**✅ Section 21: Android Components – Services, Broadcast Receivers, Content Providers**

**🔷 1. Android Components: Overview**

Android apps are built using **four main components**, each playing a critical role in app behavior:

| **Component** | **Role** |
| --- | --- |
| **Activity** | UI screen; user interaction point (e.g., home screen, settings). |
| **Service** | Background operations **without UI**. |
| **BroadcastReceiver** | Responds to system-wide or app-wide broadcast events. |
| **ContentProvider** | Shares structured data between apps/components (e.g., SQLite DB). |

**🔷 2. Services in Android**

**🔸 What is a Service?**

A **Service** runs background tasks **without a UI** and can survive even after the component (e.g., Activity) that started it is destroyed.

**🔸 Types of Services**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Foreground** | User-visible via notification; less likely to be killed by system. | Music player, download manager. |
| **Background** | Silent tasks; system may kill under resource pressure. | Sync notes periodically. |
| **Bound** | Allows client (e.g., Activity) to bind and interact directly. | Bind activity to control music. |

**🔸 Service Lifecycles**

**🧩 Unbound Service:**

* Lifecycle:  
  onCreate() → onStartCommand() → onDestroy()
* Starts using startService()
* Stops using stopService()
* Can run independently of the activity

**🧩 Bound Service:**

* Lifecycle:  
  onCreate() → onBind() → onUnbind()/onRebind() → onDestroy()
* Starts using bindService()
* Stops when all clients unbind

**🔸 Key Return Flags in onStartCommand():**

| **Return Type** | **Behavior** |
| --- | --- |
| START\_STICKY | Restarts service with null intent if killed. Good for continuous tasks. |
| START\_NOT\_STICKY | Does not restart automatically. Good for short-lived tasks. |
| START\_REDELIVER\_INTENT | Restarts and redelivers intent. |

**🔧 Implementation: Music Player Using Unbound Service**

**✅ Step-by-Step Implementation**

**📌 Step 1: Service Class – MyCustomService.java**

public class MyCustomService extends Service {

private MediaPlayer player;

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

player = MediaPlayer.create(this, Settings.System.DEFAULT\_RINGTONE\_URI);

player.setLooping(true);

player.start();

return START\_STICKY;

}

@Override

public void onDestroy() {

if (player != null) {

player.stop();

player.release(); // Avoid memory leaks

}

super.onDestroy();

}

@Override

public IBinder onBind(Intent intent) {

return null; // Not a bound service

}

}

**📌 Step 2: Start/Stop Service from MainActivity.java**

startBtn.setOnClickListener(v -> {

Intent intent = new Intent(getApplicationContext(), MyCustomService.class);

startService(intent);

});

stopBtn.setOnClickListener(v -> {

Intent intent = new Intent(getApplicationContext(), MyCustomService.class);

stopService(intent);

});

**📌 Step 3: Register Service in AndroidManifest.xml**

<service

android:name=".MyCustomService"

android:enabled="true"

android:exported="false" />

📝 android:exported="false" means it’s private to your app.

**🔷 3. Broadcast Receivers**

**🔸 What is a BroadcastReceiver?**

A **BroadcastReceiver** listens for **system or custom broadcast intents**, allowing your app to respond to events like:

* Airplane mode toggled
* Battery level low
* SMS received
* Connectivity changes

🧠 It **has no UI** and runs for a short time (max 10s).

**🔸 Static vs Dynamic Registration**

| **Type** | **When Used** | **Where Registered** |
| --- | --- | --- |
| **Static** | Legacy devices (API < 26) | In AndroidManifest.xml |
| **Dynamic** | Required for API 26+ (modern) | In code at runtime (registerReceiver()) |

**✅ Implementation: Airplane Mode Detector**

**📌 Step 1: Create AirplaneModeReceiver.java**

public class AirplaneModeReceiver extends BroadcastReceiver {

@Override

public void onReceive(Context context, Intent intent) {

if (Intent.ACTION\_AIRPLANE\_MODE\_CHANGED.equals(intent.getAction())) {

boolean isOn = intent.getBooleanExtra("state", false);

String msg = isOn ? "Airplane Mode ON" : "Airplane Mode OFF";

