used unless custom accessors entirely replace the default behavior.

First Class Delegation, Singletons

- **First Class Delegation (`by` keyword):** Allows a class to delegate the implementation of an interface to another object. This is a powerful feature for implementing patterns like Decorator or Proxy without boilerplate.

```kotlin
interface MyLogger {
    fun log(message: String)
}

class ConsoleLogger : MyLogger {
    override fun log(message: String) {
        println("Console: $message")
    }
}

class AppLogger(private val logger: MyLogger) : MyLogger by logger {
    // AppLogger delegates log() calls to ConsoleLogger
    fun logWithPrefix(message: String) {
        log("[APP] $message") // Calls the delegated log method
    }
}

val appLogger = AppLogger(ConsoleLogger())
appLogger.logWithPrefix("Hello from app!")
```

- **Singletons (`object` keyword):** Kotlin makes it easy to declare a singleton using the `object` keyword. This ensures only one instance of the class ever exists.

```kotlin
object AppSettings {
    var theme: String = "dark"
    fun saveSetting(setting: String) {
        println("Saving setting: $setting")
    }
}

// Access the singleton directly
AppSettings.theme = "light"
AppSettings.saveSetting("profile_updated")
```

Companion Objects

- **Purpose:** A companion object is a singleton object defined within a class. It can contain functions and properties that belong to the class itself, rather than to instances of the class (similar to static members in Java).
- **Access:** Members of a companion object are accessed directly via the class name.
- **Use Cases:** Factory methods, constants, utility methods related to the class.

```
class MyClass {
    companion object {
        const val TAG = "MyClass" // Constant
        fun createInstance(): MyClass { // Factory method
            return MyClass()
        }
    }
}

println(MyClass.TAG)
val instance = MyClass.createInstance()
```

Data Classes

- **Purpose:** Designed to hold data. The compiler automatically generates `equals()`, `hashCode()`, `toString()`, `copy()`, and `componentN()` functions for properties declared in the primary constructor.
- **Syntax:** `data class ClassName(...)`
- **Use Cases:** POJOs, DTOs.

```
data class User(val id: Int, val name: String, var email: String)

val user1 = User(1, "Alice", "alice@example.com")
val user2 = User(1, "Alice", "alice@example.com")

println(user1 == user2) // true (equals() generated)
println(user1.toString()) // toString() generated

val user3 = user1.copy(email = "alice.new@example.com") // copy() generated
println(user3)
```

Understanding the Android Architecture

The Android operating system is structured as a software stack, typically divided into four main layers, building on top of the Linux kernel.

Android Layered Architecture

1. **Linux Kernel Layer:**

    - The base of the Android stack.
    - Provides core system services like security, memory management, process management, network stack, and driver model.
    - Handles hardware-specific interactions (e.g., camera, Wi-Fi, audio drivers, power management).
    - Android uses a modified Linux kernel, not a standard Linux distribution.

2. **Hardware Abstraction Layer (HAL):** (Not always listed as a distinct layer, but conceptually important)

    - An interface layer between the hardware-specific device drivers (in the Linux Kernel) and the higher-level Java API framework.
    - Allows Android to be largely hardware-agnostic by providing a consistent interface for hardware components. Implementations are device-specific.

3. **Android Runtime (ART) and Native Libraries:**

    - **Android Runtime (ART):**
        - The virtual machine that executes Android applications. (Replaced Dalvik in Android 5.0 Lollipop).
        - Performs **AOT (Ahead-Of-Time) compilation** during app installation, converting Dalvik bytecode (DEX) into machine code for faster app launch and execution.
        - Also supports **JIT (Just-In-Time) compilation** for dynamic optimization.
        - Manages memory (garbage collection).
    - **Core Libraries:** Java API frameworks (e.g., `java.util`, `java.io`, `java.net`) that are accessible to developers.
    - **Native C/C++ Libraries:** Pre-built native libraries written in C/C++ that provide services to the Android platform and applications. These include:
        - `SurfaceManager`: For display rendering.
        - `Media Framework`: For audio/video playback and recording.
        - `SQLite`: Database support.
        - `OpenGL ES`: For 3D graphics.
        - `WebKit`: Browser engine (for `WebView`).
        - `libc`: Standard C library.

4. **Application Framework Layer:**

    - Provides the high-level building blocks that Android developers use to create apps.
    - Written in Java (and now Kotlin).
    - Key services and managers available via APIs:
        - `Activity Manager`: Manages the lifecycle of applications' activities.
        - `Package Manager`: Manages installed applications and packages.
        - `Window Manager`: Manages windows and drawing to the screen.
        - `Content Providers`: Allows applications to share data.
        - `Resource Manager`: Provides access to non-code resources (strings, layouts, drawables).
        - `Location Manager`: For location-based services.
        - `Notification Manager`: For creating and managing notifications.
        - `Telephony Manager`: For device telephony services.

5. **Applications Layer:**

  - This is the top layer, comprising all the applications that run on Android.
  - Includes both pre-installed system apps (e.g., Home, Contacts, Phone, Browser) and third-party apps downloaded from the Play Store.
  - Apps interact with the underlying framework layers through the Java/Kotlin APIs provided by the application framework.

Intro to IPC in Android

IPC (Inter-Process Communication): The mechanism that allows different processes (e.g., different applications, or different components within the same application if they run in separate processes) to communicate and share data. Android heavily relies on IPC because each application runs in its own Linux process, isolated from others.

IPC using Binder Interface

- **Binder:** The primary IPC mechanism in Android. It's a high-performance, lightweight RPC (Remote Procedure Call) mechanism that allows processes to call methods on objects that reside in other processes.
- **How it Works:**
  - **Service Provider:** A component (e.g., a Service) in one process implements a `Binder` interface.
  - **Service Consumer:** A component in another process gets a proxy object that implements the same interface.
  - When the consumer calls a method on the proxy, the Binder driver handles the inter-process communication, marshaling (packaging) the arguments, transmitting them across processes, unmarshaling them, executing the method on the real object in the service provider's process, and then marshaling and returning the results.
- **AIDL (Android Interface Definition Language):** Used to define the interface for Binder IPC when processes need to communicate across process boundaries (e.g., for background services used by multiple apps).

Dalvik VM

- **Dalvik Virtual Machine:** Was the process virtual machine in Android versions up to 4.4 KitKat. It executed applications written in Java (compiled to Dalvik Executable - `.dex` bytecode).
- **Just-In-Time (JIT) Compilation:** Dalvik used JIT compilation, meaning code was compiled to machine code just before it was executed.
- **Replaced by ART:** With Android 5.0 Lollipop, Dalvik was entirely replaced by ART (Android Runtime), which uses Ahead-Of-Time (AOT) compilation for better performance and battery life.

Intro to Fundamental Components of Android OS

Android applications are built as a collection of modular components, each serving a distinct purpose.

1. **Activity:**

   - **Purpose:** Represents a single screen with a user interface. It's the entry point for user interaction with your app.
   - **User Interface:** An Activity typically has an associated layout XML file that defines its UI.

   - **Lifecycle:** Activities have a well-defined lifecycle (created, started, resumed, paused, stopped, destroyed).
   - **Example:** A login screen, a list of items, a settings screen.

2. **Service:**

   - **Purpose:** A component that performs long-running operations in the background, without a user interface.
   - **No UI:** Services do not provide a UI.
   - **Use Cases:** Playing music in the background, downloading files, performing network operations.
   - **Types:**
     - **Started Service:** Runs indefinitely until explicitly stopped or it completes its work.
     - **Bound Service:** Allows other application components to bind to it and interact with it (IPC).

3. **Content Provider:**

   - **Purpose:** Manages access to a structured set of data. It provides a standard interface for applications to query, insert, update, and delete data.
   - **Data Sharing:** Enables secure data sharing between different applications (e.g., Contacts, Calendar).
   - **Abstracts Data Source:** Can store data in various ways (SQLite database, file system, network).
   - **Example:** Accessing a user's contacts or photos.

4. **Broadcast Receiver:**

   - **Purpose:** A component that allows the app to respond to system-wide broadcast announcements or custom broadcasts from other apps or your own app.
   - **Event-Driven:** They are event-driven components. When an event (e.g., battery low, SMS received, device booted) occurs, the system sends a broadcast, and receivers registered for that event are invoked.
   - **No UI:** Typically do not have a UI.
   - **Use Cases:** Receiving notifications about connectivity changes, app updates, charging status.

Activity

Activities are fundamental building blocks of Android applications, representing a single, focused thing the user can do.

Introduction to Activity

- **Definition:** An Activity is a component that provides a screen with which users can interact to do something (e.g., dial a phone number, view an email, take a photo). Each Activity is essentially a single screen in your app.
- **User Interface:** An Activity is typically associated with a layout file (XML) that defines its UI.
- **Stack:** Activities are managed as a "back stack." When a new activity starts, it's pushed onto the top of the stack. When the user presses the Back button, the current activity is popped from the stack, and the previous one resumes.

Activity Life Cycle

An Activity goes through various states from its creation to its destruction. The Android system manages this lifecycle through a series of callback methods.

- `onCreate()`:
  - **When called:** The system creates the activity.
  - **Purpose:** Perform basic application startup logic that should happen only once for the entire life of the activity (e.g., inflate layout, initialize views, bind data to lists).
  - **Mandatory:** Every activity must implement this.
- `onStart()`:
  - **When called:** The activity becomes visible to the user.
  - **Purpose:** Initialize UI elements or code that needs to be active when the user sees the activity.
- `onResume()`:
  - **When called:** The activity enters the "resumed" state; it is visible and in the foreground, and the user can interact with it.
  - **Purpose:** Start animations, acquire exclusive device capabilities (e.g., camera access), register broadcast receivers.
- `onPause()`:
  - **When called:** The activity is about to enter the "paused" state (e.g., another activity comes to the foreground, dialog appears).
  - **Purpose:** Release resources that are no longer needed while the activity is in the background, commit unsaved changes. Keep operations brief.
- `onStop()`:
  - **When called:** The activity is no longer visible to the user (e.g., another activity completely covers it, user navigates to another app, app is moving to background).
  - **Purpose:** Perform more extensive shutdown operations (e.g., save larger amounts of data, unregister receivers, stop animations).
- `onDestroy()`:
  - **When called:** The activity is about to be destroyed and removed from memory.
  - **Purpose:** Perform final cleanup (e.g., release all resources, stop background threads).
- `onRestart()`:
  - **When called:** The activity is stopped but is being re-displayed to the user (e.g., user navigates back to it from another activity). It's always followed by `onStart()` and `onResume()`.

Various states and lifetimes of an Activity

- **Running (Resumed):** The activity is in the foreground, visible to the user, and has focus.
- **Paused:** The activity is partially obscured by another activity (e.g., a transparent activity or a dialog). It is still alive (retains state and member information) but no longer has user focus.
- **Stopped:** The activity is completely hidden by another activity or moved to the background. It is still alive but might be killed by the system if memory is low.
- **Destroyed:** The activity is removed from memory.

Lifecycle Transitions:

- `onCreate` -> `onStart` -> `onResume`: Full activity startup.
- `onPause` -> `onResume`: Activity partially obscured, then resumed.

- `onPause` **->** `onStop` **->** `onRestart` **->** `onStart` **->** `onResume`: Activity moved to background, then brought back to foreground.
- `onPause` **->** `onStop` **->** `onDestroy`: Activity destroyed.

Starting an Activity

You start an Activity using an `Intent`.

```
// From an Activity
val intent = Intent(this, SecondActivity::class.java) // Explicit Intent
startActivity(intent)
```

Starting an Activity for result

Sometimes you need to start an activity and get a result back from it when it finishes (e.g., selecting a contact, taking a photo).

- `startActivityForResult()` **(Deprecated):** This method was used in older API levels.
- `ActivityResultLauncher` **(Recommended for modern Android):** Part of the Activity Result API, providing a safer and more convenient way to get results.

Example with `ActivityResultLauncher` (Kotlin):

```
class MainActivity : AppCompatActivity() {

    private lateinit var resultTextView: TextView

    // 1. Declare the launcher
    private val startForResult =
registerForActivityResult(ActivityResultContracts.StartActivityForResult()) {
result ->
        if (result.resultCode == Activity.RESULT_OK) {
            val data: Intent? = result.data
            val receivedData = data?.getStringExtra("returnData")
            resultTextView.text = "Result: $receivedData"
        } else {
            resultTextView.text = "Activity cancelled or failed."
        }
    }

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

        resultTextView = findViewById(R.id.resultTextView)
        val startButton: Button = findViewById(R.id.startButton)

        startButton.setOnClickListener {
            val intent = Intent(this, SecondActivity::class.java)
            intent.putExtra("initialData", "Hello from Main!")
            startForResult.launch(intent) // 2. Launch the activity
        }
    }
}

class SecondActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_second)

        val receivedData = intent.getStringExtra("initialData")
        val displayTextView: TextView = findViewById(R.id.displayTextView)
        displayTextView.text = "Received: $receivedData"

        val finishButton: Button = findViewById(R.id.finishButton)
        finishButton.setOnClickListener {
            val returnIntent = Intent()
            returnIntent.putExtra("returnData", "Result from Second!")
            setResult(Activity.RESULT_OK, returnIntent) // Set the result
            finish() // Finish this activity
        }
    }
}
```

AppCompat library

- **Purpose:** A support library (part of AndroidX) that provides backward compatibility for new Android features and Material Design to older Android versions.
- **Usage:** You'll typically extend `AppCompatActivity` instead of `Activity` for your activities to leverage these compatibility features.
- **Benefits:** Ensures your app looks and behaves consistently across a wide range of Android devices, even those running older OS versions.

