Chapter 5

Introduction to Audio on Android

Any smartphone worth its name these days has audio playback capabilities on par with dedicated portable media devices or MP3 players. Of course, Android-based devices are no different. These capabilities allow for the building of music player, audio book, podcast, or just about any other type of application that is centered around audio playback.

In this chapter, we'll explore what Android's capabilities are in terms of format and codec support, and we'll build a few different playback applications. Furthermore we'll look at what Android supports in the way of audio formats and metadata.

Audio Playback

As mentioned, Android supports audio playback capabilities on par with MP3 players. In fact, it probably goes a step further since it supports a fairly wide range of audio formats, more than most hardware players. This is one of the benefits of smartphones that perform functions previously relegated to dedicated hardware, since they have good faculties for running a variety of software; like a computer, they can offer a wide range of support for different and changing technologies that are simply not practical to build into the firmware of hardware-centric devices.

Supported Audio Formats

Android supports a variety of audio file formats and codecs for playback (it supports fewer for recording, which we'll discuss when we go over recording).

- *AAC*: Advanced Audio Coding codec (as well as both profiles of HE-AAC, High Efficiency AAC), .m4a (audio/m4a) or.3gp (audio/3gpp) files. AAC is a popular standard that is used by the iPod and other portable media players. Android supports this audio format inside of MPEG-4 audio files and inside of 3GP files (which are based on the MPEG-4 format). Recent additions to the AAC specification, High Efficiency AAC are also supported.
- *MP3*: MPEG-1 Audio Layer 3, .mp3 (audio/mp3) files. MP3, probably the most widely used audio codec, is supported. This allows Android to utilize the vast majority of audio available online through various web sites and music stores.
- *AMR*: Adaptive Multi-Rate codec (both AMR Narrowband, AMR-NB, and AMR Wideband, AMR-WB), .3gp (audio/3gpp) or .amr (audio/amr) files. AMR is the audio codec that has been standardized as the primary voice audio codec in use by the 3GPP (3rd Generation Partnership Project). The 3GPP is a telecommunications industry organization that creates specifications for the partner companies to use. In other words, the AMR codec is what is primarily used for voice calling applications on modern mobile phones and generally supported across mobile handset manufacturers and carriers. As such, this codec is generally useful for voice encoding but doesn't perform well for more complex types of audio such as music.
- *Ogg*: Ogg Vorbis, .ogg (application/ogg) files. Ogg Vorbis is an open source, patent-free audio codec with quality that is comparable to commercial and patent-encumbered codecs such as MP3 and AAC. It was developed by volunteers and is currently maintained by the Xiph.Org foundation.

- *PCM*: Pulse Code Modulation commonly used in WAVE or WAV files (Waveform Audio Format), .wav (audio/x-wav) files. PCM is the technique used for storing audio on computers and other digital audio devices. It is generally an uncompressed audio file with data that represents the amplitude of a piece of audio over time. The "sample rate" is how often an amplitude reading is stored. The "bit-depth" is how many bits are used to represent an individual sample. A piece of audio data with a sample rate of 16kHz and a bit-depth of 32 bits means that it will contain 32 bits of data representing the amplitude of the audio and it will have 16,000 of these per second. The higher the sample rate and the higher the bit-depth, the more accurate the digitization of the audio is. Sample rate and bit-depth also determine how large the audio file will be when its length is taken into account. Android supports PCM audio data within WAV files. WAV is a long-standing standard audio format on PCs.

Using the Built-In Audio Player via an Intent

As with using the camera, the easiest way to provide the ability to play an audio file within an application is to leverage the capabilities of the built-in "Music" application. This application plays all of the formats that Android supports, has a familiar interface to the user, and can be triggered to play a specific file via an intent.

The generic `android.content.Intent.ACTION_VIEW` intent with the data set to a Uri to the audio file and the MIME type specified allows Android to pick the appropriate application for playback. This should be the Music application, but the user may be presented with other options if he or she has other audio playback software installed.

```
Intent intent = new Intent(android.content.Intent.ACTION_VIEW);
intent.setDataAndType(audioFileUri, "audio/mp3");
startActivity(intent);
```

NOTE: MIME stands for Multipurpose Internet Mail Extensions. It was originally specified to help e-mail clients send and receive attachments. Its use, though, has extended greatly beyond e-mail to many other communication protocols, including HTTP or standard web serving. Android uses MIME types when resolving an intent, specifically to help determine which application should handle the intent.

