# Handling Telephone Calls

Android provides Built-in applications for phone calls; in some occasions we may need to make a phone call through our application. This could easily be done by using implicit Intent with appropriate actions. Also, we can use Phone State Listener and Telephony Manager classes, in order to monitor the changes in some telephony states on the device.

This chapter lists down all the simple steps to create an application which can be used to make a Phone Call. You can use Android Intent to make phone call by calling built-in Phone Call functionality of the Android. Following section explains different parts of our Intent object required to make a call.

Intent Object - Action to make Phone Call

You will use **ACTION_CALL** action to trigger built-in phone call functionality available in Android device. Following is simple syntax to create an intent with ACTION_CALL action

```
Intent phoneIntent = new Intent(Intent.ACTION_CALL);
```

You can use **ACTION_DIAL** action instead of ACTION_CALL, in that case you will have option to modify hardcoded phone number before making a call instead of making a direct call.

Intent Object - Data/Type to make Phone Call

To make a phone call at a given number 91-000-000-0000, you need to specify **tel:** as URI using setData() method as follows −

```
phoneIntent.setData(Uri.parse("tel:91-000-000-0000"));
```

The interesting point is that, to make a phone call, you do not need to specify any extra data or data type.

Example

Following example shows you in practical how to use Android Intent to make phone call to the given mobile number.

To experiment with this example, you will need actual Mobile device equipped with latest Android OS, otherwise you will have to struggle with emulator which may not work.

| Step | Description |
|---|---|
| 1 | You will use Android studio IDE to create an Android application and name it as *My Application* under a package *com.example.saira_000.myapplication*. |
| 2 | Modify *src/MainActivity.java* file and add required code to take care of making a call. |

|  |  |
|---|---|
| 3 | Modify layout XML file *res/layout/activity_main.xml* add any GUI component if required. I'm adding a simple button to Call 91-000-000-0000 number |
| 4 | No need to define default string constants.Android studio takes care of default constants. |
| 5 | Modify *AndroidManifest.xml* as shown below |
| 6 | Run the application to launch Android emulator and verify the result of the changes done in the application. |

Following is the content of the modified main activity file **src/MainActivity.java**.

```
package com.example.saira_000.myapplication;

import android.Manifest;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.net.Uri;
import android.os.Bundle;
import android.support.v4.app.ActivityCompat;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {
   private Button button;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      button = (Button) findViewById(R.id.buttonCall);

      button.setOnClickListener(new View.OnClickListener() {
         public void onClick(View arg0) {
            Intent callIntent = new Intent(Intent.ACTION_CALL);
            callIntent.setData(Uri.parse("tel:0377778888"));

            if (ActivityCompat.checkSelfPermission(MainActivity.this,
               Manifest.permission.CALL_PHONE) != PackageManager.PERMISSION_GRANTED) {
                  return;
               }
               startActivity(callIntent);
         }
      });

   }
}
```

Following will be the content of **res/layout/activity_main.xml** file −

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

   <Button
      android:id="@+id/buttonCall"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="call 0377778888" />

</LinearLayout>
```

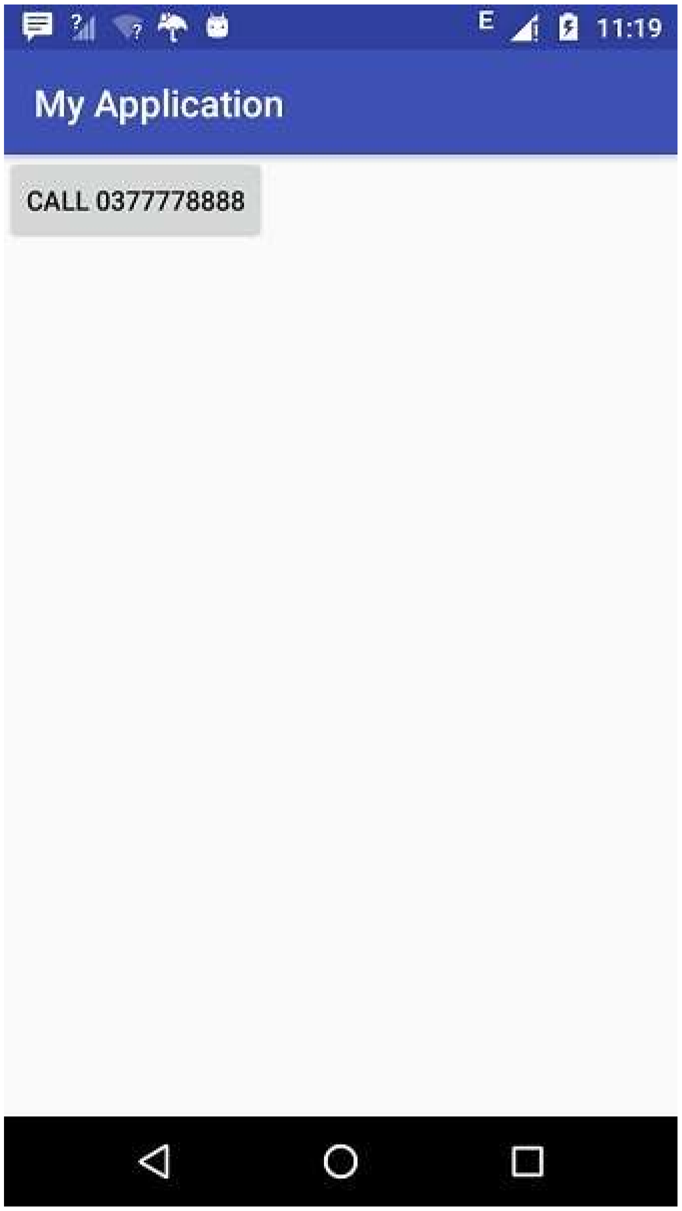
Following will be the content of **res/values/strings.xml** to define two new constants −

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="app_name">My Application</string>
</resources>
```