Intent

- **Overview of Intent:** An `Intent` is a messaging object that allows you to request an action from another app component (Activity, Service, Broadcast Receiver). It's a key mechanism for inter-component communication in Android.
- **Various uses of Intent:**
  - **Starting an Activity:** Launching a new screen.
  - **Starting a Service:** Initiating a background operation.
  - **Delivering a Broadcast:** Notifying interested components about an event.
  - **Passing Data:** Bundling data to be passed between components.

Intent: An IPC mechanism

As an IPC (Inter-Process Communication) mechanism, Intents allow your application components to interact with components in other applications, even if they run in separate processes. The Android system routes Intents to the appropriate component.

Types of Intent

1. **Explicit Intent:**
    - **Purpose:** Used to launch a specific component within your own application. You explicitly name the target component (Activity, Service, or Broadcast Receiver).
    - **Use Cases:** Navigating from one screen to another within the same app.
    - **Example:**

    ```
    val intent = Intent(this, DetailActivity::class.java) // Explicitly names DetailActivity
    startActivity(intent)
    ```

2. **Implicit Intent:**
    - **Purpose:** Used to request an action, allowing the Android system to find and launch the appropriate component that can handle that action, possibly from another application. You don't name the specific component.
    - **Use Cases:** Opening a web page, dialing a phone number, sending an email, taking a photo.
    - **Example:**

    ```
    // Open a web page
    val webpage = Uri.parse("http://www.google.com")
    val intent = Intent(Intent.ACTION_VIEW, webpage) // Action: VIEW, Data: a URL
    startActivity(intent)

    // Send an email
    val emailIntent = Intent(Intent.ACTION_SENDTO).apply {
        data = Uri.parse("mailto:example@example.com") // Only email apps should handle this
        putExtra(Intent.EXTRA_SUBJECT, "Subject")
        putExtra(Intent.EXTRA_TEXT, "Message body")
    }
    startActivity(emailIntent)
    ```

Intent Filter

An `Intent Filter` is an XML element defined in the `AndroidManifest.xml` file that tells the Android system which implicit Intents an application component (Activity, Service, Broadcast Receiver) can respond to.

- **`action`:** Declares the action the component can perform (e.g., `android.intent.action.VIEW`, `android.intent.action.SEND`).
- **`data`:** Declares the type of data (MIME type) and/or URI scheme that the component can handle (e.g., `http`, `tel`, `image/*`).

- `category`: Provides additional context about the component's capabilities (e.g., `android.intent.category.LAUNCHER` for app's main entry point, `android.intent.category.BROWSABLE` for web links).

Example `AndroidManifest.xml` snippet for an implicit intent:

```
<activity android:name=".MyWebViewerActivity" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" /> <!-- Default
category for implicit intents -->
        <category android:name="android.intent.category.BROWSABLE" /> <!--
Indicates it can be invoked by a browser -->
        <data android:scheme="http" android:host="www.example.com" /> <!-- Handles
specific host -->
        <data android:scheme="https" /> <!-- Handles any https URL -->
        <data android:mimeType="text/plain" /> <!-- Also handles text/plain data -
->
    </intent-filter>
</activity>
```

Context

- **Definition:** `Context` is an abstract class that provides access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting intents, and accessing preferences. It's like the "handle" to the operating system's services.
- **Various uses of Context:**
  - **Accessing Resources:** `context.getString(R.string.app_name)`, `context.getColor(R.color.my_color)`.
  - **Launching Activities/Services:** `context.startActivity(intent)`, `context.startService(intent)`.
  - **Inflating Layouts:** `LayoutInflater.from(context).inflate(...)`.
  - **Getting System Services:** `context.getSystemService(Context.CONNECTIVITY_SERVICE)`.
  - **Accessing Databases/Preferences:** `context.getSharedPreferences()`, `context.openOrCreateDatabase()`.
  - **Showing Toasts/Dialogs:** `Toast.makeText(context, ...)`, `AlertDialog.Builder(context, ...)`.
- **Types of Context:**
  - **`Application Context`:** A singleton instance tied to the lifecycle of the entire application. It exists as long as the application process exists. Use for global, long-lived operations.
  - **`Activity Context`:** An instance tied to the lifecycle of a specific Activity. It exists only as long as the Activity is alive. Use for UI-related operations specific to that Activity.
  - **`Service Context`:** Similar to Activity context, tied to the lifecycle of a Service.

Data sharing using Intent

Intents can carry data using `Bundle` objects.

- **Sharing primitive data using Intents:**
  - Use `putExtra()` methods to add primitive data types (String, Int, Boolean, etc.).
  - Use `getStringExtra()`, `getIntExtra()`, etc., on the receiving end.

  ```
  // Sending
  val intent = Intent(this, SecondActivity::class.java)
  intent.putExtra("username", "JohnDoe")
  intent.putExtra("userAge", 30)
  startActivity(intent)

  // Receiving in SecondActivity
  val username = intent.getStringExtra("username")
  val age = intent.getIntExtra("userAge", 0) // 0 is default value
  ```

- **Sharing objects using Intents:**
  - For custom objects, they must implement `Parcelable` (preferred for performance on Android) or `Serializable`.
  - Use `putExtra("key", myObject as Parcelable)` or `putExtra("key", myObject as Serializable)`.
  - Use `getParcelableExtra("key")` or `getSerializableExtra("key")` on the receiving end.

  Example (with Parcelable):

  ```
  // 1. Define your data class as Parcelable
  @Parcelize
  data class User(val id: Int, val name: String) : Parcelable

  // Sending
  val user = User(1, "Alice")
  val intent = Intent(this, DetailActivity::class.java)
  intent.putExtra("user_data", user)
  startActivity(intent)

  // Receiving in DetailActivity
  val userReceived: User? = intent.getParcelableExtra("user_data")
  // For older APIs: val userReceived: User? =
  intent.getParcelableExtra("user_data", User::class.java)
  ```

Navigation

Navigation in Android refers to how users move between different screens (Activities, Fragments) within your app.

- **Back Stack:** As users navigate, Activities are placed on a back stack. Pressing the Back button pops the current Activity, revealing the previous one.

- **Navigation Component:** (Recommended by Google for modern Android development) A framework that simplifies navigation within your app.
    - **Navigation Graph:** An XML file that visually represents all your app's navigation paths (destinations and actions).
    - **`NavController`:** Manages navigation within a `NavHost` (typically a `NavHostFragment`).
    - **Actions:** Define how users can navigate between destinations.
    - **Arguments:** Pass data between destinations in a type-safe way using `Safe Args` Gradle plugin.
    - **Deep Links:** Handle incoming links that directly open a specific destination in your app.

Basic Android UI

The Android UI is built using a hierarchy of `View` and `ViewGroup` objects.

View, View Group and Layout

- **`View`:**
    - **Definition:** The basic building block for user interface components. A `View` occupies a rectangular area on the screen and is responsible for drawing itself and handling events.
    - **Examples:** `TextView`, `Button`, `ImageView`, `EditText`.
- **`ViewGroup`:**
    - **Definition:** A special type of `View` that can contain (and arrange) other `View`s and `ViewGroup`s. They are the containers for UI elements.
    - **Examples:** `LinearLayout`, `RelativeLayout`, `ConstraintLayout`, `FrameLayout`, `ScrollView`.
- **Layout:**
    - **Definition:** The XML file (`.xml`) that defines the structure of your UI (which Views are present and how they are arranged). They are inflated by the Android system at runtime to create the actual View objects.

Android View System

The Android View system is a hierarchical structure. The root of the hierarchy is a `ViewGroup` (often a `ConstraintLayout` or `LinearLayout`) that contains child `View`s and other `ViewGroup`s, forming a tree.

- **Inflation:** Layout XML files are "inflated" at runtime, meaning the XML tags are converted into their corresponding Java/Kotlin `View` objects.
- **Drawing:** The Android system uses the hierarchy to draw the UI from top to bottom, passing drawing instructions to the GPU.
- **Event Handling:** Events (like touch events) are dispatched from the root view down to the appropriate child view that handles them.

Android Layouts

Layouts are `ViewGroup` subclasses that arrange their child views in specific ways.

- **`LinearLayout`:**

    - **Purpose:** Arranges child views in a single row or single column.
    - **Properties:**

        - `android:orientation`: `horizontal` or `vertical`.
        - `android:layout_weight`: Distributes available space among children.
    - **Use Cases:** Simple vertical or horizontal lists, forms.

- **`RelativeLayout`:**
    - **Purpose:** Arranges child views based on relationships between themselves or relative to the parent layout.
    - **Properties:** `android:layout_alignParentTop`, `android:layout_centerInParent`, `android:layout_below`, `android:layout_toRightOf`, etc.
    - **Use Cases:** Complex layouts where elements are positioned relative to each other. Can lead to nested view groups, which might impact performance.

- **`FrameLayout`:**
    - **Purpose:** The simplest `ViewGroup`. It arranges all its child views in a stack, with the most recently added child appearing on top.
    - **Use Cases:** Overlapping views (e.g., an `ImageView` with a `TextView` overlay), fragments.

- **`ConstraintLayout`:**
    - **Purpose:** A flexible layout system that allows you to position and size views using constraints, similar to AutoLayout in iOS. It's Google's recommended layout for building complex and flat view hierarchies.
    - **Properties:** Uses constraints to define relationships (e.g., `app:layout_constraintStart_toStartOf`, `app:layout_constraintTop_toBottomOf`).
    - **Benefits:** Reduces nested view groups, improves performance, provides a powerful visual editor in Android Studio.
    - **Use Cases:** Almost all modern complex layouts.

Button, Text View and Edit Text

These are common UI widgets for user interaction and display.

- **`Button`:**
    - **Purpose:** A clickable UI element that performs an action when tapped.
    - **XML:** `<Button android:text="Click Me" ... />`
    - **Event Listener (Kotlin):**

    ```kotlin
    val myButton: Button = findViewById(R.id.myButton)
    myButton.setOnClickListener {
        // Handle button click
    }
    ```

- **`TextView`:**
    - **Purpose:** Displays read-only text.
    - **XML:** `<TextView android:text="Hello" ... />`
    - **Kotlin:** `myTextView.text = "New Text"`

- `EditText`:
  - **Purpose:** Allows users to input and modify text.
  - **XML:** `<EditText android:hint="Enter text" android:inputType="textPersonName" ... />`
  - **Properties:** `android:hint` (placeholder), `android:inputType` (keyboard type), `android:maxLines`.
  - **Kotlin:**

    ```
    val myEditText: EditText = findViewById(R.id.myEditText)
    val enteredText = myEditText.text.toString() // Get text
    myEditText.setText("Default value") // Set text
    ```

Various UI Events and Event listeners

User interactions with the UI generate events. Event listeners are interfaces you implement to respond to these events.

- `OnClickListener`: For taps on buttons, images, or any clickable view.

  ```
  myButton.setOnClickListener { view ->
      // Handle click
  }
  ```

- `OnLongClickListener`: For long presses.
- `OnTouchListener`: For raw touch events (down, move, up).
- `OnKeyListener`: For keyboard key presses.
- `OnFocusChangeListener`: When a view gains or loses focus.
- `TextWatcher`: For listening to text changes in an `EditText` (before, on, after text changed).

Scroll View (`ScrollView`)

- **Purpose:** A `ViewGroup` that allows its content to be scrolled. It can contain only one direct child `View` or `ViewGroup`.
- **Use Cases:** When content is larger than the screen size (e.g., long forms, articles).
- **Vertical vs. Horizontal:** Default is vertical. For horizontal, use `HorizontalScrollView`.
- **XML:**

  ```
  <ScrollView
      android:layout_width="match_parent"
      android:layout_height="match_parent">
      <!-- Must contain only ONE direct child -->
      <LinearLayout
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:orientation="vertical">
          <!-- Your scrollable content here -->
  ```

```
    </LinearLayout>
</ScrollView>
```

Spinner

- **Purpose:** A dropdown list that allows users to select one item from a list.
- **XML:** `<Spinner ... />`
- **Data Source:** Typically populated using an `Adapter` (e.g., `ArrayAdapter` for simple string lists).
- **Event Listener:** `AdapterView.OnItemSelectedListener` to detect when an item is selected.

