66. Making Runtime Permission Requests in Android

In a number of the example projects created in preceding chapters, changes have been made to the AndroidManifest.xml file to request permission for the app to perform a specific task. In a couple of instances, for example, internet access permission has been requested in order to allow the app to download and display web pages. In each case up until this point, the addition of the request to the manifest was all that is required in order for the app to obtain permission from the user to perform the designated task.

There are, however, a number of permissions for which additional steps are required in order for the app to function when running on Android 6.0 or later. The first of these so-called "dangerous" permissions will be encountered in the next chapter. Before reaching that point, however, this chapter will outline the steps involved in requesting such permissions when running on the latest generations of Android.

66.1 Understanding Normal and Dangerous Permissions

Android enforces security by requiring the user to grant permission for an app to perform certain tasks. Prior to the introduction of Android 6, permission was always sought at the point that the app was installed on the device. Figure 66-1, for example, shows a typical screen seeking a variety of permissions during the installation of an app via Google Play.

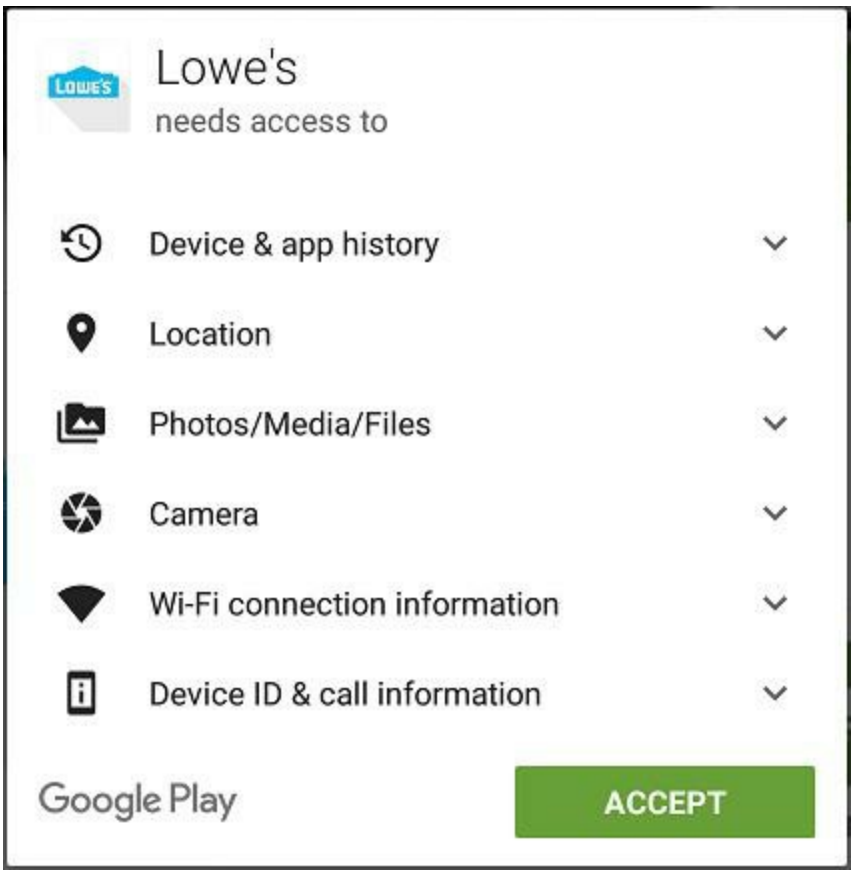

Figure 66-1

For many types of permissions this scenario still applies for apps on Android 6.0 or later. These permissions are referred to as *normal permissions* and are still required to be accepted by the user at

the point of installation. A second type of permission, referred to as *dangerous permissions* must also be declared within the manifest file in the same way as a normal permission, but must also be requested from the user when the application is first launched. When such a request is made, it appears in the form of a dialog box as illustrated in Figure 66-2:

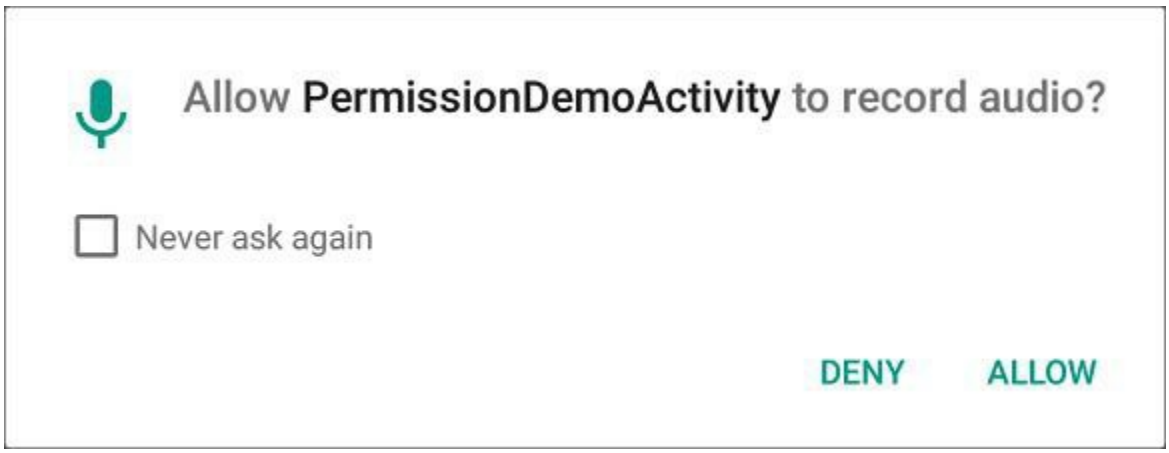

Figure 66-2

The full list of permissions that fall into the dangerous category is contained in Table 66-1:

Permission Group	Permission
Calendar	READ_CALENDAR WRITE_CALENDAR
Camera	CAMERA
Contacts	READ_CONTACTS WRITE_CONTACTS GET_ACCOUNTS
Location	ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION
Microphone	RECORD_AUDIO
	READ_PHONE_STATE CALL_PHONE READ_CALL_LOG

Phone	WRITE_CALL_LOG ADD_VOICEMAIL USE_SIP PROCESS_OUTGOING_CALLS
Sensors	BODY_SENSORS
SMS	SEND_SMS RECEIVE_SMS READ_SMS RECEIVE_WAP_PUSH RECEIVE_MMS
Storage	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE

Table 66-1

66.2 Creating the Permissions Example Project

Create a new project in Android Studio, entering *PermissionDemo* into the Application name field and *com.ebookfrenzy* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 19: Android 4.4 (KitKat). Continue to proceed through the screens, requesting the creation of an Empty Activity named *PermissionDemoActivity* with a corresponding layout named *activity_permission_demo*.

66.3 Checking for a Permission

The Android Support Library contains a number of methods that can be used to seek and manage dangerous permissions within the code of an Android app. These API calls can be made safely regardless of the version of Android on which the app is running, but will only perform meaningful tasks when executed on Android 6.0 or later.

Before an app attempts to make use of a feature that requires approval of a dangerous permission, and regardless of whether or not permission was previously granted, the code must check that the permission has been granted. This can be achieved via a call to the *checkSelfPermission()* method of the ContextCompat class, passing through as arguments a reference to the current activity and the permission being requested. The method will check whether the permission has been previously granted and return an integer value matching *PackageManager.PERMISSION_GRANTED* or *PackageManager.PERMISSION_DENIED*.

Within the *PermissionDemoActivity.java* file of the example project, modify the code to check whether permission has been granted for the app to record audio:

```
package com.ebookfrenzy.permissiondemoactivity;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.Manifest;
import android.content.pm.PackageManager;
import android.support.v4.content.ContextCompat;
import android.util.Log;

public class PermissionDemoActivity extends AppCompatActivity {

    private static String TAG = "PermissionDemo";

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

        int permission = ContextCompat.checkSelfPermission(this,
                Manifest.permission.RECORD_AUDIO);

        if (permission != PackageManager.PERMISSION_GRANTED) {
            Log.i(TAG, "Permission to record denied");
        }
    }
}
```

Run the app on a device or emulator running a version of Android that predates Android 6.0 and check the log cat output within Android Studio. After the app has launched, the output should include the "Permission to record denied" message.

Edit the *AndroidManifest.xml* file (located in the Project tool window under *app -> manifests*) and add a line to request recording permission as follows:

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

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity android:name=".PermissionDemoActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

```
</manifest>
```

Compile and run the app once again and note that this time the permission denial message does not appear. Clearly, everything that needs to be done to request this permission on older versions of Android has been done. Run the app on a device or emulator running Android 6.0 or later, however, and note that even though permission has been added to the manifest file, the check still reports that permission has been denied. This is because Android version 6 or later requires that the app also request dangerous permissions at runtime.

66.4 Requesting Permission at Runtime

A permission request is made via a call to the *requestPermissions()* method of the ActivityCompat class. When this method is called, the permission request is handled asynchronously and a method named *onRequestPermissionsResult()* is called when the task is completed.

The *requestPermissions()* method takes as arguments a reference to the current activity, together with the identifier of the permission being requested and a request code. The request code can be any integer value and will be used to identify which request has triggered the call to the *onRequestPermissionsResult()* method. Modify the *PermissionDemoActivity.java* file to declare a request code and request recording permission in the event that the permission check failed:

```
package com.ebookfrenzy.permissiondemoactivity;

import android.Manifest;
import android.content.pm.PackageManager;
import android.support.v4.content.ContextCompat;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import android.support.v4.app.ActivityCompat;

public class PermissionDemoActivity extends AppCompatActivity {

    private static String TAG = "PermissionDemo";
    private static final int RECORD_REQUEST_CODE = 101;

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

        int permission = ContextCompat.checkSelfPermission(this,
                Manifest.permission.RECORD_AUDIO);

        if (permission != PackageManager.PERMISSION_GRANTED) {
            Log.i(TAG, "Permission to record denied");
            makeRequest();
        }
    }

    protected void makeRequest() {
        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.RECORD_AUDIO},
```

```
                    RECORD_REQUEST_CODE);
        }
    }
```