Each file type has a specific (sometimes more than one) MIME type. This type is specified using at least two parts with slashes between them. The first is the more generic type, such as "audio." The second part is the more specific type, such as "mpeg." A generic type "audio" and a more specific type "mpeg" would yield a MIME type string of "audio/mpeg," which is the MIME type typically used for MP3 files.

Here is a full example of triggering the built-in audio player application through an intent:

```
package com.apress.proandroidmedia.ch5.intentaudioplayer;

import java.io.File;
import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
```

Our activity will be listening for a Button to be pressed before it triggers the playback of the audio. It implements `OnClickListener` so that it can respond.

```
public class AudioPlayer extends Activity implements OnClickListener {

    Button playButton;

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

After we set the content view to our XML, we can get a reference to our Button in code and set our activity (`this`) to be the `OnClickListener`.

```
        playButton = (Button) this.findViewById(R.id.Button01);
        playButton.setOnClickListener(this);
    }
```

When our Button is clicked, the `onClick` method is called. In this method, we construct the intent with a generic `android.content.Intent.ACTION_VIEW` and then create a `File` object that is a reference to an audio file that exists on the SD card. In this case, the audio file is one that is manually placed on the SD card in the "Music" directory, which is the standard location for music-related audio files.

```
    public void onClick(View v) {
        Intent intent = new Intent(android.content.Intent.ACTION_VIEW);

        File sdcard = Environment.getExternalStorageDirectory();
        File audioFile = new File(sdcard.getPath() + "/Music/goodmorningandroid.mp3");
```

Next, we set the data of the intent to be a Uri derived from the audio file and the type to be its MIME type, `audio/mp3`. Finally, we trigger the built-in application to launch via the `startActivity` call passing in our intent. Figure 5–1 shows the built-in application playing the audio file.

```
        intent.setDataAndType(Uri.fromFile(audioFile), "audio/mp3");
        startActivity(intent);
    }
}
```

Here is a simple Layout XML file specifying the Button with the text "Play Audio" to be used with the foregoing activity.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button android:text="Play Audio" android:id="@+id/Button01"➥
    android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
</LinearLayout>
```

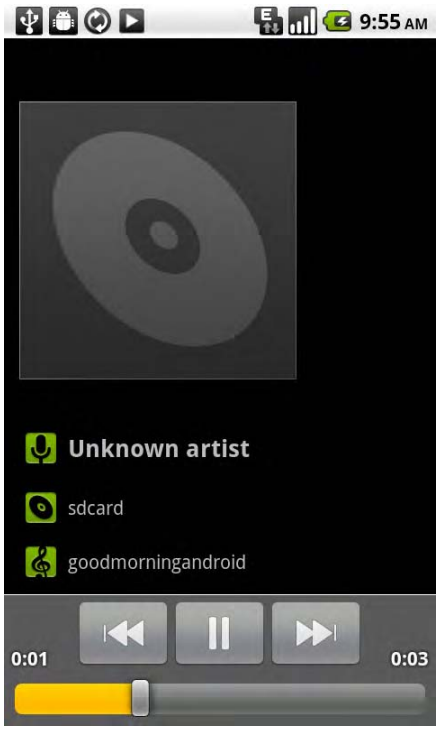


Figure 5–1. Android's built-in music player playing an audio file specified via an intent

Creating a Custom Audio-Playing Application

Of course, we aren't limited to using Android's built-in application for audio playback. We can write our own application that offers playback capabilities and more.

To enable this, Android includes a `MediaPlayer` class. This class is used for the playback and control of both audio and video. Right now we'll just be using the audio playback capabilities.

The simplest `MediaPlayer` example is to play back an audio file that is packaged with the application itself. In order to do that, an audio file should be placed within the application's raw resources. To do this using the Android Developer Tools on Eclipse, we need to create a new folder in our Project's `res` folder called `raw` as illustrated in

Figure 5–2. The Android Developer Tools will generate a resource id for this file in the `R.java` file (in the `gen` folder) with the syntax `R.raw.file_name_without_extension`.

Figure 5–2. Custom audio player Eclipse Project layout showing audio file located in `raw` folder inside `res` folder.