Example (Kotlin):

```
val spinner: Spinner = findViewById(R.id.spinner)
val languages = arrayOf("Kotlin", "Java", "Python", "Swift")
val adapter = ArrayAdapter(this, android.R.layout.simple_spinner_item, languages)
// Use built-in spinner item layout
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item) //
Set dropdown layout
spinner.adapter = adapter

spinner.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
    override fun onItemSelected(parent: AdapterView<*>?, view: View?, position:
Int, id: Long) {
        val selectedItem = parent?.getItemAtPosition(position).toString()
        Toast.makeText(this@MainActivity, "Selected: $selectedItem",
Toast.LENGTH_SHORT).show()
    }
    override fun onNothingSelected(parent: AdapterView<*>?) {
        // Do nothing
    }
}
```

Checkbox

- **Purpose:** A two-state button (checked or unchecked) that allows users to select one or more options from a set.
- **XML:** `<CheckBox ... />`
- **State:** `isChecked()` (boolean).
- **Event Listener:** `CompoundButton.OnCheckedChangeListener` to detect state changes.

Example (Kotlin):

```
val checkbox: CheckBox = findViewById(R.id.checkbox)
checkbox.setOnCheckedChangeListener { buttonView, isChecked ->
    if (isChecked) {
        Toast.makeText(this, "Checkbox checked", Toast.LENGTH_SHORT).show()
    } else {
        Toast.makeText(this, "Checkbox unchecked", Toast.LENGTH_SHORT).show()
    }
}
```

Radio Button (`RadioButton`, `RadioGroup`)

- **Purpose:** `RadioButton`s are used in conjunction with a `RadioGroup` to allow users to select only one option from a mutually exclusive set.
- **`RadioGroup`:** A `ViewGroup` that contains `RadioButton`s, ensuring that only one radio button within the group can be selected at a time.
- **XML:**

  ```
  <RadioGroup android:id="@+id/radioGroup" ...>
      <RadioButton android:id="@+id/radioMale" android:text="Male" ... />
      <RadioButton android:id="@+id/radioFemale" android:text="Female" ... />
  </RadioGroup>
  ```

- **Event Listener:** `RadioGroup.OnCheckedChangeListener`.

Example (Kotlin):

```
val radioGroup: RadioGroup = findViewById(R.id.radioGroup)
radioGroup.setOnCheckedChangeListener { group, checkedId ->
    val selectedRadioButton: RadioButton = findViewById(checkedId)
    Toast.makeText(this, "Selected: ${selectedRadioButton.text}",
Toast.LENGTH_SHORT).show()
}
```

ImageView

- **Purpose:** Displays an image.
- **XML:** `<ImageView android:src="@drawable/my_image" ... />`
- **Properties:**
  - `android:src`: Sets the image source (drawable, mipmap).
  - `android:scaleType`: Defines how the image is scaled or positioned within the `ImageView`'s bounds (e.g., `centerCrop`, `fitCenter`, `centerInside`).
- **Kotlin:** `myImageView.setImageResource(R.drawable.another_image)` or `myImageView.setImageBitmap(bitmap)`.

Toast

- **Purpose:** A small pop-up message that appears temporarily at the bottom of the screen and then fades away. Used for brief feedback to the user.
- **Non-interactive:** Users cannot interact with a Toast.
- **Example (Kotlin):**

```
Toast.makeText(this, "Hello from Toast!", Toast.LENGTH_SHORT).show()
```

SnackBar

- **Purpose:** A lightweight feedback mechanism that displays a brief message at the bottom of the screen, similar to a Toast, but can also include an optional action button.
- **Interactive:** Can include an action button for user interaction.
- **Dismissible:** Can be dismissed by swiping.
- **Requires Material Design library.**

Example (Kotlin):

```
// Requires a View (e.g., root layout, a button) to "anchor" the SnackBar
val rootView: View = findViewById(android.R.id.content) // Get content view of the
activity

Snackbar.make(rootView, "Item deleted.", Snackbar.LENGTH_LONG)
    .setAction("UNDO") {
        // Handle undo action
        Toast.makeText(this, "Undo clicked!", Toast.LENGTH_SHORT).show()
    }
    .show()
```

Animations

Android provides various animation APIs to create engaging and dynamic user interfaces.

- **Property Animation:** (Recommended for modern Android)
    - **Purpose:** Animates actual properties of `View` objects over time (e.g., `alpha`, `scaleX`, `translationY`).
    - **Classes:** `ObjectAnimator`, `ValueAnimator`.
    - **XML or Code:** Can be defined in XML (`res/animator/`) or programmatically.
    - **Interpolators:** Define the rate of change of an animation (e.g., `AccelerateInterpolator`, `DecelerateInterpolator`).
    - **Use Cases:** Complex, flexible animations for any view property.
    - **Example (Kotlin, simple alpha animation):**

    ```
    val myView: View = findViewById(R.id.myView)
    val alphaAnimator = ObjectAnimator.ofFloat(myView, "alpha", 0f, 1f)
    alphaAnimator.duration = 1000 // milliseconds
    alphaAnimator.start()
    ```

- **View Animation (Tween Animation):**
    - **Purpose:** Simple animations that operate on a `View` object (e.g., rotation, scale, translate, alpha).

  - **XML:** Defined in XML (`res/anim/`).
  - **Limitations:** Only transforms the drawing of the View, not its actual properties (e.g., clicking on a button after it moved via animation still requires clicking its original location).
  - **Use Cases:** Simple transitions, fades.
- **Drawable Animation:**
  - **Purpose:** Animates a sequence of Drawable resources (like a GIF).
  - **XML:** Defined in XML (`res/drawable/`).
  - **Use Cases:** Loading indicators, simple character animations.
- **Transition Framework:**
  - **Purpose:** Animates changes in layouts between different states (scenes).
  - **Use Cases:** Automatically animate changes when views are added/removed or properties change within a `ViewGroup`.

Notifications

Notifications are messages that Android displays outside your app's UI to keep the user informed.

- **`NotificationCompat.Builder` (AndroidX):** The primary class for building notifications, providing backward compatibility.
- **Channels (Android 8.0+):** All notifications must belong to a channel. Users can control settings for each channel.
- **Importance/Priority:** Controls how interruptive a notification is (e.g., `IMPORTANCE_HIGH` for heads-up, sound, vibration).
- **Pending Intent:** An Intent that is passed to another application and executed when that app is triggered (e.g., when the user taps the notification).

Basic Notification

```
val channelId = "my_channel_id"
val channelName = "My Notifications"
val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager

// Create Notification Channel (required for Android 8.0+)
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    val channel = NotificationChannel(channelId, channelName,
NotificationManager.IMPORTANCE_DEFAULT).apply {
        description = "Description for My Notifications"
    }
    notificationManager.createNotificationChannel(channel)
}

// Create a Pending Intent to open MainActivity when notification is tapped
val intent = Intent(this, MainActivity::class.java).apply {
    flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
}
val pendingIntent: PendingIntent = PendingIntent.getActivity(this, 0, intent,
PendingIntent.FLAG_IMMUTABLE)
```

```
// Build the notification
val builder = NotificationCompat.Builder(this, channelId)
    .setSmallIcon(R.drawable.ic_notification)
    .setContentTitle("My Notification Title")
    .setContentText("This is the detailed content of my notification.")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    .setContentIntent(pendingIntent) // Set the intent to be launched when tapped
    .setAutoCancel(true) // Automatically dismisses the notification when tapped

// Show the notification
notificationManager.notify(1, builder.build()) // 1 is a unique ID for this
notification
```

Expanded Notification

Notifications can display more content in an expanded state.

- **Big Text Style (`NotificationCompat.BigTextStyle`):** For long text messages.
- **Big Picture Style (`NotificationCompat.BigPictureStyle`):** For large images.
- **Inbox Style (`NotificationCompat.InboxStyle`):** For displaying a list of short messages.

Example (Big Text Style):

```
// ... (same initial setup as basic notification) ...

val longText = "This is a very long text that demonstrates the Big Text Style in
notifications. It can span multiple lines to provide more information to the user
without having to open the app."

val builder = NotificationCompat.Builder(this, channelId)
    .setSmallIcon(R.drawable.ic_notification)
    .setContentTitle("Expanded Notification")
    .setContentText("Tap to see more...")
    .setStyle(NotificationCompat.BigTextStyle().bigText(longText)) // Apply Big
Text Style
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    .setContentIntent(pendingIntent)
    .setAutoCancel(true)

notificationManager.notify(2, builder.build())
```

Progress Dialog (`ProgressDialog` - Deprecated, `ProgressBar` - Recommended)

- **`ProgressDialog`:** (DEPRECATED - Do not use in modern Android). Used to show indeterminate or determinate progress. It's a blocking dialog and leads to bad UX.
- **`ProgressBar`:** (RECOMMENDED) A UI element to show progress directly within your layout.
    - **Indeterminate:** Spinning wheel (`android:indeterminate="true"`) for ongoing operations.

    - **Determinate:** Horizontal bar (`android:style="?android:attr/progressBarStyleHorizontal"`) for tasks with measurable progress.

Example (XML):

```xml
<ProgressBar
    android:id="@+id/progressBar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:indeterminate="true"
    android:visibility="gone" /> <!-- Initially hidden -->

<ProgressBar
    android:id="@+id/progressBarHorizontal"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:max="100"
    android:progress="0"
    android:visibility="gone" />
```

Example (Kotlin):

```kotlin
val progressBar: ProgressBar = findViewById(R.id.progressBar)
progressBar.visibility = View.VISIBLE // Show spinner
// ... perform background task ...
progressBar.visibility = View.GONE // Hide spinner

val progressBarHorizontal: ProgressBar = findViewById(R.id.progressBarHorizontal)
progressBarHorizontal.visibility = View.VISIBLE
progressBarHorizontal.progress = 50 // Update progress
```

Media Player

Android provides APIs for playing audio and video.

Playing Audio (`MediaPlayer`)

- **`MediaPlayer`:** A versatile class for playing audio and video files.
- **States:** Has a complex state machine (Idle, Initialized, Prepared, Started, Paused, PlaybackCompleted, Stopped, End, Error).
- **Sources:** Can play from local files, assets, content URIs, or streaming URLs.

Example (Kotlin, playing audio from raw resource):

```kotlin
private var mediaPlayer: MediaPlayer? = null
```

```
    fun playAudio() {
        mediaPlayer = MediaPlayer.create(this, R.raw.my_audio_file) // my_audio_file
    in res/raw
        mediaPlayer?.setOnCompletionListener {
            Toast.makeText(this, "Audio playback complete!",
    Toast.LENGTH_SHORT).show()
            releaseMediaPlayer() // Release resources after playback
        }
        mediaPlayer?.start() // Start playback
    }

    fun pauseAudio() {
        mediaPlayer?.pause()
    }

    fun stopAudio() {
        mediaPlayer?.stop()
        releaseMediaPlayer()
    }

    private fun releaseMediaPlayer() {
        mediaPlayer?.release() // Release native resources
        mediaPlayer = null
    }

    override fun onDestroy() {
        super.onDestroy()
        releaseMediaPlayer() // Ensure resources are released on Activity destruction
    }
```

Playing Video (`VideoView`, `MediaPlayer`, `ExoPlayer`)

- **`VideoView`:** A simple `View` for playing videos. Best for basic playback.
- **`MediaPlayer`:** Can also be used for video, often combined with a `SurfaceView` or `TextureView` to render video frames.
- **`ExoPlayer`:** (RECOMMENDED for modern Android) An open-source media player library developed by Google. Provides more advanced features, customization, and better handling of adaptive streaming formats (DASH, HLS).

Example (`VideoView` in XML):

```
<VideoView
    android:id="@+id/videoView"
    android:layout_width="match_parent"
    android:layout_height="300dp"
    android:layout_centerInParent="true" />
```

Example (Kotlin, `VideoView`):

```
val videoView: VideoView = findViewById(R.id.videoView)

// Set video path from raw resource
val videoPath = "android.resource://" + packageName + "/" + R.raw.my_video_file
val uri = Uri.parse(videoPath)
videoView.setVideoURI(uri)

// Add media controls (play/pause, seek bar)
val mediaController = MediaController(this)
mediaController.setAnchorView(videoView)
videoView.setMediaController(mediaController)

videoView.setOnPreparedListener { mp ->
    mp.isLooping = true // Loop video
    videoView.start() // Start playback when prepared
}

videoView.setOnErrorListener { mp, what, extra ->
    Toast.makeText(this, "Video playback error: $what, $extra",
Toast.LENGTH_LONG).show()
    false // Return false to indicate the error was handled
}
```

Fragment

Fragments represent a behavior or a portion of user interface in an Activity. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse fragments across multiple activities.

Creating a Fragment

1. **Create a Kotlin/Java class:** Extend `androidx.fragment.app.Fragment`.
2. **Create a layout XML:** Define the UI for the fragment in `res/layout/fragment_my.xml`.
3. **Inflate the layout:** In the fragment's `onCreateView()`, inflate the layout.