Toast.makeText(context, msg, Toast.LENGTH\_SHORT).show();

}

}

}

**📌 Step 2: Register Dynamically in MainActivity.java**

private AirplaneModeReceiver receiver;

@Override

protected void onResume() {

super.onResume();

IntentFilter filter = new IntentFilter(Intent.ACTION\_AIRPLANE\_MODE\_CHANGED);

receiver = new AirplaneModeReceiver();

registerReceiver(receiver, filter); // Dynamic registration

}

@Override

protected void onPause() {

super.onPause();

if (receiver != null) {

unregisterReceiver(receiver); // Avoid leaks

}

}

**🔷 4. Content Providers (Intro Only)**

Allows structured data to be shared between apps using URIs and ContentResolver.

**📦 Common Use Cases:**

* Sharing notes, contacts, images
* Accessing system-wide data like SMS, calendar
* Backend SQLite access via ContentProvider

🧠 Instructor mentioned use-case examples but didn't implement code here.

**✅ Tools, APIs & Classes Used**

| **Tool/API/Class** | **Purpose** |
| --- | --- |
| Service | Background task base class |
| MediaPlayer | Audio/video playback |
| BroadcastReceiver | Event listener for system-wide broadcasts |
| IntentFilter | Specifies broadcast types (e.g., airplane mode) |
| Settings.System.DEFAULT\_RINGTONE\_URI | System's default ringtone used in MediaPlayer |
| startService(), stopService() | Manage unbound service lifecycle |
| registerReceiver() / unregisterReceiver() | Manage dynamic receiver lifecycle |
| onStartCommand(), onDestroy() | Service lifecycle callbacks |
| getBooleanExtra("state", false) | Get airplane mode status from broadcast intent |

**🛠️ Best Practices & Modern Approaches**

| **Area** | **Best Practice** |
| --- | --- |
| **Service Lifecycle** | Stop service in onDestroy(), release all resources like MediaPlayer |
| **Long Tasks** | Use startForeground() for visible ongoing work (Android 8+) |
| **Dynamic Receivers** | Always unregister in onPause() or onStop() to avoid memory leaks |
| **Use Cases** | For background sync or retries, use WorkManager (modern, battery optimized) |
| **Broadcast Events** | Avoid long-running code in onReceive() (max 10s); use WorkManager if needed |
| **Security** | From Android 12+, android:exported is mandatory for components with filters |
| **Permissions** | Use <uses-permission> where needed (e.g., network access, boot complete) |

**🔁 Part B: Crucial Topics Not Covered but Needed for Mastery**

| **Topic** | **Why You Should Learn It** |
| --- | --- |
| **Bound Services with Binder** | Enables two-way communication between Activity and Service |
| **Foreground Services** | Required for persistent tasks like downloading, navigation (needs notifications) |
| **ContentProvider Implementation** | Learn query(), insert(), update(), delete() + URIs |
| **JobIntentService/WorkManager** | Replaces background IntentService; handles deferrable + guaranteed tasks |
| **Testing Services & Receivers** | Use ServiceTestRule, BroadcastReceiverTestCase, mock Intents |
| **RxJava/LiveData & Services** | Stream service updates to UI in MVVM pattern |
| **LocalBroadcastManager (deprecated)** | Previously used for app-local events; now prefer Kotlin Flow or SharedFlow |
| **Notification Channel** | Required for foreground services in Android 8.0+ |

**📌 Final Checklist for Future Project Building**

✅ Use **Unbound Services** for background jobs  
✅ Use **Bound Services** for Activity → Service interaction  
✅ Use **Foreground Services** for persistent user-visible tasks  
✅ Prefer **WorkManager** for battery-efficient background jobs  
✅ Always **unregister** dynamic receivers  
✅ Dynamically register receivers from **API 26+**  
✅ Mention all components in AndroidManifest.xml  
✅ **Release MediaPlayer** or other heavy objects  
✅ Learn **ContentProvider** for sharing local app data  
✅ Apply **permissions and security policies** properly

**Instructor’s Advice:**  
*“Services and Broadcast Receivers enable background operations and event-driven workflows. Mastering lifecycle management prevents resource leaks and makes your app robust and battery-friendly.”*

🚨 **Common Pitfalls to Avoid**