Next, implement the *onRequestPermissionsResult()* method so that it reads as follows:

```
    @Override
    public void onRequestPermissionsResult(int requestCode,
                                           String permissions[], int[]
    grantResults) {
        switch (requestCode) {
            case RECORD_REQUEST_CODE: {

                if (grantResults.length == 0
                        || grantResults[0] !=
                          PackageManager.PERMISSION_GRANTED) {

                    Log.i(TAG, "Permission has been denied by user");
                } else {
                    Log.i(TAG, "Permission has been granted by user");
                }
                return;
            }
        }
    }
```

Compile and run the app on an Android 6 or later emulator or device and note that a dialog seeking permission to record audio appears as shown in Figure 66-3:

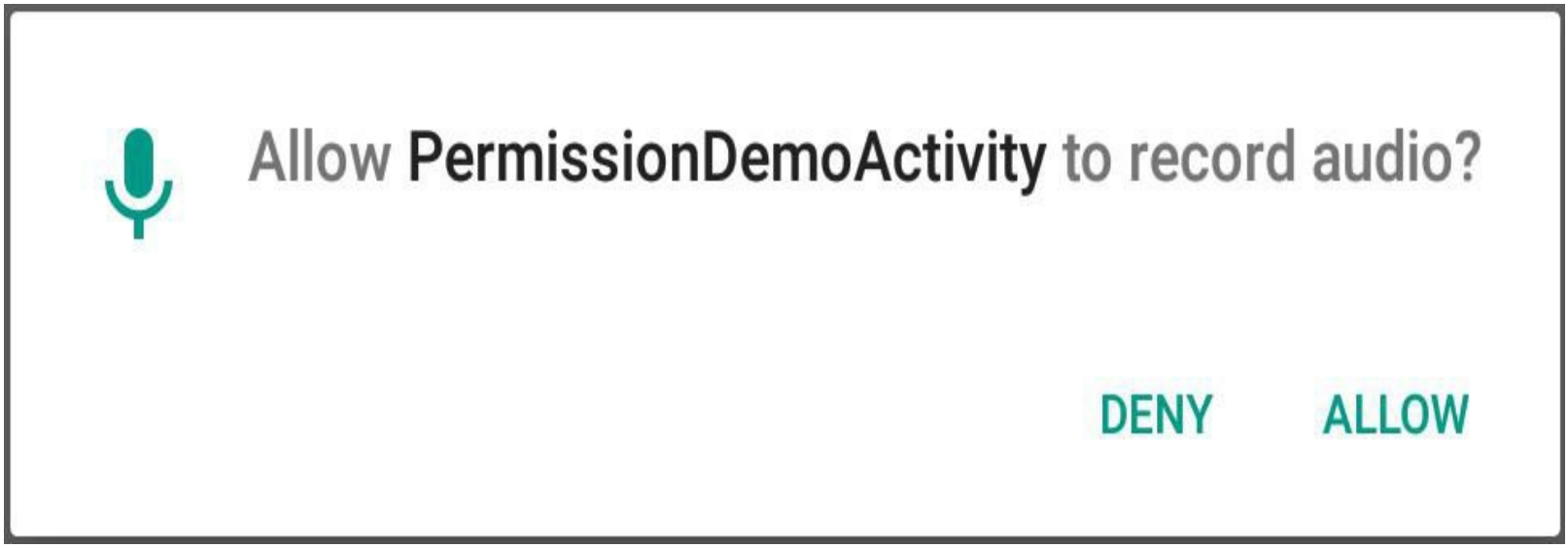

Figure 66-3

Tap the Allow button and check that the "Permission has been granted by user" message appears in the LogCat panel.

Once the user has granted the requested permission, the *checkSelfPermission()* method call will return a PERMISSION_GRANTED result on future app invocations until the user uninstalls and re-installs the app or changes the permissions for the app in Settings.

66.5 Providing a Rationale for the Permission Request

As is evident from Figure 66-3, the user has the option to deny the requested permission. In this case, the app will continue to request the permission each time that it is launched by the user unless the user

selected the "Never ask again" option prior to clicking on the Deny button. Repeated denials by the user may indicate that the user doesn't understand why the permission is required by the app. The user might, therefore, be more likely to grant permission if the reason for the requirements is explained when the request is made. Unfortunately, it is not possible to change the content of the request dialog to include such an explanation.

An explanation is best included in a separate dialog which can be displayed before the request dialog is presented to the user. This raises the question as to when to display this explanation dialog. The Android documentation recommends that an explanation dialog only be shown in the event that the user has previously denied the permission and provides a method to identify when this is the case.

A call to the *shouldShowRequestPermissionRationale()* method of the ActivityCompat class will return a true result if the user has previously denied a request for the specified permission, and a false result if the request has not previously been made. In the case of a true result, the app should display a dialog containing a rationale for needing the permission and, once the dialog has been read and dismissed by the user, the permission request should be repeated.