Example (Kotlin):

```
// MyFragment.kt
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Button
import android.widget.TextView
import androidx.fragment.app.Fragment

class MyFragment : Fragment() {

    private lateinit var messageTextView: TextView
    private lateinit var clickButton: Button
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        val view = inflater.inflate(R.layout.fragment_my, container, false)
        messageTextView = view.findViewById(R.id.messageTextView)
        clickButton = view.findViewById(R.id.clickButton)
        return view
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        messageTextView.text = "Hello from Fragment!"
        clickButton.setOnClickListener {
            messageTextView.text = "Button clicked in Fragment!"
        }
    }
}
```

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

    <TextView
        android:id="@+id/messageTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="24sp"
        android:padding="16dp" />

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

</LinearLayout>
```

Fragment Life Cycle

Fragments have their own lifecycle, which is closely tied to the lifecycle of their host Activity.

- `onAttach(Context context)`: Called when the fragment is first associated with its activity.
- `onCreate(Bundle savedInstanceState)`: Called to do initial creation of the fragment (non-UI setup).

- `onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)`: Called to create and return the view hierarchy associated with the fragment.
- `onViewCreated(View view, Bundle savedInstanceState)`: Called immediately after `onCreateView` has returned, but before any saved state has been restored to the view. Good place to initialize views.
- `onActivityCreated(Bundle savedInstanceState)`: Called when the activity's `onCreate()` method has returned. (Deprecated since API 28, use `onViewCreated` or `onStart` instead).
- `onStart()`: Fragment becomes visible.
- `onResume()`: Fragment becomes active and visible.
- `onPause()`: Fragment is no longer interactive.
- `onStop()`: Fragment is no longer visible.
- `onDestroyView()`: View hierarchy associated with the fragment is being removed.
- `onDestroy()`: Fragment is no longer in use.
- `onDetach()`: Fragment is no longer associated with its activity.

Communication between Fragment and Activity

Fragments should not directly interact with their host Activity (or other fragments) to maintain modularity. Use interfaces for communication.

1. **Define an Interface:** In the fragment, define a public interface with methods for communication.
2. **Implement Interface in Activity:** The host Activity implements this interface.
3. **Pass Reference:** The fragment gets a reference to the Activity's implementation of the interface (e.g., in `onAttach()`).
4. **Call Method:** The fragment calls methods on the interface reference.

Example (Fragment to Activity):

```kotlin
// MyFragment.kt
interface MyFragmentListener {
    fun onButtonClicked(message: String)
}

private var listener: MyFragmentListener? = null

override fun onAttach(context: Context) {
    super.onAttach(context)
    if (context is MyFragmentListener) {
        listener = context
    } else {
        throw RuntimeException("$context must implement MyFragmentListener")
    }
}

override fun onDetach() {
    super.onDetach()
    listener = null
}

// ... inside onViewCreated() click listener
```

```
clickButton.setOnClickListener {
    listener?.onButtonClicked("Hello from Fragment Button!")
}
```

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

        // Add fragment to activity (example)
        if (savedInstanceState == null) {
            supportFragmentManager.beginTransaction()
                .add(R.id.fragment_container, MyFragment())
                .commit()
        }
    }

    override fun onButtonClicked(message: String) {
        Toast.makeText(this, "Activity received: $message",
Toast.LENGTH_SHORT).show()
    }
}
```

Fragment Transactions: Add, Replace, and Remove fragments

`FragmentManager` is used to manage fragments within an Activity. `FragmentTransaction` is used to perform operations (add, replace, remove, show, hide) on fragments.

- **`add()`:** Adds a fragment to the activity's view hierarchy.

  ```
  supportFragmentManager.beginTransaction()
      .add(R.id.fragment_container, MyFragment(), "myFragmentTag") // Add to a
  container view
      .commit() // Commit the transaction
  ```

- **`replace()`:** Replaces an existing fragment in a container with a new one. The old fragment's view is destroyed.

  ```
  supportFragmentManager.beginTransaction()
      .replace(R.id.fragment_container, AnotherFragment())
      .addToBackStack(null) // Allows going back to previous fragment with
  back button
      .commit()
  ```

- **`remove()`:** Removes a fragment from the activity's view hierarchy.

```
val fragment = supportFragmentManager.findFragmentByTag("myFragmentTag")
if (fragment != null) {
    supportFragmentManager.beginTransaction()
        .remove(fragment)
        .commit()
}
```

- **`commit()` vs `commitNow()` vs `commitAllowingStateLoss()`:** `commit()` is asynchronous. `commitNow()` is synchronous. `commitAllowingStateLoss()` allows loss if activity state is saved, but generally avoid unless specific need.
- **`addToBackStack()`:** Adds the transaction to the activity's back stack, allowing the user to navigate back to the previous fragment state by pressing the Back button.

List View

`ListView` is a UI component for displaying scrollable lists of items. While still functional, `RecyclerView` is the modern and recommended approach for lists.

Displaying lists in Android

- `ListView` shows items in a vertically scrollable list. Each item is represented by a `View`.

Adapter

- **Purpose:** An `Adapter` acts as a bridge between a data source (e.g., an `ArrayList` of strings or custom objects) and an `AdapterView` (like `ListView`, `Spinner`, `GridView`). It's responsible for fetching data and creating `View`s for each item in the list.

Simple lists using Array Adapter

`ArrayAdapter` is a basic adapter suitable for displaying lists of simple data (e.g., `String` arrays).

Example (Kotlin):

```
// In activity_main.xml
<ListView
    android:id="@+id/listView"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

// In MainActivity.kt
val listView: ListView = findViewById(R.id.listView)
val data = arrayOf("Apple", "Banana", "Orange", "Grape", "Strawberry")

// Create an ArrayAdapter using the default simple list item layout
val adapter = ArrayAdapter(this, android.R.layout.simple_list_item_1, data)
```

```
// Set the adapter to the ListView
listView.adapter = adapter
```

Custom List using Base Adapter

For more complex list items with custom layouts (e.g., an image, multiple text fields), you need to create a custom `Adapter` that typically extends `BaseAdapter` or `ArrayAdapter` (if still managing a simple array).

Steps:

1. **Define Custom Item Layout:** Create an XML layout file (e.g., `list_item_custom.xml`) for each row.
2. **Create Custom Adapter Class:**
    - Subclass `BaseAdapter`.
    - Override `getCount()`, `getItem()`, `getItemId()`.
    - Override `getView()`: This is where you inflate the custom item layout, find child views, and populate them with data. Implement the ViewHolder pattern for performance.

Example (Conceptual):

```
// list_item_custom.xml
<LinearLayout ...>
    <ImageView android:id="@+id/itemImageView" ... />
    <TextView android:id="@+id/itemTitle" ... />
    <TextView android:id="@+id/itemDescription" ... />
</LinearLayout>

// MyCustomAdapter.kt
class MyCustomAdapter(private val context: Context, private val dataList:
List<MyDataItem>) : BaseAdapter() {

    override fun getCount(): Int = dataList.size
    override fun getItem(position: Int): Any = dataList[position]
    override fun getItemId(position: Int): Long = position.toLong()

    override fun getView(position: Int, convertView: View?, parent: ViewGroup?):
View {
        val view: View
        val holder: ViewHolder

        if (convertView == null) {
            view = LayoutInflater.from(context).inflate(R.layout.list_item_custom,
parent, false)
            holder = ViewHolder(view)
            view.tag = holder
        } else {
            view = convertView
            holder = view.tag as ViewHolder
        }

        val item = dataList[position]
        holder.title.text = item.title
```

```
            holder.description.text = item.description
            // holder.imageView.setImageResource(item.imageResId) // Set image

            return view
        }

        private class ViewHolder(view: View) {
            val title: TextView = view.findViewById(R.id.itemTitle)
            val description: TextView = view.findViewById(R.id.itemDescription)
            // val imageView: ImageView = view.findViewById(R.id.itemImageView)
        }
    }
```

Various events on List View items

- `OnItemClickListener`: For detecting clicks on individual list items.

  ```
  listView.onItemClickListener = AdapterView.OnItemClickListener { parent,
  view, position, id ->
      val selectedItem = parent.getItemAtPosition(position).toString()
      Toast.makeText(this, "Clicked: $selectedItem",
  Toast.LENGTH_SHORT).show()
  }
  ```

- `OnItemLongClickListener`: For long presses on list items.
- `OnScrollListener`: For detecting scrolling events.

Introduction to Material Design

- **Material Design:** A design language developed by Google. It provides guidelines for visual, motion, and interaction design across platforms and devices.
- **Principles:** Emphasizes realistic lighting and shadows, bold graphics, intentional animation, and a consistent user experience.
- **Components:** Provides a rich set of pre-built UI components (e.g., Floating Action Buttons, CardViews, Bottom Navigation, Snackbars, Navigatoin Drawers) that adhere to Material Design guidelines.

Adding material design support to your app.

- **Dependency:** Include the Material Components for Android library in your `app/build.gradle` file:

  ```
  implementation 'com.google.android.material:material:1.10.0' // Use the
  latest stable version
  ```

- **Theme:** Ensure your app's theme inherits from a Material Design theme (e.g., `Theme.MaterialComponents.DayNight.NoActionBar`) in `res/values/themes.xml`.

- **Use Material Components:** Replace standard Android views with their Material Design counterparts (e.g., `com.google.android.material.button.MaterialButton` instead of `Button`, `com.google.android.material.textfield.TextInputLayout` for `EditText`).

Introduction to Recycler View

`RecyclerView` is the most powerful and flexible UI component for displaying large sets of data, making it the **recommended** choice for lists and grids in modern Android development.

List View vs Recycler View: Advantages of Recycler View

Feature	ListView	RecyclerView
Performance	Can be inefficient for large lists (creates many views)	Highly efficient due to view recycling and ViewHolder pattern
View Recycling	Basic recycling built-in	Enforces ViewHolder pattern, efficient recycling of views
Layout Management	Only vertical scrolling list	Separates layout management from view, supports various layouts (Linear, Grid, Staggered Grid)
Item Animations	No built-in item animations	Provides `ItemAnimator` for built-in and custom item animations
Item Decorations	Basic dividers	Flexible `ItemDecoration` for custom dividers, spacing, etc.
Touch Handling	Basic `OnItemClickListener`	Flexible `ItemTouchHelper` for swipe-to-dismiss, drag-and-drop
Setup	Simpler setup for basic lists	More boilerplate initially, but more powerful

Implementing `RecyclerView.Adapter` class

The `RecyclerView.Adapter` manages the data collection and binds it to the views displayed in `RecyclerView`.

Steps:

1. **Create Data Model:** Define a data class (e.g., `Product`).
2. **Create Item Layout:** Design the XML layout for a single item (e.g., `item_product.xml`).
3. **Create `ViewHolder` Class:**
   - Extend `RecyclerView.ViewHolder`.
   - Holds references to the `View`s within a single item layout to avoid `findViewById` calls repeatedly.
4. **Create Adapter Class:**
   - Extend `RecyclerView.Adapter<YourAdapter.YourViewHolder>`.
   - Override `onCreateViewHolder()`: Inflates the item layout and returns a new `ViewHolder` instance.
   - Override `onBindViewHolder()`: Binds data from your data source to the `ViewHolder`'s views.

- Override `getItemCount()`: Returns the total number of items in the data set.

Example (Kotlin):

```
// 1. Data Model (e.g., in a separate file)
data class Product(val id: Int, val name: String, val price: Double)

// 2. Item Layout (item_product.xml)
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="16dp">
    <TextView android:id="@+id/productName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="18sp"
        android:textStyle="bold" />
    <TextView android:id="@+id/productPrice"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16sp" />
</LinearLayout>

// 3. Adapter Class (e.g., in a separate file MyProductAdapter.kt)
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView

class ProductAdapter(private val productList: List<Product>) :
    RecyclerView.Adapter<ProductAdapter.ProductViewHolder>() {

    // 4. Implement ViewHolder
    class ProductViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val productName: TextView = itemView.findViewById(R.id.productName)
        val productPrice: TextView = itemView.findViewById(R.id.productPrice)
    }

    // Called when RecyclerView needs a new ViewHolder
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ProductViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.item_product, parent, false)
        return ProductViewHolder(view)
    }

    // Called to display the data at the specified position
    override fun onBindViewHolder(holder: ProductViewHolder, position: Int) {
        val product = productList[position]
        holder.productName.text = product.name
        holder.productPrice.text = "$${product.price}"
    }

    // Returns the total number of items in the data set
    override fun getItemCount(): Int = productList.size
}
```

Implementing `RecyclerView.ViewHolder` class

As seen above, `RecyclerView.ViewHolder` is a nested class within your `RecyclerView.Adapter`. It caches references to the individual `View`s inside each item layout, preventing repeated `findViewById()` calls and improving scroll performance.

Handling click events in RecyclerView

`RecyclerView` does not have a direct `OnItemClickListener` like `ListView`. You need to implement click handling manually, typically by setting an `OnClickListener` on the `itemView` within the `ViewHolder` or on individual views within the item.