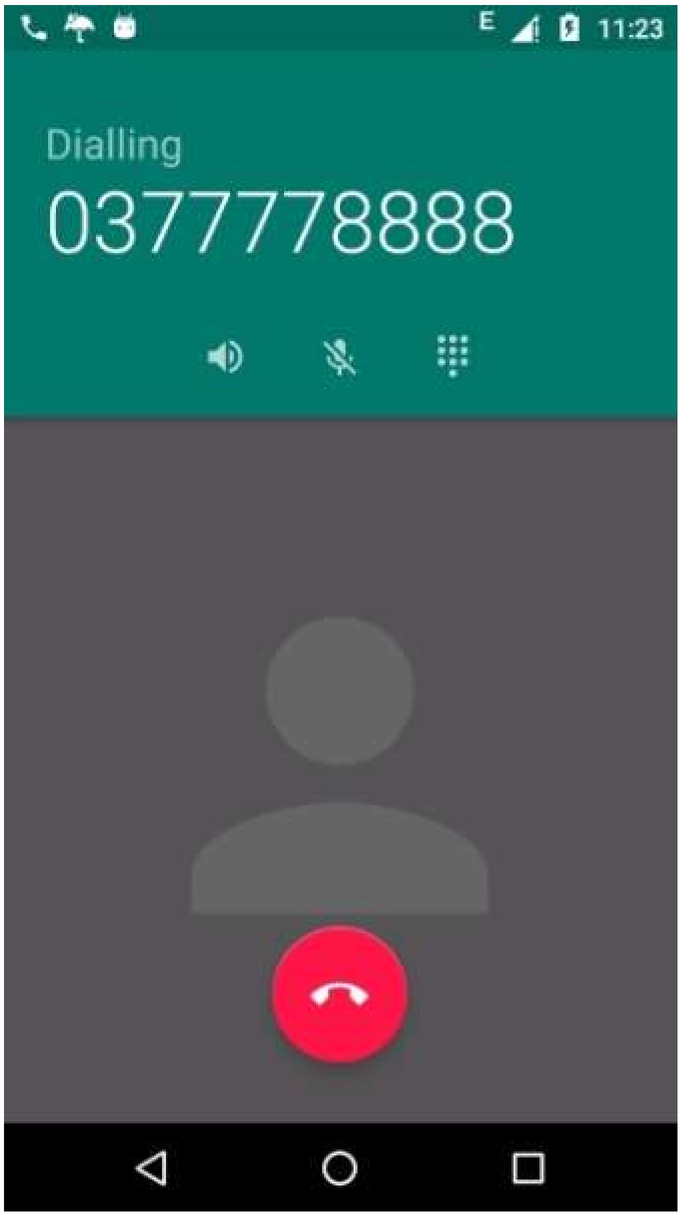
Following is the default content of **AndroidManifest.xml** −

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.saira_000.myapplication" >

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

   <application
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >

      <activity
         android:name="com.example.saira_000.myapplication.MainActivity"
         android:label="@string/app_name" >
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>