To add this functionality to the example app, modify the *onCreate()* method so that it reads as follows:

```
.
.
import android.app.AlertDialog;
import android.content.DialogInterface;
.
.
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_permission_demo);

    int permission = ContextCompat.checkSelfPermission(this,
            Manifest.permission.RECORD_AUDIO);

    if (permission != PackageManager.PERMISSION_GRANTED) {
        Log.i(TAG, "Permission to record denied");

        if (ActivityCompat.shouldShowRequestPermissionRationale(this,
                Manifest.permission.RECORD_AUDIO)) {
            AlertDialog.Builder builder =
                    new AlertDialog.Builder(this);
            builder.setMessage("Permission to access the microphone is
required for this app to record audio.")
                    .setTitle("Permission required");

            builder.setPositiveButton("OK",
                    new DialogInterface.OnClickListener() {

                public void onClick(DialogInterface dialog, int id) {
                    Log.i(TAG, "Clicked");
                    makeRequest();
                }
            });
```

```
                AlertDialog dialog = builder.create();
                dialog.show();
            } else {
                makeRequest();
            }
        }
    }
```

The method still checks whether or not the permission has been granted, but now also identifies whether a rationale needs to be displayed. If the user has previously denied the request, a dialog is displayed containing an explanation and an OK button on which a listener is configured to call the *makeRequest()* method when the button is tapped. In the event that the permission request has not previously been made, the code moves directly to seeking permission.

66.6 Testing the Permissions App

On the Android 6 device or emulator session on which testing is being performed, launch the Settings app, select the Apps option and scroll to and select the PermissionDemo app. On the app settings screen, tap the uninstall button to remove the app from the device.

Run the app once again and, when the permission request dialog appears, click on the Deny button. Terminate the app, run it a second time and verify that the rationale dialog appears. Tap the OK button and, when the permission request dialog appears, tap the Allow button.

Return to the Settings app, select the Apps option and select the PermissionDemo app once again from the list. Once the settings for the app are listed, verify that the Permissions section lists the *Microphone* permission:

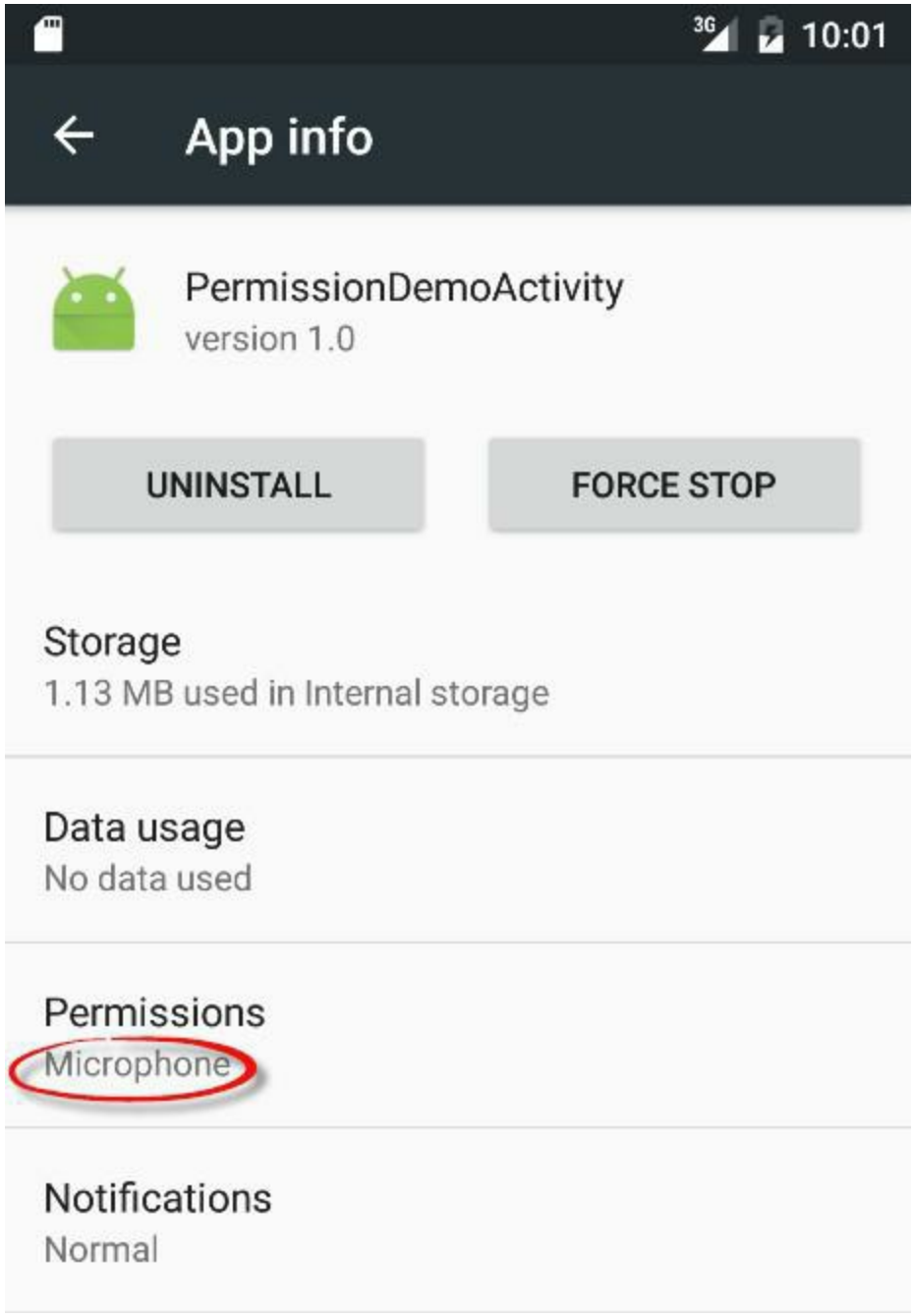

Figure 66-4

66.7 Summary

Prior to the introduction of Android 6.0 the only step necessary for an app to request permission to access certain functionality was to add an appropriate line to the application's manifest file. The user would then be prompted to approve the permission at the point that the app was installed. This is still the case for most permissions, with the exception of a set of permissions that are considered dangerous. Permissions that are considered dangerous usually have the potential to allow an app to violate the user's privacy such as allowing access to the microphone, contacts list or external storage.

