Module 4

Mobile Application Development using Android

Services, Broadcast Receivers & System Permissions

1. Overview of Services

1.1 What is a Service?

A Service is an Android component that performs long-running operations in the background without providing a user interface. Unlike Activities, Services don't have a UI and continue running even when the user switches to another application.

Key Characteristics:

- Runs in the background
- No user interface
- Can be started, bound, or both
- Runs on the main thread by default
- Can be used for network operations, file I/O, music playback, etc.

1.2 Service Types

Service Type	Description	Use Case
Started Service	Performs a single operation and stops itself	Downloading a file, uploading data
Bound Service	Allows other components to bind and interact with it	Music player, exposing APIs to other apps
Foreground Service	User-visible service with a persistent notification	Music playback, fitness tracking, navigation
IntentService	Handles asynchronous requests on a worker thread (Deprecated API 30+)	Processing queue of tasks

1.3 Service Lifecycle Methods

Method	Description	When Called
onCreate()	Initialize service	Service is created
onStartCommand()	Process a start request	startService() called
onBind()	Allow binding	bindService() called
onUnbind()	Release binding	All clients unbind
onDestroy()	Clean up resources	Service is stopped

2. Creating a Started Service

2.1 Basic Service Implementation (Java)

```
public class MyService extends Service {
    private static final String TAG = "MyService";

    @Override
    public void onCreate() {
        super.onCreate();
        Log.d(TAG, "Service Created");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.d(TAG, "Service Started with ID: " + startId);
        if (intent != null) {
            String action = intent.getStringExtra("ACTION");
        }
        return START_STICKY; // START_NOT_STICKY | START_REDELIVER_INTENT
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d(TAG, "Service Destroyed");
    }

    @Override
    public IBinder onBind(Intent intent) { return null; }
}
```

Return Flags:

- START_STICKY — Restart if killed, don't redeliver intent
- START_NOT_STICKY — Don't restart if killed
- START_REDELIVER_INTENT — Restart and redeliver last intent

2.2 Kotlin Implementation

```
class MyService : Service() {
    private val TAG = "MyService"
```

```
    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 with ID: $startId")
        intent?.let { val action = it.getStringExtra("ACTION") }
        return START_STICKY
    }

    override fun onDestroy() { super.onDestroy() }
    override fun onBind(intent: Intent?): IBinder? = null
}
```

2.3 Registering in Manifest

```
<service
    android:name=".MyService"
    android:enabled="true"
    android:exported="false" />

<!-- Foreground Service Permission -->
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
```

2.4 Starting and Stopping the Service

```
// Start service
Intent intent = new Intent(this, MyService.class);
intent.putExtra("ACTION", "DOWNLOAD");
startService(intent);

// Stop service
Intent intent = new Intent(this, MyService.class);
stopService(intent);
```

3. Foreground Service

3.1 Overview

A Foreground Service is a service that the user is aware of, displayed with a persistent notification in the notification drawer.

Requirements:

- Must display a notification
- Runs even when app is in background
- User can see it's running
- System prioritizes it (less likely to be killed)

Common Use Cases:

- Music playback
- Fitness tracking
- Navigation
- File download/upload
- Voice recording

3.2 Creating a Foreground Service

Step 1: Add Permission to Manifest

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

Step 2: Create Notification Channel (Android 8.0+)

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    NotificationChannel channel = new NotificationChannel(
        CHANNEL_ID, "Foreground Service", NotificationManager.IMPORTANCE_LOW);
    NotificationManager manager = getSystemService(NotificationManager.class);
    manager.createNotificationChannel(channel);
}
```

Step 3: Start as Foreground

```
Notification notification = new NotificationCompat.Builder(this, CHANNEL_ID)
    .setContentTitle("Foreground Service")
    .setContentText("Service is running...")
    .setSmallIcon(R.drawable.ic_foreground)
    .setPriority(NotificationCompat.PRIORITY_LOW)
    .build();

startForeground(NOTIFICATION_ID, notification);
```

3.3 Starting from Activity

```
Intent intent = new Intent(this, ForegroundService.class);
intent.putExtra("ACTION", "START");