Starting the Media Player

Creating a `MediaPlayer` for this audio file is straightforward. We instantiate a `MediaPlayer` object using the static method `create`, passing in `this` as a Context (which Activity is descended from) and the generated resource ID of the audio file.

```
MediaPlayer mediaPlayer = MediaPlayer.create(this, R.raw.goodmorningandroid);
```

Following that, we simply call the `start` method on the `MediaPlayer` object to play it.

```
mediaPlayer.start();
```

LOCAL ASSETS

When placing assets in the `res` folder of an Android Developer Tools/Eclipse project, a couple of things have to be considered: the file extension and using Uris.

File Extensions

The extension is removed, so files with the same base name but different extensions will cause issues. You wouldn't want to put a file named `goodmorningandroid.mp3` and another file named `goodmorningandroid.m4a` in there. Instead, if you would like to provide the same audio in multiple formats, it would be better if you included the format as part of the file name so that you can differentiate between them and the Android Developer Tools doesn't have problems generating the resource ID. If you name them `goodmorningandroid_mp3.mp3` and `goodmorningandroid_m4a.m4a`, you will be able to reference them as `R.raw.goodmorningandroid_mp3` and `R.raw.goodmorningandroid_m4a` respectively.

Uris for Resource Files

While resource IDs are great for some purposes, they don't suit all. As we already know, many things in Android can be accomplished using a Uri. Fortunately, it is easy to construct a Uri for a file that has been placed in the resources. The resource ID can be appended to the end of a string, which can be used to construct the Uri. The string must start with `android.resource://`, followed by the package name of the application that the resources are local to, followed by the resource ID of the file.

Here is an example:

```
Uri fileUri = Uri.parse("android.resource://com.apress.proandroidmedia.ch5.customaudio/"
+ R.raw.goodmorningandroid);
```

To use the `MediaPlayer` with a Uri instead of a resource ID, which we will have to do if the file isn't part of the application, we can call a `create` method passing in the context and the Uri.

```
MediaPlayer mediaPlayer = MediaPlayer.create(this, fileUri);
```

Controlling Playback

The `MediaPlayer` class has several nested classes that are interfaces for listening to events that the `MediaPlayer` sends. These events relate to state changes.

For instance, the `MediaPlayer` will call the `onCompletion` method on a class that implements the `OnCompletionListener` and is registered via the `setOnCompletionListener`. This will be done when an audio file is done playing.

Here is a full example of an activity that infinitely repeats, playing the same audio file by using the `OnCompletionListener`. The `MediaPlayer` object is initialized and playback started in the `onStart` method, with playback stopped and the `MediaPlayer` object released in the `onStop` method. This prevents the audio from playing when the activity is no longer in the front but restarts it when the activity is brought to the front again.

```
package com.apress.proandroidmedia.ch5.customaudio;

import android.app.Activity;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;
import android.os.Bundle;

public class CustomAudioPlayer extends Activity implements OnCompletionListener {

    MediaPlayer mediaPlayer;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

    }

    public void onStart() {
        super.onStart();
        mediaPlayer = MediaPlayer.create(this, R.raw.goodmorningandroid);
        mediaPlayer.setOnCompletionListener(this);
        mediaPlayer.start();
    }

    public void onStop() {
        super.onStop();
        mediaPlayer.stop();
        mediaPlayer.release();
    }

    public void onCompletion(MediaPlayer mp) {
        mediaPlayer.start();
    }
}
```

Of course, this could be done without the `OnCompletionListener` by simply setting the `MediaPlayer` to loop using the `setLooping(true)` method.

Let's take this a step further and make it so that the playback is controlled by touch events. This code might be a good starting point for making a DJ audio scratching application.

```
package com.apress.proandroidmedia.ch5.customaudio;

import android.app.Activity;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;
import android.os.Bundle;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.View.OnTouchListener;
import android.widget.Button;
```

Our activity will implement the OnCompletionListener, as did the previous example, but will also implement the OnTouchListener, so that it can respond to touch events, and the OnClickListener, so that it can respond when a user clicks a button.

```
public class CustomAudioPlayer extends Activity implements OnCompletionListener,
OnTouchListener, OnClickListener {
```

Of course, we'll need a reference to the MediaPlayer object. We need access to a View so that we can register that we want touch events, and we'll need access to any buttons that we define in the Layout XML file—in this case, a button for stopping playback and a button for starting playback.

```
    MediaPlayer mediaPlayer;
    View theView;
    Button stopButton, startButton;
```

We'll declare a variable that will contain a saved position in the audio file. We'll use this position later to determine where to start playing the audio file.

```
    int position = 0;

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

Using our normal findViewById function, we'll get access to the Buttons and the View that are defined in the Layout XML.