As outlined in this chapter, apps based on Android 6 or later must now request dangerous permission approval from the user when the app launches in addition to including the permission request in the manifest file.

67. Android Audio Recording and Playback using MediaPlayer and MediaRecorder

This chapter will provide an overview of the MediaRecorder class and explain the basics of how this class can be used to record audio or video. The use of the MediaPlayer class to play back audio will also be covered. Having covered the basics, an example application will be created to demonstrate these techniques in action. In addition to looking at audio and video handling, this chapter will also touch on the subject of saving files to the SD card.

67.1 Playing Audio

In terms of audio playback, most implementations of Android support AAC LC/LTP, HE-AACv1 (AAC+), HE-AACv2 (enhanced AAC+), AMR-NB, AMR-WB, MP3, MIDI, Ogg Vorbis, and PCM/WAVE formats.

Audio playback can be performed using either the MediaPlayer or the AudioTrack classes. AudioTrack is a more advanced option that uses streaming audio buffers and provides greater control over the audio. The MediaPlayer class, on the other hand, provides an easier programming interface for implementing audio playback and will meet the needs of most audio requirements.

The MediaPlayer class has associated with it a range of methods that can be called by an application to perform certain tasks. A subset of some of the key methods of this class is as follows:

- **create()** – Called to create a new instance of the class, passing through the Uri of the audio to be played.
- **setDataSource()** – Sets the source from which the audio is to play.
- **prepare()** – Instructs the player to prepare to begin playback.
- **start()** – Starts the playback.
- **pause()** – Pauses the playback. Playback may be resumed via a call to the *resume()* method.
- **stop()** – Stops playback.
- **setVolume()** – Takes two floating-point arguments specifying the playback volume for the left and right channels.
- **resume()** – Resumes a previously paused playback session.
- **reset()** – Resets the state of the media player instance. Essentially sets the instance back to the uninitialized state. At a minimum, a reset player will need to have the data source set again and the *prepare()* method called.
- **release()** – To be called when the player instance is no longer needed. This method ensures that any resources held by the player are released.

In a typical implementation, an application will instantiate an instance of the MediaPlayer class, set the source of the audio to be played and then call *prepare()* followed by *start()*. For example:

```
MediaPlayer mediaPlayer = new MediaPlayer();

mediaPlayer.setDataSource("http://www.yourcompany.com/myaudio.mp3");
mediaPlayer.prepare();
mediaPlayer.start();
```

67.2 Recording Audio and Video using the MediaRecorder Class

As with audio playback, recording can be performed using a number of different techniques. One option is to use the MediaRecorder class, which, as with the MediaPlayer class, provides a number of methods that are used to record audio:

- **setAudioSource()** – Specifies the source of the audio to be recorded (typically this will be MediaRecorder.AudioSource.MIC for the device microphone).
- **setVideoSource()** – Specifies the source of the video to be recorded (for example MediaRecorder.VideoSource.CAMERA).
- **setOutputFormat()** – Specifies the format into which the recorded audio or video is to be stored (for example MediaRecorder.OutputFormat.AAC_ADTS).
- **setAudioEncoder()** – Specifies the audio encoder to be used for the recorded audio (for example MediaRecorder.AudioEncoder.AAC).
- **setOutputFile()** – Configures the path to the file into which the recorded audio or video is to be stored.
- **prepare()** – Prepares the MediaRecorder instance to begin recording.
- **start()** - Begins the recording process.
- **stop()** – Stops the recording process. Once a recorder has been stopped, it will need to be completely reconfigured and prepared before being restarted.
- **reset()** – Resets the recorder. The instance will need to be completely reconfigured and prepared before being restarted.
- **release()** – Should be called when the recorder instance is no longer needed. This method ensures all resources held by the instance are released.

A typical implementation using this class will set the source, output and encoding format and output file. Calls will then be made to the *prepare()* and *start()* methods. The *stop()* method will then be called when recording is to end, followed by the *reset()* method. When the application no longer needs the recorder instance, a call to the *release()* method is recommended:

```
MediaRecorder mediaRecorder = new MediaRecorder();

mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.AAC_ADTS);
mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
mediaRecorder.setOutputFile(audioFilePath);

mediaRecorder.prepare();
mediaRecorder.start();
.
.
.
mediaRecorder.stop()
mediaRecorder.reset()
mediaRecorder.release()
```

In order to record audio, the manifest file for the application must include the android.permission.RECORD_AUDIO permission:

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

As outlined in the chapter entitled *Making Runtime Permission Requests in Android 6.0*, access to

the microphone falls into the category of dangerous permissions. To support Android 6, therefore, a specific request for microphone access must also be made when the application launches, the steps for which will be covered later in this chapter.