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    startForegroundService(intent);
} else {
    startService(intent);
}
```

4. Bound Service

4.1 Overview

A Bound Service allows other components (Activities, Fragments) to bind to it and interact with it through an interface.

Key Points:

- Multiple components can bind to the service
- Service lives as long as there are clients bound
- Clients can call methods on the service
- Clients must unbind when done

4.2 Binder Class

```
public class BoundService extends Service {
    private final IBinder binder = new LocalBinder();

    public class LocalBinder extends Binder {
        public BoundService getService() {
            return BoundService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return binder;
    }

    // Methods exposed to clients
    public int getCounter() { return counter; }
    public boolean isServiceRunning() { return isRunning; }
}
```

4.3 Binding from Activity

```
private ServiceConnection connection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        BoundService.LocalBinder binder = (BoundService.LocalBinder) service;
        boundService = binder.getService();
        isBound = true;
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
        isBound = false;
    }
};

// Bind
Intent intent = new Intent(this, BoundService.class);
bindService(intent, connection, Context.BIND_AUTO_CREATE);
```

```
// Unbind (in onDestroy)
if (isBound) { unbindService(connection); isBound = false; }
```

5. Broadcast Receivers

5.1 Overview

A Broadcast Receiver is an Android component that responds to system-wide broadcast announcements.

Common Uses:

- Respond to system events (boot completed, battery low)
- Respond to network changes
- Receive intents from other apps
- Schedule background work

5.2 System Broadcasts Reference

Broadcast Action	Description
ACTION_BOOT_COMPLETED	After system reboot
ACTION_POWER_CONNECTED	Charger connected
ACTION_POWER_DISCONNECTED	Charger disconnected
ACTION_BATTERY_LOW	Low battery warning
ACTION_BATTERY_OKAY	Battery status normal
ACTION_AIRPLANE_MODE_CHANGED	Airplane mode toggled on/off
ACTION_NETWORK_STATE_CHANGED	Wi-Fi/Data connection change
ACTION_SCREEN_ON / ACTION_SCREEN_OFF	Display state changed
ACTION_TIME_CHANGED	System time changed

5.3 Creating a Broadcast Receiver

```
public class MyBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (action == null) return;

        switch (action) {
            case Intent.ACTION_BOOT_COMPLETED:
                // Start service or schedule work
                break;
            case Intent.ACTION_POWER_CONNECTED:
                showNotification(context, "Charger Connected");
                break;
            case Intent.ACTION_BATTERY_LOW:
                showNotification(context, "Battery Low Warning");
                break;
```

```
            case "CUSTOM_ACTION":
                String data = intent.getStringExtra("DATA");
                showNotification(context, "Custom: " + data);
                break;
        }
    }
}
```

5.4 Registration Methods

Method 1: Static Registration (Manifest)

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

<receiver android:name=".MyBroadcastReceiver" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
    <intent-filter>
        <action android:name="CUSTOM_ACTION" />
    </intent-filter>
</receiver>
```

Method 2: Dynamic Registration (Code)

```
receiver = new MyBroadcastReceiver();
IntentFilter filter = new IntentFilter();
filter.addAction(Intent.ACTION_POWER_CONNECTED);
filter.addAction(Intent.ACTION_BATTERY_LOW);
filter.addAction("CUSTOM_ACTION");

// Register (in onStart)
registerReceiver(receiver, filter);

// Unregister (in onStop) - prevents memory leaks!
unregisterReceiver(receiver);
```

5.5 Sending Broadcasts

```
// Normal broadcast
Intent intent = new Intent("CUSTOM_ACTION");
intent.putExtra("DATA", "Hello from activity!");
sendBroadcast(intent);

// Ordered broadcast
sendOrderedBroadcast(intent, null);

// Local broadcast (app-only, more secure)
LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
```

6. System Permissions

6.1 Overview

App permissions help support user privacy by protecting access to restricted data and restricted actions. Android categorizes permissions into different types based on their protection level.