```
        stopButton = (Button) this.findViewById(R.id.StopButton);
        startButton = (Button) this.findViewById(R.id.StartButton);
```

Since our activity can respond to Click events, we'll make it so that it is registered as the listener on both Buttons.

```
        startButton.setOnClickListener(this);
        stopButton.setOnClickListener(this);

        theView = this.findViewById(R.id.theview);
```

Then we'll make our activity (this) respond to the touch events.

```
        theView.setOnTouchListener(this);
```

In this application, our audio file is called goodmorningandroid.mp3, and it has been placed in the res/raw folder of our project. We'll create our MediaPlayer object using this file. Also, as in the previous example, we are setting our activity to be the OnCompletionListener for our MediaPlayer.

```
        mediaPlayer = MediaPlayer.create(this, R.raw.goodmorningandroid);
        mediaPlayer.setOnCompletionListener(this);
        mediaPlayer.start();
    }
```

Here our onCompletion method is defined. It gets called whenever the MediaPlayer has finished playing our audio file. In this case, we'll call start first to make the audio play

and then call seek to the saved position. The audio needs to be playing before we can seek.

```
    public void onCompletion(MediaPlayer mp) {
        mediaPlayer.start();
        mediaPlayer.seekTo(position);
    }
```

When the user triggers a touch event, the `onTouch` method is called. In this method, we are paying attention only to the `ACTION_MOVE` touch event, which is triggered when the user drags a finger across the surface of the View. In this case, we'll make sure the `MediaPlayer` is playing and then calculate where we should seek to based upon where on the screen the touch event occurs. If it occurs toward the right boundary of the screen, we'll seek toward the end of the file. If it occurs toward the left boundary of the screen, we'll seek to near the beginning of the file. We save this value in the position variable, so that when the audio file finishes, it will seek back to that point when it starts playing again (in the `onCompletion` method).

```
    public boolean onTouch(View v, MotionEvent me) {

        if (me.getAction() == MotionEvent.ACTION_MOVE)
        {
            if (mediaPlayer.isPlaying()) {
                position = (int) (me.getX() *
                    mediaPlayer.getDuration()/theView.getWidth());
                Log.v("SEEK",""+position);
                mediaPlayer.seekTo(position);
            }
        }

        return true;
    }
```

Last we have an `onClick` method that responds to the Button clicks. These clicks pause and start the audio playback.

```
    public void onClick(View v) {
        if (v == stopButton) {
            mediaPlayer.pause();
        } else if (v == startButton) {
            mediaPlayer.start();
        }

    }
}
```

Here is the Layout XML file that is being referred to in the foregoing code example:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button android:text="Start" android:id="@+id/StartButton"↩
    android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
```

```
    <Button android:text="Stop" android:id="@+id/StopButton"↩
     android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
    <View android:layout_width="fill_parent" android:layout_height="fill_parent"↩
    android:id="@+id/theview" />
</LinearLayout>
```

As you can see, building a custom audio player on Android opens up some interesting possibilities. Applications can be built that do more than just play audio straight through. Turning this into a full-fledged DJ application for the phone could be fun.

MediaStore for Audio

We explored using the `MediaStore` for images early on in this book. Much of what we learned can be leveraged for the storage and retrieval of other types of media, including audio. In order to provide a robust mechanism for browsing and searching for audio, Android includes a `MediaStore.Audio` package, which defines the standard content provider.

Accessing Audio from the MediaStore

Accessing audio files that are stored using the `MediaStore` provider is consistent with our previous uses of the `MediaStore`. In this case, we'll be using the `android.provider.MediaStore.Audio` package.

One of the easiest ways to illustrate the use of the `MediaStore` for audio is to go through a sample application. The following code creates an activity that queries the MediaStore for any audio file and simply plays the first one returned.

```
package com.apress.proandroidmedia.ch5.audioplayer;

import java.io.File;
import android.app.Activity;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.provider.MediaStore;

public class AudioPlayer extends Activity {

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

To use the `MediaStore`, we need to specify which data we want returned. We do this by creating an array of strings using the constants located in the `android.provider.MediaStore.Audio.Media` class. Those constants are all of the standard fields that are saved in the `MediaStore` for use with audio.

In this case, we are asking for the DATA column, which contains the path to the actual audio file. We are also asking for the internal ID, the Title, Display Name, MIME-Type, Artist, Album, and which type of audio file it is, alarm, music, ring tone, or notification type.

Other columns such as date added (DATE_ADDED), date modified (DATE_MODIFIED), file size (SIZE), and so on are available as well.