Common approach: Define a callback interface in the Adapter:

```
// In ProductAdapter.kt (add to class definition)
class ProductAdapter(
    private val productList: List<Product>,
    private val clickListener: (Product) -> Unit // Lambda for click handling
) : RecyclerView.Adapter<ProductAdapter.ProductViewHolder>() {

    class ProductViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val productName: TextView = itemView.findViewById(R.id.productName)
        val productPrice: TextView = itemView.findViewById(R.id.productPrice)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ProductViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.item_product, parent, false)
        return ProductViewHolder(view)
    }

    override fun onBindViewHolder(holder: ProductViewHolder, position: Int) {
        val product = productList[position]
        holder.productName.text = product.name
        holder.productPrice.text = "$${product.price}"

        // Set click listener on the entire item view
        holder.itemView.setOnClickListener {
            clickListener(product) // Invoke the lambda passed from
Activity/Fragment
        }
    }
    // ... getItemCount()
}
```

```
// In your Activity/Fragment where RecyclerView is used
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val recyclerView: RecyclerView = findViewById(R.id.recyclerView)
        recyclerView.layoutManager = LinearLayoutManager(this) // Set a
LayoutManager

        val products = listOf(
            Product(1, "Laptop", 1200.0),
            Product(2, "Mouse", 25.0)
        )

        val adapter = ProductAdapter(products) { product ->
            Toast.makeText(this, "Clicked: ${product.name}",
Toast.LENGTH_SHORT).show()
            // Start a new activity, show details, etc.
        }
        recyclerView.adapter = adapter
    }
}
```

ActionBar (`Toolbar`)

The ActionBar was a core component for app navigation and actions in older Android versions. It has largely been replaced by the more flexible `Toolbar` in modern Android development.

Adding Action Bar items (deprecated for `Toolbar`)

In older Android versions, you would define menu items in `res/menu/main_menu.xml` and inflate them in `Activity.onCreateOptionsMenu()`.

Handling click events on ActionBar items (deprecated for `Toolbar`)

In older Android, you would override `Activity.onOptionsItemSelected()` to handle clicks.

ToolBar

- **Purpose:** `Toolbar` is a more flexible and customizable replacement for the traditional ActionBar. It's a `ViewGroup` that can be placed anywhere in your layout, providing more control over its appearance and behavior.
- **Benefits:** Can be themed independently, supports more complex layouts, allows placing any `View` inside it.

Using Toolbar as ActionBar

To get `Toolbar` to function as your Activity's ActionBar:

1. **Theme:** Set your Activity's theme to `NoActionBar` (e.g., `Theme.MaterialComponents.DayNight.NoActionBar`) in `res/values/themes.xml`.
2. **Add Toolbar to Layout:** Include a `Toolbar` in your Activity's layout XML.
3. **Set as ActionBar:** Call `setSupportActionBar()` in your Activity's `onCreate()`.

Example (XML):

```xml
<!-- activity_main.xml -->
<androidx.constraintlayout.widget.ConstraintLayout ...>

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="0dp"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" />

    <!-- Your other UI elements -->

</androidx.constraintlayout.widget.ConstraintLayout>
```

Example (Kotlin):

```kotlin
// MainActivity.kt
import androidx.appcompat.widget.Toolbar

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

        val toolbar: Toolbar = findViewById(R.id.toolbar)
        setSupportActionBar(toolbar) // Set the Toolbar as the Activity's
ActionBar
    }
}
```

Menus and Popups

Once you have a `Toolbar` set as an ActionBar, you can interact with menus in a familiar way.

- **Options Menu:** The menu that appears when you tap the three-dot icon (overflow menu).
  1. Define menu items in `res/menu/main_menu.xml`.
  2. Override `onCreateOptionsMenu(Menu menu)` in your Activity to inflate the menu.
  3. Override `onOptionsItemSelected(MenuItem item)` to handle item clicks.

- **`MenuItem` attributes:** `android:id`, `android:title`, `android:icon`, `app:showAsAction` (`ifRoom`, `always`, `withText`, `never`).

Example (`main_menu.xml`):

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/action_settings"
        android:title="Settings"
        android:icon="@drawable/ic_settings"
        app:showAsAction="ifRoom" />
    <item
        android:id="@+id/action_search"
        android:title="Search"
        app:showAsAction="collapseActionView|ifRoom"
        app:actionViewClass="androidx.appcompat.widget.SearchView" />
</menu>
```

Example (Kotlin):

```kotlin
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.main_menu, menu)
    return true
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.action_settings -> {
            Toast.makeText(this, "Settings clicked", Toast.LENGTH_SHORT).show()
            true
        }
        R.id.action_search -> {
            Toast.makeText(this, "Search clicked", Toast.LENGTH_SHORT).show()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

Contextual Action Modes (`ActionMode`)

- **Purpose:** Provides a temporary `ActionBar` that overlays the app's `ActionBar` when the user selects items or performs specific actions (e.g., long-pressing on a list item to select multiple items).
- **Use Cases:** Multi-selection, batch actions.
- **Implementation:** Implement `ActionMode.Callback` and start `ActionMode` via `startSupportActionMode()`.

PopupMenu (`PopupMenu`)

- **Purpose:** Displays a modal popup menu anchored to a `View`.
- **Use Cases:** Contextual menus for individual items, overflow menus for small groups of actions.
- **Example:**

  ```kotlin
  val button: Button = findViewById(R.id.popupMenuButton)
  button.setOnClickListener { view ->
      val popup = PopupMenu(this, view)
      popup.menuInflater.inflate(R.menu.popup_menu_example, popup.menu)
      popup.setOnMenuItemClickListener { item ->
          when (item.itemId) {
              R.id.action_one -> { Toast.makeText(this, "Item One",
  Toast.LENGTH_SHORT).show(); true }
              R.id.action_two -> { Toast.makeText(this, "Item Two",
  Toast.LENGTH_SHORT).show(); true }
              else -> false
          }
      }
      popup.show()
  }
  ```

PopupWindow (`PopupWindow`)

- **Purpose:** A floating container that can display any `View` content on top of the current Activity window. More flexible than `PopupMenu`.
- **Use Cases:** Custom dropdowns, tooltips, hints.

Tab-based UI using View Pagers (`ViewPager2`)

- **`ViewPager2`:** (Recommended, replaces `ViewPager`) A `ViewGroup` that allows users to swipe horizontally between pages of content. It's commonly used with `TabLayout` to provide a tabbed UI.
- **`TabLayout` (Material Design):** Displays a horizontal strip of tabs.

Steps:

1. **Add Dependencies:** `com.google.android.material:material` for `TabLayout`, `androidx.viewpager2:viewpager2`.
2. **Layout:** Add `TabLayout` and `ViewPager2` to your XML.
3. **Fragment State Adapter:** Create a custom `FragmentStateAdapter` to provide fragments for each page.
4. **Connect `TabLayout` and `ViewPager2`:** Use `TabLayoutMediator`.

Example (Conceptual):

```xml
<!-- activity_main.xml -->
<LinearLayout ...>
    <com.google.android.material.tabs.TabLayout
        android:id="@+id/tabLayout" ... />
```

```
    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/viewPager" ... />
</LinearLayout>
```

```
// MainActivity.kt
import com.google.android.material.tabs.TabLayout
import com.google.android.material.tabs.TabLayoutMediator
import androidx.viewpager2.adapter.FragmentStateAdapter
import androidx.viewpager2.widget.ViewPager2

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

        val tabLayout: TabLayout = findViewById(R.id.tabLayout)
        val viewPager: ViewPager2 = findViewById(R.id.viewPager)

        val adapter = ViewPagerAdapter(this)
        viewPager.adapter = adapter

        // Connect TabLayout and ViewPager2
        TabLayoutMediator(tabLayout, viewPager) { tab, position ->
            tab.text = when (position) {
                0 -> "Tab 1"
                1 -> "Tab 2"
                else -> "Tab 3"
            }
        }.attach()
    }

    // ViewPagerAdapter.kt (nested class or separate file)
    private inner class ViewPagerAdapter(activity: AppCompatActivity) :
FragmentStateAdapter(activity) {
        override fun getItemCount(): Int = 3 // Number of tabs

        override fun createFragment(position: Int): Fragment {
            return when (position) {
                0 -> MyFragment.newInstance("Content for Tab 1")
                1 -> MyFragment.newInstance("Content for Tab 2")
                else -> MyFragment.newInstance("Content for Tab 3")
            }
        }
    }
}
```

Navigation Drawer (`DrawerLayout`)

- **Purpose:** A panel that slides out from the side of the screen (typically left) to reveal navigation options.

- `DrawerLayout` **(AndroidX):** The root layout for a navigation drawer. It contains two child views: the main content view and the navigation drawer view.
- `NavigationView` **(Material Design):** A common view to populate the content of the navigation drawer with menu items.

Steps:

1. **Add Dependencies:** `com.google.android.material:material`.
2. **Layout:** Use `DrawerLayout` as the root, containing your main content layout and a `NavigationView`.
3. **Toggle Drawer:** Use a `Toolbar` to show a drawer toggle icon.
4. **Handle Item Clicks:** Set `NavigationView.setNavigationItemSelectedListener`.

Example (Conceptual):

```xml
<!-- activity_main.xml -->
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">

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

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" />

        <!-- Your main activity content here -->

    </LinearLayout>

    <!-- Navigation drawer view -->
    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_main"
        app:menu="@menu/activity_main_drawer" />

</androidx.drawerlayout.widget.DrawerLayout>
```

```
// MainActivity.kt
import androidx.appcompat.app.ActionBarDrawerToggle
import androidx.drawerlayout.widget.DrawerLayout
import com.google.android.material.navigation.NavigationView

class MainActivity : AppCompatActivity(),
NavigationView.OnNavigationItemSelectedListener {

    private lateinit var drawerLayout: DrawerLayout

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

        val toolbar: Toolbar = findViewById(R.id.toolbar)
        setSupportActionBar(toolbar)

        drawerLayout = findViewById(R.id.drawer_layout)
        val navView: NavigationView = findViewById(R.id.nav_view)

        val toggle = ActionBarDrawerToggle(
            this, drawerLayout, toolbar, R.string.navigation_drawer_open,
R.string.navigation_drawer_close
        )
        drawerLayout.addDrawerListener(toggle)
        toggle.syncState()

        navView.setNavigationItemSelectedListener(this)
    }

    override fun onNavigationItemSelected(item: MenuItem): Boolean {
        when (item.itemId) {
            R.id.nav_home -> Toast.makeText(this, "Home selected",
Toast.LENGTH_SHORT).show()
            R.id.nav_gallery -> Toast.makeText(this, "Gallery selected",
Toast.LENGTH_SHORT).show()
            // ...
        }
        drawerLayout.closeDrawer(GravityCompat.START)
        return true
    }

    override fun onBackPressed() {
        if (drawerLayout.isDrawerOpen(GravityCompat.START)) {
            drawerLayout.closeDrawer(GravityCompat.START)
        } else {
            super.onBackPressed()
        }
    }
}
```

Android Storage

Android provides several options for storing data persistently on the device, each suited for different use cases.

Storing data in Shared Preferences

- **Purpose:** Store small amounts of primitive data (key-value pairs) in XML files. Best for user preferences, settings, or simple session data.
- **Data Types:** `Boolean`, `Float`, `Int`, `Long`, `String`, and `Set<String>`.
- **Privacy:** Private to the application.
- **Retrieval:** Use `Context.getSharedPreferences()`.

Example (Kotlin):

```
// Save data
val sharedPref = getSharedPreferences("MyPrefs", Context.MODE_PRIVATE)
with (sharedPref.edit()) {
    putString("username", "JohnDoe")
    putBoolean("notifications_enabled", true)
    apply() // Asynchronously commit changes
    // or commit() for synchronous (less common due to potential UI blocking)
}

// Retrieve data
val username = sharedPref.getString("username", "Guest")
val notificationsEnabled = sharedPref.getBoolean("notifications_enabled", false)

// Remove data
sharedPref.edit().remove("username").apply()
```

Storing organized Data into SQLite

- **Purpose:** Store larger amounts of structured, private data using a relational database. Android includes built-in SQLite support.
- **Database:** Each application has its own private SQLite database.
- **`SQLiteOpenHelper`:** A helper class to manage database creation and version upgrades.
- **`SQLiteDatabase`:** Represents the SQLite database. Provides methods for executing SQL commands.

Content Values and Cursors

- **`ContentValues`:** A class used to insert or update a set of column_name/value pairs into a database table. It's similar to a `HashMap`.

  ```
  val values = ContentValues().apply {
      put("column_name", "value")
      put("another_column", 123)
  }
  ```

- `Cursor`: An interface that provides random read-write access to the result set returned by a database query. It's essentially a pointer to a row in the result set.
  - You iterate over a `Cursor` to read data.
  - Always close the `Cursor` after use to prevent memory leaks.