6.2 Permission Categories

Category	Description	Examples
Normal	Minimal risk, automatically granted at install	INTERNET, VIBRATE, SET_ALARM
Dangerous	Access private user data, must request at runtime	CAMERA, LOCATION, READ_CONTACTS
Signature	Same certificate as defining app	Some system-level permissions
Special	Powerful operations requiring extra steps	SYSTEM_ALERT_WINDOW

6.3 Dangerous Permission Groups

Group	Permissions
CALENDAR	READ_CALENDAR, WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS, WRITE_CONTACTS, GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE, CALL_PHONE, READ_CALL_LOG
SENSORS	BODY_SENSORS
SMS	SEND_SMS, RECEIVE_SMS, READ_SMS
STORAGE	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE

6.4 Requesting Runtime Permissions (Java)

```
private static final int PERMISSION_REQUEST_CODE = 100;
private static final String[] REQUIRED_PERMISSIONS = {
    Manifest.permission.CAMERA,
    Manifest.permission.READ_EXTERNAL_STORAGE,
    Manifest.permission.ACCESS_FINE_LOCATION
};

// Check permission
if (checkSelfPermission(Manifest.permission.CAMERA)
```

```
        != PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(this,
        REQUIRED_PERMISSIONS, PERMISSION_REQUEST_CODE);
}

// Handle result
@Override
public void onRequestPermissionsResult(int requestCode,
        String[] permissions, int[] grantResults) {
    if (requestCode == PERMISSION_REQUEST_CODE) {
        boolean allGranted = true;
        for (int result : grantResults) {
            if (result != PackageManager.PERMISSION_GRANTED) {
                allGranted = false; break;
            }
        }
        if (allGranted) proceedWithApp();
        else { Toast.makeText(this, "Permissions required", ...).show(); finish(); }
    }
}
```

6.5 Permission Best Practices

Minimize Permission Requests:

- Request only permissions your app actually needs
- Consider if you can use system APIs without permissions
- Check dependencies for inherited permissions

Request in Context:

- Request permissions when user actually needs the feature
- Don't request at app startup
- Explain why the permission is needed

Handle Denials Gracefully:

- App should degrade gracefully without permissions
- Provide informative feedback when permissions are denied
- Allow users to use alternative features

7. Key Concepts Summary

Concept	Purpose	Key Methods
Service	Background operation	onStartCommand(), stopSelf()
Foreground Service	User-visible background	startForeground(), notification
Bound Service	Client interaction	onBind(), ServiceConnection
Broadcast Receiver	System event response	onReceive(), register/unregister
Runtime Permissions	User-controlled access	checkSelfPermission(), requestPermissions()

8. Interview Questions

- What is a Service and what are its types?
- Explain the difference between a started service and a bound service.
- What is a foreground service and when would you use it?
- Explain the service lifecycle.
- What is a Broadcast Receiver and how is it used?
- What is the difference between static and dynamic broadcast receiver registration?
- Explain the different types of permissions in Android.
- What is the difference between normal and dangerous permissions?
- How do you request runtime permissions in Android?
- What are permission groups and why are they used?

9. Practice Exercises

Exercise 1: Simple Background Service

- Create a service that logs a message every 5 seconds
- Start and stop the service from an activity
- Display the service status

Exercise 2: Foreground Service with Notification

- Create a foreground service with progress notification
- Simulate a file download with progress updates
- Add stop and pause functionality

Exercise 3: Bound Service for Counter

- Create a bound service that maintains a counter

- Bind from activity and display the counter value
- Add methods to increment, decrement, and reset

Exercise 4: Broadcast Receiver for Battery

- Create a broadcast receiver for battery status
- Display battery level when connected/disconnected
- Show low battery warning notification

Exercise 5: Permission Management

- Create an app that requires camera and storage permissions
- Implement permission request flow with rationale
- Handle permission denial gracefully