    </activity>

  </application>
</manifest>
```

Let's try to run your **My Application** application. I assume you have connected your actual Android Mobile device with your computer. To run the app from Android studio, open one of your project's activity files and click Run ▶ icon from the toolbar.Select your mobile device as an option and then check your mobile device which will display following screen −



Now use **Call** button to make phone call as shown below –



## Build a calling app:

A calling app allows users to receive or place audio or video calls on their device. Calling apps use their own user interface for the calls instead of using the default Phone app interface, as shown in the following screenshot.

An example of a calling app using its own user interface

The Android framework includes the android. Telecom package, which contains classes that help you build a calling app according to the telecom framework. Building your app according to the telecom framework provides the following benefits:

- Your app interoperates correctly with the native telecom subsystem in the device.
- Your app interoperates correctly with other calling apps that also adhere to the framework.
- The framework helps your app manage audio and video routing.
- The framework helps your app determine whether its calls have focus

## Manifest declarations and permissions

In your app manifest, declare that your app uses the MANAGE_OWN_CALLS permission, as shown in the following example:

```
<manifest … >
   <uses-permission android:name="android.permission.MANAGE_OWN_CALLS"/>
</manifest>
```

For more information about declaring app permissions, see Permissions.

You must declare a service that specifies the class that implements the ConnectionService class in your app. The telecom subsystem requires that the service declares the BIND_TELECOM_CONNECTION_SERVICE permission to be able to bind to it. The following example shows how to declare the service in your app manifest:

```
<service android:name="com.example.MyConnectionService"
   android:permission="android.permission.BIND_TELECOM_CONNECTION_SERVICE">
   <intent-filter>
      <action android:name="android.telecom.ConnectionService" />
   </intent-filter>
</service>
```

For more information about declaring app components, including services, see App Components.

## Implement the connection service

Your calling app must provide an implemention of the ConnectionService class that the telecom subsystem can bind to. Your ConnectionService implementation should override the following methods:

onCreateOutgoingConnection(PhoneAccountHandle, ConnectionRequest)

> The telecom subsystem calls this method in response to your app calling placeCall(Uri, Bundle) to create a new outgoing call. Your app returns a new instance of your Connection class implementation (for more information, see Implement the connection) to represent the new outgoing call. You can further customize the outgoing connection by performing the following actions:
>
> - Your app should call the setConnectionProperties(int) method with the PROPERTY_SELF_MANAGED constant as the argument to indicate that the connection originated from a calling app.

- If your app supports putting calls on hold, call the setConnectionCapabilities(int) method and set the argument to the bit mask value of the CAPABILITY_HOLD and CAPABILITY_SUPPORT_HOLD constants.
- To set the name of the caller, use the setCallerDisplayName(String, int) method passing the PRESENTATION_ALLOWED constant as the int parameter to indicate that the caller's name should be shown.
- To ensure that the outgoing call has the appropriate video state, call the setVideoState(int) method of the Connection object and send the value returned by the getVideoState() method of the ConnectionRequest object.

onCreateOutgoingConnectionFailed(PhoneAccountHandle, ConnectionRequest)

The telecom subsystem calls this method when your app calls the placeCall(Uri, Bundle) method and the outgoing call cannot be placed. In response to this situation, your app should inform the user (for example, using an alert box or toast) that the outgoing call could not be placed. Your app may not be able to place a call if there is an ongoing emergency call, or if there is an ongoing call in another app which cannot be put on hold before placing your call.

onCreateIncomingConnection(PhoneAccountHandle, ConnectionRequest)

The telecom subsystem calls this method when your app calls the addNewIncomingCall(PhoneAccountHandle, Bundle) method to inform the system of a new incoming call in your app. Your app returns a new instance of your Connection implementation (for more information, see Implement the connection) to represent the new incoming call. You can further customize the incoming connection by performing the following actions:

- Your app should call the setConnectionProperties(int) method with the PROPERTY_SELF_MANAGED constant as the argument to indicate that the connection originated from a calling app.
- If your app supports putting calls on hold, call the setConnectionCapabilities(int) method and set the argument to the bit mask value of the CAPABILITY_HOLD and CAPABILITY_SUPPORT_HOLD constants.
- To set the name of the caller, use the setCallerDisplayName(String, int) method passing the PRESENTATION_ALLOWED constant as the int parameter to indicate that the caller's name should be shown.
- To specify the phone number or address of the incoming call, use the setAddress(Uri, int) method of the Connection object.

- To ensure that the outgoing call has the appropriate video state, call the setVideoState(int) method of the Connection object and send the value returned by the getVideoState() method of the ConnectionRequest object.

onCreateIncomingConnectionFailed(PhoneAccountHandle, ConnectionRequest)

The telecom subsystem calls this method when your app calls the addNewIncomingCall(PhoneAccountHandle, Bundle) method to inform Telecom of a new incoming call, but the incoming call isn't permitted (for more information, see calling constraints). Your app should silently reject the incoming call, optionally posting a notification to inform the user of the missed call.

**Implement the connection**

Your app should create a subclass of Connection to represent the calls in your app. You should override the following methods in your implementation:

onShowIncomingCallUi()

The telecom subsystem calls this method when you add a new incoming call and your app should show its incoming call UI.

**Note:** When you add a new incoming call, the telecom subsystem shows the incoming call UI on behalf of your app if there is an ongoing call in another app that cannot be put on hold. The user can decide whether to answer the incoming call and drop the current one or reject the incoming call. In these cases, the **onShowIncomingCallUI()** method of your app isn't called.

onCallAudioStateChanged(CallAudioState)

The telecom subsystem calls this method to inform your app that the current audio route or mode has changed. This is called in response to your app changing the audio mode using the setAudioRoute(int) method. This method may also be called if the system changes the audio route (for example, when a Bluetooth headset disconnects).

onHold()

The telecom subsystem calls this method when it wants to put a call on hold. In response to this request, your app should hold the call and then invoke the setOnHold() method to inform the system that the call is being held. The telecom subsystem may call this method when an in-call service, such as Android Auto, that is showing your call wants to relay a user request to put the call on hold. The telecom subsystem also calls this method if the user makes a call active in another app. For more information about in-call services, see InCallService.

onUnhold()

The telecom subsystem calls this method when it wants to resume a call that has been put on hold. Once your app has resumed the call, it should invoke the setActive() method to inform the system that the call is no longer on hold. The telecom subsystem may call this method when an in-call service, such as Android Auto, that is showing your call wants to relay a request to resume the call. For more information about in-call services, see InCallService.

onAnswer()

The telecom subsystem calls this method to inform your app that an incoming call should be answered. Once your app has answered the call, it should invoke the setActive() method to inform the system that the call has been answered. The telecom subsystem may call this method when your app adds a new incoming call and there is already an ongoing call in another app which cannot be put on hold. The telecom subsystem displays the incoming call UI on behalf of your app in these instances. The framework provides an overloaded method that provides support to specify the video state in which to answer the call. For more information, see onAnswer(int).

onReject()

The telecom subsystem calls this method when it wants to reject an incoming call. Once your app has rejected the call, it should call the setDisconnected(DisconnectCause) and specify REJECTED as the parameter. Your app should then call the destroy() method to inform the system that the app has processed the call. The telecom subsystem calls this method when the user has rejected an incoming call from your app.

onDisconnect()