67.3 About the Example Project

The remainder of this chapter will work through the creation of an example application intended to demonstrate the use of the MediaPlayer and MediaRecorder classes to implement the recording and playback of audio on an Android device.

When developing applications that make use of specific hardware features, the microphone being a case in point, it is important to check the availability of the feature before attempting to access it in the application code. The application created in this chapter will, therefore, also demonstrate the steps involved in detecting the presence of a microphone on the device.

Once completed, this application will provide a very simple interface intended to allow the user to record and playback audio. The recorded audio will need to be stored within an audio file on the device. That being the case, this tutorial will also briefly explore the mechanism for using SD Card storage.

67.4 Creating the AudioApp Project

Create a new project in Android Studio, entering *AudioApp* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *AudioAppActivity* with a corresponding layout resource file named *activity_audio_app*.

67.5 Designing the User Interface

Once the new project has been created, select the *activity_audio_app.xml* file from the Project tool window and with the Layout Editor tool in Design mode, select the “Hello World!” TextView and delete it from the layout.

Drag and drop three Button views onto the layout. The positioning of the buttons is not of paramount importance to this example, though Figure 67-1 shows a suggested layout.

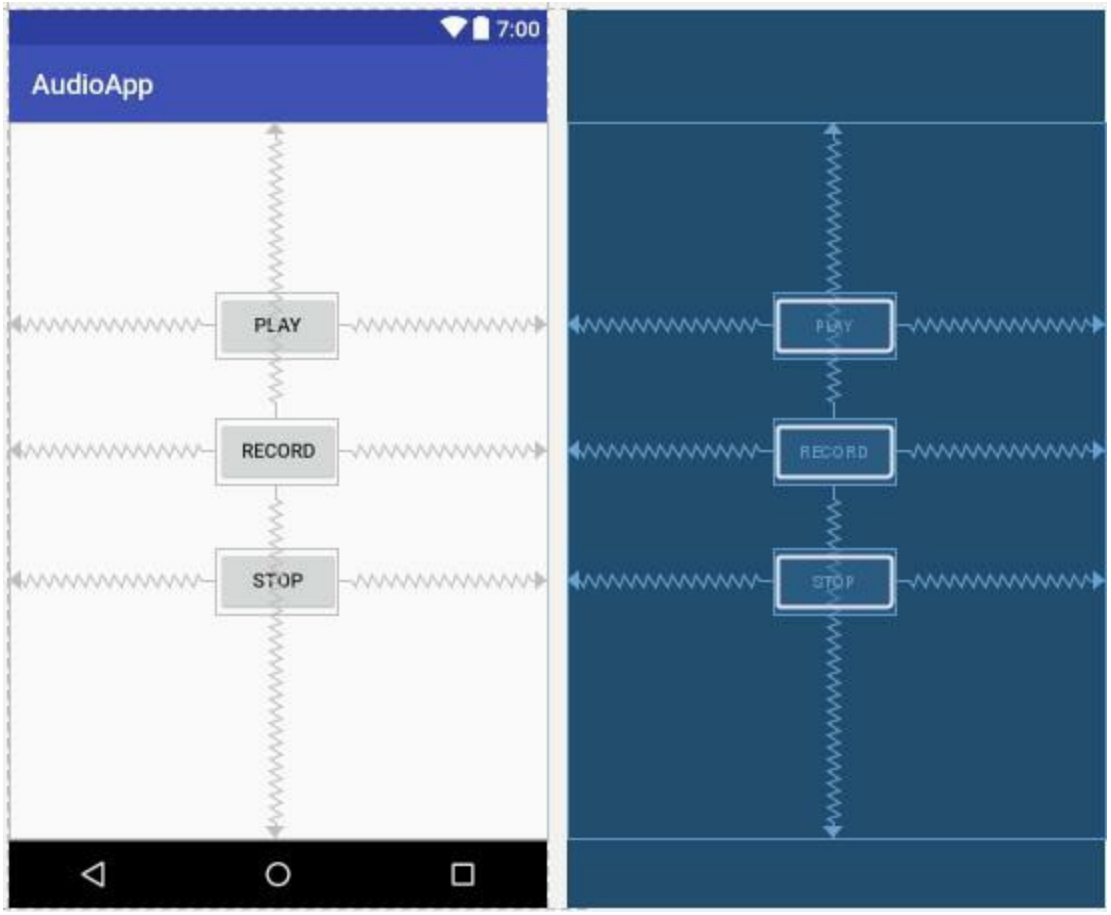

Figure 67-1

Configure the buttons to display string resources that read *Play, Record* and *Stop* and give them view IDs of *playButton*, *recordButton*, and *stopButton* respectively.

Select the Play button and, within the Properties panel, configure the *onClick* property to call a method named *playAudio* when selected by the user. Repeat these steps to configure the remaining buttons to call methods named *recordAudio* and *stopAudio* respectively.

67.6 Checking for Microphone Availability

Attempting to record audio on a device without a microphone will cause the Android system to throw an exception. It is vital, therefore, that the code check for the presence of a microphone before making such an attempt. There are a number of ways of doing this, including checking for the physical presence of the device. An easier approach, and one that is more likely to work on different Android devices, is to ask the Android system if it has a package installed for a particular *feature*. This involves creating an instance of the Android PackageManager class and then making a call to the object's *hasSystemFeature()* method. *PackageManager.FEATURE_MICROPHONE* is the feature of interest in this case.