CRUD operations on SQLite DB

Steps:

1. **Define Schema:** Create constants for table names, column names.
2. **Create `SQLiteOpenHelper` subclass:** Implement `onCreate()` (for table creation) and `onUpgrade()` (for database version changes).
3. **Get `SQLiteDatabase` instance:** Use `getWritableDatabase()` for write operations, `getReadableDatabase()` for read-only.

Example (Simplified DB Helper and CRUD):

```
// DatabaseHelper.kt
import android.content.ContentValues
import android.content.Context
import android.database.Cursor
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper

class MyDbHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME,
null, DATABASE_VERSION) {

    companion object {
        const val DATABASE_NAME = "MyDatabase.db"
        const val DATABASE_VERSION = 1
        const val TABLE_USERS = "users"
        const val COLUMN_ID = "_id" // Often required for CursorAdapter with
ListView
        const val COLUMN_NAME = "name"
        const val COLUMN_EMAIL = "email"

        const val SQL_CREATE_ENTRIES =
            "CREATE TABLE ${TABLE_USERS} (" +
                    "$COLUMN_ID INTEGER PRIMARY KEY AUTOINCREMENT," +
                    "$COLUMN_NAME TEXT," +
                    "$COLUMN_EMAIL TEXT UNIQUE)"

        const val SQL_DELETE_ENTRIES = "DROP TABLE IF EXISTS ${TABLE_USERS}"
    }

    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(SQL_CREATE_ENTRIES)
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        db.execSQL(SQL_DELETE_ENTRIES)
        onCreate(db)
    }

    // MARK: - CRUD Operations

    // Create (Insert)
    fun insertUser(name: String, email: String): Long {
        val db = writableDatabase
        val values = ContentValues().apply {
            put(COLUMN_NAME, name)
            put(COLUMN_EMAIL, email)
        }
        val newRowId = db.insert(TABLE_USERS, null, values)
        db.close()
        return newRowId
    }

    // Read (Query)
    fun getAllUsers(): Cursor {
        val db = readableDatabase
        return db.query(
            TABLE_USERS,
            arrayOf(COLUMN_ID, COLUMN_NAME, COLUMN_EMAIL),
            null, null, null, null, null
        )
    }

    // Update
    fun updateUser(id: Long, newName: String, newEmail: String): Int {
        val db = writableDatabase
        val values = ContentValues().apply {
            put(COLUMN_NAME, newName)
            put(COLUMN_EMAIL, newEmail)
        }
        val count = db.update(
            TABLE_USERS,
            values,
            "$COLUMN_ID = ?",
            arrayOf(id.toString())
        )
        db.close()
        return count
    }

    // Delete
    fun deleteUser(id: Long): Int {
        val db = writableDatabase
        val count = db.delete(
            TABLE_USERS,
            "$COLUMN_ID = ?",
            arrayOf(id.toString())
        )
        db.close()
```

```
            return count
        }
    }
```

```
    // In MainActivity.kt
    class MainActivity : AppCompatActivity() {
        private lateinit var dbHelper: MyDbHelper

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

            dbHelper = MyDbHelper(this)

            // Example Usage:
            val newUserId = dbHelper.insertUser("Alice", "alice@example.com")
            println("New user inserted with ID: $newUserId")

            val cursor = dbHelper.getAllUsers()
            if (cursor.moveToFirst()) {
                do {
                    val id =
    cursor.getLong(cursor.getColumnIndexOrThrow(MyDbHelper.COLUMN_ID))
                    val name =
    cursor.getString(cursor.getColumnIndexOrThrow(MyDbHelper.COLUMN_NAME))
                    val email =
    cursor.getString(cursor.getColumnIndexOrThrow(MyDbHelper.COLUMN_EMAIL))
                    println("User: ID=$id, Name=$name, Email=$email")
                } while (cursor.moveToNext())
            }
            cursor.close()

            // dbHelper.updateUser(newUserId, "Alice Smith",
    "alice.smith@example.com")
            // dbHelper.deleteUser(newUserId)
        }
    }
```

Content Providers

Content Providers are standard interfaces for sharing data between applications. They abstract the underlying data storage mechanism.

Access mobile contacts/SMS using Content Provider

Android's system apps (like Contacts, SMS) expose their data via Content Providers. You access them using a `ContentResolver` and a `Cursor`.

- **Permissions:** You need to declare appropriate permissions in `AndroidManifest.xml` (e.g., `READ_CONTACTS`, `READ_SMS`) and request them at runtime.
- **`ContentResolver`:** Provides a generic API to perform CRUD operations on any Content Provider.
- **`Uri`:** Identifies the data in the Content Provider (e.g., `ContactsContract.Contacts.CONTENT_URI`).

Example (Reading Contacts - Conceptual):

```
// In AndroidManifest.xml
<uses-permission android:name="android.permission.READ_CONTACTS" />

// In your Activity/Fragment (after requesting runtime permission)
fun readContacts() {
    val projection = arrayOf(
        ContactsContract.Contacts._ID,
        ContactsContract.Contacts.DISPLAY_NAME,
        ContactsContract.Contacts.HAS_PHONE_NUMBER
    )
    val cursor: Cursor? = contentResolver.query(
        ContactsContract.Contacts.CONTENT_URI,
        projection,
        null, null, null
    )

    cursor?.use {
        val idColumn = it.getColumnIndexOrThrow(ContactsContract.Contacts._ID)
        val nameColumn =
it.getColumnIndexOrThrow(ContactsContract.Contacts.DISPLAY_NAME)
        val hasPhoneColumn =
it.getColumnIndexOrThrow(ContactsContract.Contacts.HAS_PHONE_NUMBER)

        while (it.moveToNext()) {
            val contactId = it.getLong(idColumn)
            val displayName = it.getString(nameColumn)
            val hasPhoneNumber = it.getInt(hasPhoneColumn) > 0
            println("Contact: ID=$contactId, Name=$displayName, Has
Phone=$hasPhoneNumber")

            if (hasPhoneNumber) {
                // Query for phone numbers for this contact
                val phoneCursor: Cursor? = contentResolver.query(
                    ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
                    null,
                    ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = ?",
                    arrayOf(contactId.toString()),
                    null
                )
                phoneCursor?.use { phCur ->
                    val phoneNumColumn =
phCur.getColumnIndexOrThrow(ContactsContract.CommonDataKinds.Phone.NUMBER)
                    while (phCur.moveToNext()) {
                        val phoneNumber = phCur.getString(phoneNumColumn)
                        println("  Phone: $phoneNumber")
                        }
                    }
                }
            }
        }
    }
```

Create content provider for custom DB

You can create your own Content Provider to expose your app's private data (e.g., SQLite database) to other applications in a structured and secure way.

Steps:

1. **Define Contract Class:** A public `final` class that defines URIs, table names, column names.
2. **Create `ContentProvider` subclass:**
   - Override `onCreate()`: Initialize database helper.
   - Override `query()`, `insert()`, `update()`, `delete()`, `getType()`.
   - Implement URI matching using `UriMatcher`.
3. **Declare in `AndroidManifest.xml`:**

   ```xml
   <provider
       android:name=".MyCustomContentProvider"
       android:authorities="com.example.myapp.myprovider"
       android:exported="false" <!-- Usually false for app-private data -->
       android:grantUriPermissions="false" />
   ```

Multi-Tasking in Android

Android supports multi-tasking by running each application in its own Linux process, managing threads within these processes.

Thread mechanism in Android

- **Main Thread (UI Thread):** The single thread that handles all UI operations (drawing, event handling). Long-running operations on this thread will cause ANRs (Application Not Responding) errors.
- **Worker Threads (Background Threads):** Any other thread used for long-running operations (network requests, heavy computations) to keep the UI responsive.

Java Threads and Runnable interface

Standard Java threading mechanisms can be used in Android.

- **`Thread` class:**

  ```
  Thread {
      // Long-running operation
  ```

```
    println("Running on a new thread (Thread class)")
}.start()
```

- `Runnable` **interface:**

```kotlin
val myRunnable = Runnable {
    // Long-running operation
    println("Running on a new thread (Runnable interface)")
}
Thread(myRunnable).start()
```

Threads and Thread Handlers (`Handler`)

- **`Handler`**: Allows you to send and process `Message` and `Runnable` objects associated with a specific thread's `Looper`. Crucial for communicating back to the main (UI) thread from a background thread.

```kotlin
// In Activity/Fragment
private val uiHandler = Handler(Looper.getMainLooper()) // Handler on the main thread

fun doBackgroundWork() {
    Thread {
        // Long-running task
        val result = "Background task completed!"

        // Post result back to UI thread
        uiHandler.post {
            myTextView.text = result
            Toast.makeText(this, "UI updated!", Toast.LENGTH_SHORT).show()
        }
    }.start()
}
```

AsyncTask (`AsyncTask` - Deprecated, `Kotlin Coroutines`, `ExecutorService` - Recommended)

- **`AsyncTask`**: (DEPRECATED in API 30) A helper class for performing short background operations and publishing results on the UI thread without dealing with threads and handlers directly.
- **Modern Alternatives:**
  - **Kotlin Coroutines:** The recommended way for asynchronous programming in Kotlin.
  - **`ExecutorService` / `ThreadPoolExecutor`**: For managing a pool of threads for concurrent tasks.
  - **`LiveData` / `Flow` with Coroutines:** For reactive data streams.

Services in Android

Services are application components that can perform long-running operations in the background without a UI.

Types of Service

1. **Started Service:**
   - **Purpose:** A service that is started when an application component (e.g., an Activity) calls `startService()` and runs indefinitely in the background until it's explicitly stopped by `stopService()` or `stopSelf()`, or it completes its work.
   - **Lifecycle:** `onCreate()` -> `onStartCommand()` -> `onDestroy()`.
   - **Use Cases:** Downloading a file, playing music.
2. **Bounded Service:**
   - **Purpose:** A service that allows other application components to bind to it and interact with it (IPC). Multiple components can bind to the same service. The service runs as long as at least one client is bound to it.
   - **Lifecycle:** `onCreate()` -> `onBind()` -> `onUnbind()` -> `onDestroy()`.
   - **Use Cases:** Client-server communication within an app or across apps, live data updates.

Service Life Cycle

- **`onCreate()`:** Called when the service is first created. Perform one-time setup here.
- **`onStartCommand(Intent intent, int flags, int startId)`:** (For Started Services) Called every time a client calls `startService()`. Return value indicates how the system should handle the service if it's killed.
- **`onBind(Intent intent)`:** (For Bounded Services) Called when another component binds to the service. Returns an `IBinder` object that clients can use to interact with the service.
- **`onUnbind(Intent intent)`:** (For Bounded Services) Called when all clients have unbound from the service.
- **`onDestroy()`:** Called when the service is no longer used and is being destroyed. Perform final cleanup.

Starting a service

- **`startService(Intent)`:** For started services.
- **`bindService(Intent, ServiceConnection, flags)`:** For bounded services.

Example (Started Service - Kotlin):

```kotlin
// MyStartedService.kt
class MyStartedService : Service() {
    private val TAG = "MyStartedService"

    override fun onCreate() {
        super.onCreate()
        Log.d(TAG, "Service Created")
    }

    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        Log.d(TAG, "Service Started")
        // Perform long-running task in a background thread
        GlobalScope.launch(Dispatchers.IO) { // Using Kotlin Coroutines
            for (i in 0..10) {
                delay(1000)
                Log.d(TAG, "Service running... $i")
            }
            stopSelf() // Stop service when work is done
        }
        return START_STICKY // If killed by system, try to restart with null
intent
    }

    override fun onBind(intent: Intent?): IBinder? {
        return null // Not a bounded service
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.d(TAG, "Service Destroyed")
    }
}

// In AndroidManifest.xml
<service android:name=".MyStartedService" />

// From Activity to start/stop
startService(Intent(this, MyStartedService::class.java))
// stopService(Intent(this, MyStartedService::class.java))
```

IPC between Service and Activity using Binder

For bounded services, the `onBind()` method returns an `IBinder` object that clients use to interact with the service. This `IBinder` typically points to a custom `Binder` subclass defined within the service.

Example (Bounded Service - Conceptual):

```
// MyBoundedService.kt
class MyBoundedService : Service() {
    private val binder = MyBinder() // Instance of our custom Binder

    inner class MyBinder : Binder() {
        fun getService(): MyBoundedService = this@MyBoundedService // Return
service instance
    }

    override fun onBind(intent: Intent?): IBinder = binder

    fun doSomeWork(data: String): String {
        return "Service processed: $data"
    }
}

// In Activity to bind
private lateinit var myService: MyBoundedService
private var isBound = false

private val connection = object : ServiceConnection {
    override fun onServiceConnected(className: ComponentName, service: IBinder) {
        val binder = service as MyBoundedService.MyBinder
        myService = binder.getService()
        isBound = true
        Log.d("MainActivity", "Service Bound. Calling service method.")
        val result = myService.doSomeWork("Hello from Activity!")
        Toast.makeText(this@MainActivity, result, Toast.LENGTH_LONG).show()
    }

    override fun onServiceDisconnected(arg0: ComponentName) {
        isBound = false
        Log.d("MainActivity", "Service Unbound.")
    }
}

override fun onCreate(savedInstanceState: Bundle?) {
    // ...
    val bindButton: Button = findViewById(R.id.bindButton)
    bindButton.setOnClickListener {
        Intent(this, MyBoundedService::class.java).also { intent ->
            bindService(intent, connection, Context.BIND_AUTO_CREATE)
        }
    }
    // Call unbindService(connection) in onDestroy or onStop
}
```

Intent Service (`IntentService` - Deprecated)

- **`IntentService`:** (DEPRECATED in API 30) A subclass of `Service` that handles asynchronous requests on a single worker thread. It automatically stops itself when its work is done.
- **Use Cases:** Short, independent, queue-like background operations (e.g., uploading log data).
- **Modern Alternative:** Use a plain `Service` with Kotlin Coroutines or `WorkManager`.