The telecom subsystem calls this method when it wants to disconnect a call. Once the call has ended, your app should call the setDisconnected(DisconnectCause) method and specify LOCAL as the parameter to indicate that a user request caused the call to be disconected. Your app should then call the destroy() method to inform the telecom subsystem that the app has processed the call. The system may call this method when the user has disconnected a call through another in-call service such as Android Auto. The system also calls this method when your call must be disconnected to allow other call to be placed, for example, if the user wants to place an emergency call. For more information about in-call services, see InCallService.

**Handle common calling scenarios**

Making use of the ConnectionService API in your call flow involves interacting with the other classes in the android.telecom package. The following sections describe common calling scenarios and how your app should use the APIs to handle them.

**Answer incoming calls**

The flow to handle incoming calls changes whether there are calls in other apps or not. The reason for the difference in the flows is that the telecom framework must establish some constraints when there are active calls in other apps to ensure a stable environment for all calling apps on the device. For more information, see Calling constraints.

**No active calls in other apps**

To answer incoming calls when there are no active calls in other apps, follow these steps:

1. Your app receives a new incoming call using its usual mechanisms.
2. Use the addNewIncomingCall(PhoneAccountHandle, Bundle) method to inform the telecom subsystem about the new incoming call.
3. The telecom subsystem binds to your app's ConnectionService implementation and requests a new instance of the Connection class representing the new incoming call using the onCreateIncomingConnection(PhoneAccountHandle, ConnectionRequest) method.
4. The telecom subsystem informs your app that it should show its incoming call user interface using the onShowIncomingCallUi() method.
5. Your app shows its incoming UI using a notification with an associated full-screen intent. For more information, see onShowIncomingCallUi().
6. Call the setActive() method if the user accepts the incoming call, or setDisconnected(DisconnectCause) specifying REJECTED as the parameter followed by a call to the destroy() method if the user rejects the incoming call.

**Active calls in other apps which cannot be put on hold**

To answer incoming calls when there are active calls in other apps which can't be put on hold, follow these steps:

1. Your app receives a new incoming call using its usual mechanisms.
2. Use the addNewIncomingCall(PhoneAccountHandle, Bundle) method to inform the telecom subsystem about the new incoming call.

3. The telecom subsystem binds to your app's ConnectionService implementation and requests a new instance of the Connection object representing the new incoming call using the onCreateIncomingConnection(PhoneAccountHandle, ConnectionRequest) method.
4. The telecom subsystem displays the incoming call UI for your incoming call.
5. If the user accepts the call, the telecom subsystem calls the onAnswer() method. You should call the setActive() method to indicate to the telecom subsystem that the call is now connected.
6. If the user rejects the call, the telecom subsystem calls the onReject() method. You should call the setDisconnected(DisconnectCause) method specifying REJECTED as the parameter followed by a call to the destroy() method.

**Place outgoing calls**

The flow for placing an outgoing call involves handling the possibility that the call cannot be placed because of constraints imposed by the telecom framework. For more information, see Calling constraints.

To place an outgoing call, follow these steps:

1. The user initiates an outgoing call within your app.
2. Use the placeCall(Uri, Bundle) method to inform the telecom subsystem about the new outgoing call. Take the following considerations for the method parameters:
   - The Uri parameter represents the address where the call is being placed to. For regular phone numbers, use the tel: URI scheme.
   - The Bundle parameter allows you to provide information about your calling app by adding the PhoneAccountHandle object of your app to the EXTRA_PHONE_ACCOUNT_HANDLE extra. Your app must provide the PhoneAccountHandle object to every outgoing call.
   - The Bundle parameter also allows you to specify if the outgoing call includes video by specifying the STATE_BIDIRECTIONAL value in the EXTRA_START_CALL_WITH_VIDEO_STATE extra. Consider that by default, the telecom subsystem routes video calls to the speakerphone.
3. The telecom subsystem binds to your app's ConnectionService implementation.
4. If your app isn't able to place an outgoing call, the telecom subsystem calls the onCreateOutgoingConnectionFailed(PhoneAccountHandle, ConnectionRequest) method to inform your app that the call cannot be placed at the current time. Your app should inform the user that the call cannot be placed.

5. If your app is able to place the outgoing call, the telecom subsystem calls the onCreateOutgoingConnection(PhoneAccountHandle, ConnectionRequest) method. Your app should return an instance of your Connection class to represent the new outgoing call. For more information about the properties that you should set in the connection, see Implement the connection service.
6. When the outgoing call has been connected, call the setActive() method to inform the telecom subsystem that the call is active.

**End a call**

To end a call, follow these steps:

1. Call the setDisconnected(DisconnectCause) sending LOCAL as the parameter if the user terminated the call, or send REMOTE as the parameter if the other party terminated the call.
2. Call the destroy() method.

**Calling constraints**

To ensure a consistent and simple calling experience for your users, the telecom framework enforces some constraints for managing calls on the device. For example, consider that the user has installed two calling apps which implement the self-managed ConnectionService API, FooTalk and BarTalk. In this case, the following constraints apply:

- On devices running on API level 27 or lower, only one app can maintain an ongoing call at any given time. This constraint means that while a user has an ongoing call using the FooTalk app, the BarTalk app can't initiate or receive a new call.

  On devices running on API level 28 or higher, if both FooTalk and BarTalk declare CAPABILITY_SUPPORT_HOLD and CAPABILITY_HOLD permissions, then the user can maintain more than one ongoing call by switching between the apps to initiate or answer another call.
- If the user is engaged in regular managed calls (for example, using the built-in Phone or Dialer app), the user cannot be in calls originated from calling apps. This means that if the user is in a regular call using their mobile carrier, they cannot also be in a FooTalk or BarTalk call concurrently.
- The telecom subsystem disconnects your app's calls if the user dials an emergency call.
- Your app cannot receive or place calls while the user is in an emergency call.
- If there is an ongoing call in the another calling app when your app receives an incoming call, answering the incoming call ends any ongoing calls in the other app. You app should not display its usual incoming call user interface. The telecom framework displays the incoming call user interface and informs the user that answering the new call will end their ongoing call(s). This means if the user is in a FooTalk call and the BarTalk app

receives an incoming call, the telecom framework informs the user that they have a new incoming BarTalk call and that answering the BarTalk call will end their FooTalk call.