For the purposes of this example, we will create a method named *hasMicrophone()* that may be called upon to check for the presence of a microphone. Within the Project tool window, locate and double-click on the *AudioAppActivity.java* file and modify it to add this method:

```
package com.ebookfrenzy.audioapp;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.pm.PackageManager;
```

```
public class AudioAppActivity extends AppCompatActivity {

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

    protected boolean hasMicrophone() {
        PackageManager pmanager = this.getPackageManager();
        return pmanager.hasSystemFeature(
                PackageManager.FEATURE_MICROPHONE);
    }
}
```

67.7 Performing the Activity Initialization

The next step is to modify the *onCreate()* method of the activity to perform a number of initialization tasks. Remaining within the *AudioAppActivity.java* file, modify the method as follows:

```
package com.ebookfrenzy.audioapp;

import java.io.IOException;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.pm.PackageManager;
import android.media.MediaRecorder;
import android.os.Environment;
import android.widget.Button;
import android.view.View;
import android.media.MediaPlayer;

public class AudioAppActivity extends AppCompatActivity {

        private static MediaRecorder mediaRecorder;
        private static MediaPlayer mediaPlayer;

        private static String audioFilePath;
        private static Button stopButton;
        private static Button playButton;
        private static Button recordButton;

        private boolean isRecording = false;

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

               recordButton =
                       (Button) findViewById(R.id.recordButton);
                playButton = (Button) findViewById(R.id.playButton);
               stopButton = (Button) findViewById(R.id.stopButton);
```

```
                if (!hasMicrophone())
                {
                        stopButton.setEnabled(false);
                        playButton.setEnabled(false);
                        recordButton.setEnabled(false);
                } else {
                        playButton.setEnabled(false);
                        stopButton.setEnabled(false);
                }

                audioFilePath =

          Environment.getExternalStorageDirectory().getAbsolutePath()
                  + "/myaudio.3gp";
        }
    .
    .
    }
```

The added code begins by obtaining references to the three button views in the user interface. Next, the previously implemented *hasMicrophone()* method is called to ascertain whether the device includes a microphone. If it does not, all the buttons are disabled, otherwise only the Stop and Play buttons are disabled.

The next line of code needs a little more explanation:

```
audioFilePath =
        Environment.getExternalStorageDirectory().getAbsolutePath()
                + "/myaudio.3gp";
```

The purpose of this code is to identify the location of the SD card storage on the device and to use that to create a path to a file named *myaudio.3gp* into which the audio recording will be stored. The path of the SD card (which is referred to as external storage even though it is internal to the device on many Android devices) is obtained via a call to the *getExternalStorageDirectory()* method of the Android Environment class.

When working with external storage it is important to be aware that such activity by an application requires permission to be requested in the application manifest file. For example:

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

67.8 Implementing the recordAudio() Method

When the user touches the Record button, the *recordAudio()* method will be called. This method will need to enable and disable the appropriate buttons and configure the MediaRecorder instance with information about the source of the audio, the output format and encoding, and the location of the file into which the audio is to be stored. Finally, the *prepare()* and *start()* methods of the MediaRecorder object will need to be called. Combined, these requirements result in the following method implementation in the *AudioAppActivity.java* file:

```
public void recordAudio (View view) throws IOException
{
   isRecording = true;
   stopButton.setEnabled(true);
   playButton.setEnabled(false);
   recordButton.setEnabled(false);
```

```
    try {
      mediaRecorder = new MediaRecorder();
      mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
      mediaRecorder.setOutputFormat(
           MediaRecorder.OutputFormat.THREE_GPP);
      mediaRecorder.setOutputFile(audioFilePath);
      mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
      mediaRecorder.prepare();
    } catch (Exception e) {
          e.printStackTrace();
    }

    mediaRecorder.start();
}
```

67.9 Implementing the stopAudio() Method

The *stopAudio()* method is responsible for enabling the Play button, disabling the Stop button and then stopping and resetting the MediaRecorder instance. The code to achieve this reads as outlined in the following listing and should be added to the *AudioAppAcitivy.java* file:

```
public void stopAudio (View view)
{

        stopButton.setEnabled(false);
        playButton.setEnabled(true);

        if (isRecording)
        {
               recordButton.setEnabled(false);
               mediaRecorder.stop();
               mediaRecorder.release();
               mediaRecorder = null;
               isRecording = false;
        } else {
               mediaPlayer.release();
                mediaPlayer = null;
                recordButton.setEnabled(true);
        }
}
```

67.10 Implementing the playAudio() method

The *playAudio()* method will simply create a new MediaPlayer instance, assign the audio file located on the SD card as the data source and then prepare and start the playback:

```
public void playAudio (View view) throws IOException
{
        playButton.setEnabled(false);
        recordButton.setEnabled(false);
        stopButton.setEnabled(true);

        mediaPlayer = new MediaPlayer();
        mediaPlayer.setDataSource(audioFilePath);
        mediaPlayer.prepare();
        mediaPlayer.start();
```

```
}
```