Broadcast Receiver

Broadcast Receivers allow applications to receive and respond to system-wide or app-specific broadcast messages.

- **Registration:**
  - **Manifest-declared receivers:** Declared in `AndroidManifest.xml`. Can be launched by the system even if the app is not running.
  - **Context-registered receivers:** Registered dynamically in code using `Context.registerReceiver()`. Only active as long as the registering context (Activity, Service) is alive.

- `onReceive(Context context, Intent intent)`: The method called when a broadcast is received. It should perform very quick operations, as it runs on the main thread. For long-running tasks, it should offload to a Service or `WorkManager`.

Handle custom app-based event.

You can define custom broadcast actions for communication within your own app.

```
// Sending a custom broadcast
val customIntent = Intent("com.example.myapp.CUSTOM_ACTION").apply {
    putExtra("message", "Data from custom broadcast")
}
sendBroadcast(customIntent)

// Registering (in MainActivity.kt)
private val customReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        if (intent?.action == "com.example.myapp.CUSTOM_ACTION") {
            val message = intent.getStringExtra("message")
            Toast.makeText(context, "Custom broadcast received: $message",
Toast.LENGTH_LONG).show()
        }
    }
}

override fun onResume() {
    super.onResume()
    val filter = IntentFilter("com.example.myapp.CUSTOM_ACTION")
    registerReceiver(customReceiver, filter)
}

override fun onPause() {
    super.onPause()
    unregisterReceiver(customReceiver)
}
```

Handle system event.

Broadcast Receivers are commonly used to listen for system-wide events.

```
// In AndroidManifest.xml (for a manifest-declared receiver)
<receiver android:name=".BootCompletedReceiver" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>

// BootCompletedReceiver.kt
class BootCompletedReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        if (intent?.action == Intent.ACTION_BOOT_COMPLETED) {
            Toast.makeText(context, "Device Boot Completed!",
Toast.LENGTH_LONG).show()
            // Start a service or perform background work
        }
    }
}
```

Android Location Based Services

Android provides powerful APIs for determining device location, displaying maps, and interacting with location data.

Fused Location using Google API (`FusedLocationProviderClient`)

- **`FusedLocationProviderClient`:** Part of Google Play Services, it's the recommended API for location services. It intelligently combines various location sources (GPS, Wi-Fi, cell towers) to provide optimal accuracy and battery efficiency.
- **Permissions:** `ACCESS_FINE_LOCATION` (GPS) and/or `ACCESS_COARSE_LOCATION` (Wi-Fi/Cell towers). Request at runtime.
- **Features:**
  - `getLastLocation()`: Get the last known location.
  - `requestLocationUpdates()`: Get continuous location updates.
  - Location Settings Dialog: Prompts user to enable location services if disabled.

Example (Getting last location - Kotlin):

```
// build.gradle (app-level)
implementation 'com.google.android.gms:play-services-location:21.0.1'

// AndroidManifest.xml
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

// In Activity
private lateinit var fusedLocationClient: FusedLocationProviderClient

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    fusedLocationClient = LocationServices.getFusedLocationProviderClient(this)

    // Example button click to get location
    findViewById<Button>(R.id.getLocationButton).setOnClickListener {
        checkLocationPermission()
    }
}

private fun checkLocationPermission() {
    if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
        getLastLocation()
    } else {

requestLocationPermissionLauncher.launch(Manifest.permission.ACCESS_FINE_LOCATION)
    }
}

private val requestLocationPermissionLauncher =
registerForActivityResult(ActivityResultContracts.RequestPermission()) {
isGranted: Boolean ->
    if (isGranted) {
        getLastLocation()
    } else {
        Toast.makeText(this, "Location permission denied.",
Toast.LENGTH_SHORT).show()
    }
}

@SuppressLint("MissingPermission") // Suppress lint warning, as permission is
checked
private fun getLastLocation() {
    fusedLocationClient.lastLocation
        .addOnSuccessListener { location : Location? ->
            if (location != null) {
                val lat = location.latitude
                val lon = location.longitude
                Toast.makeText(this, "Lat: $lat, Lon: $lon",
Toast.LENGTH_LONG).show()
            } else {
                Toast.makeText(this, "Last location unknown.",
Toast.LENGTH_SHORT).show()
            }
        }
        .addOnFailureListener { e ->
            Toast.makeText(this, "Failed to get location: ${e.message}",
Toast.LENGTH_LONG).show()
        }
}
```

Displaying Map, marking the current location, Adding various map options (`Google Maps Android API`)

- **Google Maps Android API:** Allows you to embed Google Maps directly into your app.
- **Dependencies:** `com.google.android.gms:play-services-maps`.
- **API Key:** Requires a Google Maps API key (obtained from Google Cloud Console) in your `AndroidManifest.xml`.
- **`SupportMapFragment` or `MapView`:** To display the map.
- **`GoogleMap` object:** The main class for map interaction once the map is ready.

Example (Conceptual - simplified):

```
// build.gradle (app-level)
implementation 'com.google.android.gms:play-services-maps:18.1.0'

// AndroidManifest.xml
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="YOUR_GOOGLE_MAPS_API_KEY" />

// activity_main.xml
<fragment
    android:id="@+id/map"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

// MainActivity.kt
import com.google.android.gms.maps.GoogleMap
import com.google.android.gms.maps.OnMapReadyCallback
import com.google.android.gms.maps.SupportMapFragment
import com.google.android.gms.maps.model.LatLng
import com.google.android.gms.maps.model.MarkerOptions

class MainActivity : AppCompatActivity(), OnMapReadyCallback {

    private lateinit var googleMap: GoogleMap

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

        val mapFragment = supportFragmentManager.findFragmentById(R.id.map) as
SupportMapFragment
        mapFragment.getMapAsync(this)
    }

    override fun onMapReady(map: GoogleMap) {
        googleMap = map
        val sydney = LatLng(-34.0, 151.0) // Example location
        googleMap.addMarker(MarkerOptions().position(sydney).title("Marker in
Sydney"))
        googleMap.moveCamera(CameraUpdateFactory.newLatLngZoom(sydney, 10f)) //
Zoom level

        // Add current location layer (requires ACCESS_FINE_LOCATION permission)
        if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
            googleMap.isMyLocationEnabled = true
        }

        // Various map options
        googleMap.uiSettings.isZoomControlsEnabled = true
        googleMap.uiSettings.isCompassEnabled = true
        googleMap.uiSettings.isMyLocationButtonEnabled = true
    }
}
```

Regular location update using `LocationListener` (Deprecated for FusedLocation)

- **`LocationListener`:** An older API for receiving location updates directly from `LocationManager`.
- **Limitations:** More verbose, less efficient than Fused Location Provider.
- **Modern Android:** Use `FusedLocationProviderClient` for location updates.

Telephony Manager Call and SMS (`TelephonyManager`, `SmsManager`)

- **`TelephonyManager`:** Provides information about the telephony services on the device (e.g., network type, device ID, call state).
- **`SmsManager`:** Sends SMS messages.
- **Permissions:** `READ_PHONE_STATE`, `CALL_PHONE`, `SEND_SMS`, `RECEIVE_SMS`, `READ_SMS`.

Example (Making a Call):

```
// AndroidManifest.xml
<uses-permission android:name="android.permission.CALL_PHONE" />

// In Activity (after requesting runtime permission)
fun makeCall(phoneNumber: String) {
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.CALL_PHONE) ==
PackageManager.PERMISSION_GRANTED) {
        val intent = Intent(Intent.ACTION_CALL).apply {
            data = Uri.parse("tel:$phoneNumber")
        }
        startActivity(intent)
    } else {
        // Request CALL_PHONE permission
    }
}
```

Example (Sending SMS):

```
// AndroidManifest.xml
<uses-permission android:name="android.permission.SEND_SMS" />

// In Activity (after requesting runtime permission)
fun sendSms(phoneNumber: String, message: String) {
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS) ==
PackageManager.PERMISSION_GRANTED) {
        SmsManager.getDefault().sendTextMessage(phoneNumber, null, message, null,
null)
        Toast.makeText(this, "SMS Sent!", Toast.LENGTH_SHORT).show()
    } else {
        // Request SEND_SMS permission
    }
}
```

Bluetooth

Android's Bluetooth APIs allow apps to perform common Bluetooth tasks like scanning for devices, pairing, and managing connections.

- **Permissions:** `BLUETOOTH`, `BLUETOOTH_ADMIN`, `ACCESS_FINE_LOCATION` (for scanning), `BLUETOOTH_SCAN`, `BLUETOOTH_ADVERTISE`, `BLUETOOTH_CONNECT` (Android 12+).
- **`BluetoothAdapter`:** Represents the local Bluetooth adapter.
- **Use Cases:** Connecting to wearables, headphones, IoT devices.

Android Sensors: Proximity Sensor (`SensorManager`, `SensorEvent`)

Android devices have various built-in sensors (accelerometer, gyroscope, light sensor, proximity sensor, etc.).

- **`SensorManager`:** Allows you to access and manage device sensors.
- **`Sensor`:** Represents a specific sensor on the device.
- **`SensorEventListener`:** Interface for receiving sensor data changes.

Example (Proximity Sensor):

```
// In Activity
private lateinit var sensorManager: SensorManager
private var proximitySensor: Sensor? = null

private val proximitySensorListener = object : SensorEventListener {
    override fun onSensorChanged(event: SensorEvent?) {
        if (event?.sensor?.type == Sensor.TYPE_PROXIMITY) {
            val distance = event.values[0]
            if (distance == 0.0f) { // Often 0.0f means "near" on most devices
                Toast.makeText(this@MainActivity, "Object is Near!",
Toast.LENGTH_SHORT).show()
            } else {
                Toast.makeText(this@MainActivity, "Object is Far!",
Toast.LENGTH_SHORT).show()
            }
        }
    }
    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {
        // Not used for proximity sensor usually
    }
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
    proximitySensor = sensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY)
    if (proximitySensor == null) {
        Toast.makeText(this, "Proximity sensor not available.",
Toast.LENGTH_SHORT).show()
    }
}

override fun onResume() {
    super.onResume()
    proximitySensor?.let { sensor ->
        sensorManager.registerListener(proximitySensorListener, sensor,
SensorManager.SENSOR_DELAY_NORMAL)
    }
}

override fun onPause() {
    super.onPause()
    sensorManager.unregisterListener(proximitySensorListener) // Unregister to
save battery
}
```

Consuming REST API

Modern Android apps frequently interact with backend servers to fetch and send data via RESTful APIs.

Using 3rd party library eg- Using Retrofit Libraries

While `HttpURLConnection` (or OkHttp directly) can be used, libraries greatly simplify REST API consumption.

- **Retrofit:** (RECOMMENDED) A type-safe HTTP client for Android and Java. It simplifies making HTTP requests by converting API endpoints into Java/Kotlin interfaces. It uses `OkHttp` internally.
- **Dependencies:** `com.squareup.retrofit2:retrofit`, `com.squareup.retrofit2:converter-gson` (for JSON parsing).

Steps:

1. **Define API Interface:** Create a Kotlin interface that defines your API endpoints with annotations.
2. **Create Retrofit Instance:** Build a `Retrofit` object, specifying the base URL and converters (e.g., GSON).
3. **Make Requests:** Create an instance of your API interface and call its methods. Handle responses using callbacks or Coroutines.