67.11 Configuring and Requesting Permissions

Before testing the application, it is essential that the appropriate permissions be requested within the manifest file for the application. Specifically, the application will require permission to record audio and to access the external storage (SD card). Within the Project tool window, locate and double-click on the *AndroidManifest.xml* file to load it into the editor and modify the XML to add the two permission tags:

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

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity android:name=".AudioAppActivity"
            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>
```

The above steps will be adequate to ensure that the user enables these permissions when the app is installed on devices running versions of Android pre-dating Android 6.0. Both microphone and external storage access are categorized in Android as being dangerous permissions because they give the app the potential to compromise the user's privacy. In order for the example app to function on Android 6 or later devices, therefore, code needs to be added to specifically request these two permissions at app runtime.

Edit the *AudioAppActivity.java* file and begin by adding some additional import directives and constants to act as request identification codes for the permissions being requested:

```
package com.ebookfrenzy.audioapp;

import java.io.IOException;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.content.pm.PackageManager;
import android.media.MediaRecorder;
import android.os.Environment;
```

```
import android.widget.Button;
import android.view.View;
import android.media.MediaPlayer;
import android.widget.Toast;
import android.support.v4.content.ContextCompat;
import android.Manifest;
import android.support.v4.app.ActivityCompat;

public class AudioAppActivity extends AppCompatActivity {

    private static final int RECORD_REQUEST_CODE = 101;
    private static final int STORAGE_REQUEST_CODE = 102;

    private static MediaRecorder mediaRecorder;
    private static MediaPlayer mediaPlayer;
.
.
.
```

Next, a method needs to be added to the class, the purpose of which is to take as arguments the permission to be requested and the corresponding request identification code. Remaining with the *AudioAppActivity.java* class file, implement this method as follows:

```
protected void requestPermission(String permissionType, int requestCode) {
    int permission = ContextCompat.checkSelfPermission(this,
                permissionType);

    if (permission != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
                new String[]{permissionType}, requestCode
        );
    }
}
```

Using the steps outlined in the *Making Runtime Permission Requests in Android 6.0* chapter of this book, the above method verifies that the specified permission has not already been granted before making the request, passing through the identification code as an argument.

When the request has been handled, the *onRequestPermissionsResult()* method will be called on the activity, passing through the identification code and the results of the request. The next step, therefore, is to implement this method within the *AudioAppActivity.java* file as follows:

```
@Override
public void onRequestPermissionsResult(int requestCode,
                        String permissions[], int[] grantResults) {
    switch (requestCode) {
        case RECORD_REQUEST_CODE: {

            if (grantResults.length == 0
                 || grantResults[0] !=
                      PackageManager.PERMISSION_GRANTED) {

                recordButton.setEnabled(false);

                Toast.makeText(this,
                      "Record permission required",
```

```
                                Toast.LENGTH_LONG).show();
            } else {
                requestPermission(
                     Manifest.permission.WRITE_EXTERNAL_STORAGE,
                        STORAGE_REQUEST_CODE);
            }
            return;
        }
        case STORAGE_REQUEST_CODE: {

            if (grantResults.length == 0
                    || grantResults[0] !=
                      PackageManager.PERMISSION_GRANTED) {
                recordButton.setEnabled(false);
                Toast.makeText(this,
                      "External Storage permission required",
                      Toast.LENGTH_LONG).show();
            }
            return;
        }
    }
}
```

The above code checks the request identifier code to identify which permission request has returned before checking whether or not the corresponding permission was granted. If the user grants permission to access the microphone the code then proceeds to request access to the external storage. In the event that either permission was denied, a message is displayed to the user indicating the app will not function. In both instances, the record button is also disabled.

All that remains prior to testing the app is to call the newly added *requestPermission()* method for microphone access when the app launches. Remaining in the *AudioAppActivity.java* file, modify the *onCreate()* method as follows:

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

    recordButton = (Button) findViewById(R.id.recordButton);
    playButton = (Button) findViewById(R.id.playButton);
    stopButton = (Button) findViewById(R.id.stopButton);

    if (!hasMicrophone())
    {
        stopButton.setEnabled(false);
        playButton.setEnabled(false);
        recordButton.setEnabled(false);
    } else {
        playButton.setEnabled(false);
        stopButton.setEnabled(false);
    }

    audioFilePath =
            Environment.getExternalStorageDirectory().getAbsolutePath()
                    + "/myaudio.3gp";
```

```
        requestPermission(Manifest.permission.RECORD_AUDIO,
                                                RECORD_REQUEST_CODE);
    }
```

67.12 Testing the Application

Compile and run the application on an Android device containing a microphone, allow the requested permissions and touch the Record button. After recording, touch Stop followed by Play, at which point the recorded audio should play back through the device speakers. If running on Android 6.0 or later, note that the app requests permission to use the external storage and to record audio when first launched.

67.13 Summary

The Android SDK provides a number of mechanisms for the implementation of audio recording and playback. This chapter has looked at two of these, in the form of the MediaPlayer and MediaRecorder classes. Having covered the theory of using these techniques, this chapter worked through the creation of an example application designed to record and then play back audio. In the course of working with audio in Android, this chapter also looked at the steps involved in ensuring that the device on which the application is running has a microphone before attempting to record audio. The use of external storage in the form of an SD card was also covered.