Example (Kotlin with Retrofit and Coroutines):

```
// build.gradle (app-level)
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.1'
implementation 'com.jakewharton.retrofit:retrofit2-kotlin-coroutines-
adapter:0.9.2' // For Coroutine support
```

```
// 1. Data Model (same as before for Codable)
// data class Product(val id: Int, val name: String, val price: Double, val
description: String?)

// 2. API Interface
interface ProductApiService {
    @GET("products")
    suspend fun getProducts(): List<Product> // suspend function for Coroutines

    @POST("products")
    suspend fun createProduct(@Body product: Product): Product
}

// 3. Retrofit Instance (e.g., in a Singleton object or injected)
object RetrofitClient {
    private const val BASE_URL = "https://api.example.com/" // Replace with your
API base URL

    val instance: ProductApiService by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create()) // For JSON
parsing
            .build()
            .create(ProductApiService::class.java)
    }
}

// 4. Making Requests in an Activity/Fragment using Coroutines
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        findViewById<Button>(R.id.fetchButton).setOnClickListener {
            fetchProducts()
        }
        findViewById<Button>(R.id.createButton).setOnClickListener {
            val newProduct = Product(id = 0, name = "Retrofit Test", price = 99.9,
description = null)
            createProduct(newProduct)
        }
    }

    private fun fetchProducts() {
        lifecycleScope.launch { // Use lifecycleScope for coroutine tied to
lifecycle
            try {
                val products = RetrofitClient.instance.getProducts()
                withContext(Dispatchers.Main) {
                    Toast.makeText(this@MainActivity, "Fetched ${products.size}
products", Toast.LENGTH_SHORT).show()
                    // Update RecyclerView or other UI
                }
            } catch (e: Exception) {
                withContext(Dispatchers.Main) {
                    Toast.makeText(this@MainActivity, "Error fetching products:
${e.message}", Toast.LENGTH_LONG).show()
                    e.printStackTrace()
                }
            }
        }
    }

    private fun createProduct(product: Product) {
        lifecycleScope.launch {
            try {
                val createdProduct =
RetrofitClient.instance.createProduct(product)
                withContext(Dispatchers.Main) {
                    Toast.makeText(this@MainActivity, "Created product:
${createdProduct.name}", Toast.LENGTH_SHORT).show()
                }
            } catch (e: Exception) {
                withContext(Dispatchers.Main) {
                    Toast.makeText(this@MainActivity, "Error creating product:
${e.message}", Toast.LENGTH_LONG).show()
                    e.printStackTrace()
                }
            }
        }
    }
}
```

JSON Parsing

As covered in Kotlin basics, `Gson` (used by `converter-gson` in Retrofit) or `Jackson` are common libraries for mapping JSON to Java/Kotlin objects. Kotlin's `kotlinx.serialization` is also a powerful, idiomatic choice.

Using Social media

Integrating social media login usually involves using SDKs provided by the social media platforms.

Facebook Login via Facebook SDK

- **Setup:** Add Facebook SDK dependency, configure `AndroidManifest.xml` (app ID, client token), create `strings.xml` entries.
- **Login Button:** Use `com.facebook.login.widget.LoginButton` or manually trigger `LoginManager`.
- **Callback:** Implement `FacebookCallback<LoginResult>` to handle login success/failure.
- **Permissions:** Request `public_profile` and `email` for basic info.

Google Login via Firebase

- **Firebase Integration:** Add Firebase to your Android project.
- **Authentication Service:** Enable Google Sign-In in Firebase Console.

- **Dependencies:** `com.google.firebase:firebase-auth-ktx`, `com.google.android.gms:play-services-auth`.
- **`GoogleSignInClient`:** Create a Google Sign-In client.
- **`ActivityResultLauncher`:** Launch Google Sign-In UI for user.
- **`FirebaseAuth.signInWithCredential()`:** Exchange Google ID token for Firebase credential.

Twitter Login via Firebase

- **Firebase Integration:** Similar to Google Login, enable Twitter Sign-In in Firebase Console.
- **Twitter Developer Account:** Create a Twitter Developer account and app to get API key/secret.
- **Dependencies:** `com.google.firebase:firebase-auth-ktx`.
- **Manual OAuth:** Often requires a more manual OAuth flow or specific Twitter SDK before integrating with Firebase.

Push Notifications

Push notifications (or simply "notifications" in Android context) are messages sent from a server to a user's device.

- **Purpose:** To inform users about events, updates, or promotions, even when the app is not actively running.
- **FCM (Firebase Cloud Messaging):** Google's primary solution for sending push notifications to Android (and iOS/web) devices.
- **Key Components:**
  - **FCM Server:** Your backend server or a third-party service uses FCM to send messages.
  - **FCM Client (App):** Your Android app receives messages.
  - **Registration Token:** Each device registers with FCM and gets a unique token. Your server uses this token to target specific devices.

Steps:

1. **Firebase Project Setup:** Create a Firebase project and add your Android app to it (download `google-services.json`).
2. **Dependencies:** `com.google.firebase:firebase-messaging-ktx`.
3. **Service (`FirebaseMessagingService`):** Create a service that extends `FirebaseMessagingService` to handle incoming messages:
   - `onNewToken(token: String)`: Called when a new FCM registration token is generated (send this to your server).
   - `onMessageReceived(remoteMessage: RemoteMessage)`: Called when an FCM message is received while the app is in the foreground. If the app is in the background/killed, FCM automatically handles display notifications (unless it's a data message).
4. **Display Notification:** Inside `onMessageReceived`, build and display a notification using `NotificationCompat.Builder` (as covered previously).

Example (`MyFirebaseMessagingService.kt`):

```
// build.gradle (app-level)
implementation 'com.google.firebase:firebase-messaging-ktx'
implementation platform('com.google.firebase:firebase-bom:32.4.0') // BOM for
consistent versions

// MyFirebaseMessagingService.kt
import com.google.firebase.messaging.FirebaseMessagingService
import com.google.firebase.messaging.RemoteMessage
import android.app.NotificationManager
import android.app.NotificationChannel
import android.os.Build
import androidx.core.app.NotificationCompat

class MyFirebaseMessagingService : FirebaseMessagingService() {

    override fun onNewToken(token: String) {
        super.onNewToken(token)
        // Send this token to your app server.
        println("FCM Token: $token")
    }

    override fun onMessageReceived(remoteMessage: RemoteMessage) {
        super.onMessageReceived(remoteMessage)

        remoteMessage.notification?.let { notification ->
            val title = notification.title
            val body = notification.body
            sendNotification(title, body)
        }

        remoteMessage.data.isNotEmpty().let {
            // Handle data messages (key-value pairs)
            println("Data payload: ${remoteMessage.data}")
        }
    }

    private fun sendNotification(title: String?, body: String?) {
        val channelId = "default_notification_channel_id"
        val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE)
as NotificationManager

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            val channel = NotificationChannel(channelId, "Default Channel",
NotificationManager.IMPORTANCE_DEFAULT)
            notificationManager.createNotificationChannel(channel)
        }

        val notificationBuilder = NotificationCompat.Builder(this, channelId)
            .setSmallIcon(R.drawable.ic_notification) // Use a small icon
            .setContentTitle(title)
            .setContentText(body)
            .setPriority(NotificationCompat.PRIORITY_DEFAULT)
            .setAutoCancel(true)
        notificationManager.notify(0, notificationBuilder.build())
    }
}
```

```xml
<!-- AndroidManifest.xml (inside <application> tag) -->
<service
    android:name=".MyFirebaseMessagingService"
    android:exported="false">
    <intent-filter>
        <action android:name="com.google.firebase.MESSAGING_EVENT" />
    </intent-filter>
</service>
```

Android Design Patterns

Architectural patterns help structure your app, making it more robust, testable, and maintainable.

MVP (Model-View-Presenter)

- **Model:** Represents the data and business logic. Independent of the UI.
- **View:** Passive interface that displays data and routes user input to the Presenter. (e.g., Activity, Fragment).
- **Presenter:** Acts as a middleman. It contains the presentation logic, interacts with the Model to retrieve/manipulate data, and updates the View. It has no knowledge of Android-specific UI components.
- **Benefits:** High testability (Presenter can be unit tested without Android dependencies), strong separation of concerns.

MVVM (Model-View-ViewModel)

- **Model:** Represents data and business logic.
- **View:** Displays UI and sends user events to the ViewModel. It observes changes in the ViewModel. (e.g., Activity, Fragment).
- **ViewModel:** Exposes data streams from the Model to the View. It holds the UI-related data that can survive configuration changes (e.g., screen rotation). It handles UI logic (e.g., formatting data for display). It communicates with the Model. **It should not hold references to Views or Context to avoid memory leaks.**
- **Benefits:** Excellent for testability (ViewModel is framework-agnostic), LiveData/Flow simplifies UI updates, good for handling configuration changes.
- **Android Components for MVVM:**
  - **`ViewModel` (from Android Architecture Components):** A class that stores and manages UI-related data in a lifecycle-conscious way.
  - **`LiveData` / `Flow`:** Observable data holders that are lifecycle-aware.
  - **Data Binding Library:** Allows declarative binding of UI components in layouts to data sources in your app.

Example (MVVM - Conceptual):

```kotlin
// 1. Model (Data + Repository)
data class User(val id: String, val name: String)
interface UserRepository {
    suspend fun getUserById(userId: String): User
}
class NetworkUserRepository : UserRepository { /* ... */ }

// 2. ViewModel
class UserViewModel(private val userRepository: UserRepository) : ViewModel() {
    val userName: LiveData<String> = liveData {
        // Fetch data from repository on a background thread
        val user = userRepository.getUserById("123")
        emit(user.name) // Emit value to LiveData
    }

    // Other UI-related logic
    fun refreshUser() { /* ... */ }
}

// 3. View (Activity/Fragment)
class UserProfileActivity : AppCompatActivity() {
    private val viewModel: UserViewModel by viewModels {
        // ViewModelFactory (if custom dependencies)
        object : ViewModelProvider.Factory {
            override fun <T : ViewModel> create(modelClass: Class<T>): T {
                return UserViewModel(NetworkUserRepository()) as T
            }
        }
    }

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

        // Observe LiveData from ViewModel
        viewModel.userName.observe(this) { name ->
            findViewById<TextView>(R.id.userNameTextView).text = name
        }
        // User interactions (e.g., button click)
        findViewById<Button>(R.id.refreshButton).setOnClickListener {
            viewModel.refreshUser()
        }
    }
}
```

Intro to app uploading process on Google Play.

Publishing your Android app involves preparing it for production and submitting it to the Google Play Store.

Publish App to Play Store

1. **Google Play Developer Account:**

    - Register for a Google Play Developer account (play.google.com/console). This requires a one-time registration fee.

2. **Prepare Your App for Release:**

    - **Build Type:** Ensure you are building a `release` build.
    - **Signing:** Sign your app with a unique **release key**.
        - **Generate Keystore:** Use Java's `keytool` utility to generate a new keystore file (`.jks` or `.keystore`). Keep this file and its password, alias, and key password safe; you'll need it for all future updates.
        - **Sign AAB/APK:** Configure your `app/build.gradle` to use your release keystore for signing the `release` build type.
    - **App Bundle (AAB - Recommended):** Build an Android App Bundle (`.aab`) instead of an APK. AABs are smaller and optimized by Google Play to deliver optimized APKs to users based on their device configurations.
        - `Build > Generate Signed Bundle / APK...` in Android Studio. Select "Android App Bundle."
    - **Version Codes and Names:** Increment `versionCode` (always an integer) and update `versionName` (public-facing string) in `app/build.gradle` for each new release.
    - **ProGuard/R8:** Ensure code shrinking (obfuscation, optimization, resource shrinking) is enabled for release builds (`minifyEnabled true` in `buildTypes.release`). Configure `proguard-rules.pro` if needed to prevent essential code from being removed.

3. **Google Play Console Setup:**

    - **Create New App:** In the Play Console, click "Create app" or "All applications" -> "Create application".
    - **App Details:** Fill in the basic app information (name, default language, app/game, free/paid).
    - **Dashboard:** Navigate to your app's dashboard.

4. **Release Management:**

    - **Testing Tracks (Internal, Closed, Open):**
        - **Internal Testing:** For quick internal checks. Upload your AAB/APK and quickly distribute to a small group of testers via email.
        - **Closed Testing:** For a larger beta test group, usually chosen by you.
        - **Open Testing:** Open to anyone on Google Play, providing a public beta.
    - **Production Track:** This is where you release your app to the general public.

5. **Store Listing Information:**

    - **App Name, Short Description, Full Description:** Compelling text for your app page.
    - **Screenshots:** Minimum 2 screenshots for phone, tablet screenshots are highly recommended. High-quality and visually appealing.
    - **Feature Graphic (Banner):** 1024x500 px.
    - **App Icon:** 512x512 px (PNG).

- **Category:** Select appropriate app category.
- **Contact Details, Privacy Policy URL:** Essential for user trust and compliance.
- **Release Notes/What's New:** Describe changes in your new version.

6. **App Content and Targeting:**

    - **Content Rating:** Complete the content rating questionnaire.
    - **Target Audience & Content:** Declare age groups and if your app contains ads.
    - **Privacy Policy:** Upload or link to your app's privacy policy.

7. **Rollout Release:**

    - Once all details are complete and your AAB/APK is uploaded to a track (e.g., Production), you can review and "Start rollout to Production."
    - **Phased Rollout:** You can choose a phased rollout (e.g., 1%, then 5%, 10%, 50%, 100%) to gradually release updates and monitor for issues.

Post-Publishing:

- **Monitoring:** Monitor your app's performance in the Play Console (crashes, ANRs, vitals).
- **Updates:** To release a new version, simply increment `versionCode`, build a new AAB/APK, and create a new release in the Play Console.

Remember to follow Google Play's Developer Program Policies carefully to avoid issues during the review process.