```
String[] columns = {
    MediaStore.Audio.Media.DATA,
    MediaStore.Audio.Media._ID,
    MediaStore.Audio.Media.TITLE,
    MediaStore.Audio.Media.DISPLAY_NAME,
    MediaStore.Audio.Media.MIME_TYPE,
    MediaStore.Audio.Media.ARTIST,
    MediaStore.Audio.Media.ALBUM,
    MediaStore.Audio.Media.IS_RINGTONE,
    MediaStore.Audio.Media.IS_ALARM,
    MediaStore.Audio.Media.IS_MUSIC,
    MediaStore.Audio.Media.IS_NOTIFICATION
};
```

We query the MediaStore by calling the managedQuery method in Activity. The managedQuery method takes in the Uri for the content provider, in this case, the audio MediaStore, android.provider.MediaStore.Audio.Media.EXTERNAL_CONTENT_URI. This Uri specifies that we want audio stored on the SD card. If we wanted audio files that are stored in the internal memory, we would use android.provider.MediaStore.Audio.Media.INTERNAL_CONTENT_URI.

In addition to the Uri to the MediaStore, the managedQuery method takes in the array of columns that we want returned, an SQL WHERE clause, the values for the WHERE clause, and an SQL ORDER BY clause.

In this example, we aren't using the WHERE and ORDER BY clauses, so we'll pass in null for those arguments.

```
Cursor cursor = managedQuery(MediaStore.Audio.Media.EXTERNAL_CONTENT_URI,↩
columns, null, null, null);
```

The managedQuery method returns a Cursor object. The Cursor class allows interaction with a dataset returned from a database query.

The first thing we'll do is create a couple of variables to hold the column numbers for some of the columns we want to access from the results. This isn't absolutely necessary, but it is nice to have the index around so we don't have to call the method on the Cursor object each time we need them. The way we get them is to pass in the constant value for the column we want to the getColumnIndex method on the Cursor.

```
int fileColumn = cursor.getColumnIndex (MediaStore.Audio.Media.DATA);
```

The first one is the index of the column containing the path to the actual audio file. We got the foregoing index by passing in the constant that represents that column, android.provider.MediaStore.Audio.Media.DATA.

Next we are getting a couple of other indexes, not all of which we are actually using, so the extras are here merely for illustration purposes.

```
        int titleColumn = cursor.getColumnIndex (MediaStore.Audio.Media.TITLE);
        int displayColumn = cursor.getColumnIndex (MediaStore.Audio.Media.DISPLAY_NAME);
        int mimeTypeColumn = cursor.getColumnIndex (MediaStore.Audio.Media.MIME_TYPE);
```

The data returned by the `MediaStore` available in the `Cursor` is organized in rows as well as by columns. We can get the first result returned by calling the `moveToFirst` method and retrieving the results from there. The method will return a Boolean false if no rows are returned, so we can wrap it in an if statement to make sure there is data.

```
        if (cursor.moveToFirst()) {
```

To get the actual data, we call one of the "get" methods on the `Cursor` and pass in the index for the column we want to retrieve. If the data is expected to be a String, we call `getString`. If it is expected to be an integer, we call `getInt`. There are corresponding "get" methods for all of the primitive data types.

```
            String audioFilePath = cursor.getString(fileColumn);
            String mimeType = cursor.getString(mimeTypeColumn);

            Log.v("AUDIOPLAYER",audioFilePath);
            Log.v("AUDIOPLAYER",mimeType);
```

Once we have the path to the file and the MIME type, we can use those to construct the intent to launch the built-in audio player application and play that file. (Alternatively we could use the `MediaPlayer` class as illustrated previously to play the audio file directly.) In order to turn the path to the audio file into a Uri that we can pass into the intent, we construct a `File` object and use the `Uri.fromFile` method to get the Uri. There are other ways to do the same, but this is probably the most straightforward.

```
            Intent intent = new Intent(android.content.Intent.ACTION_VIEW);
            File newFile = new File(audioFilePath);
            intent.setDataAndType(Uri.fromFile(newFile), mimeType);
            startActivity(intent);
        }
    }
}
```

That finishes off our basic illustration of using the `MediaStore` for audio.

Now, let's take it a step further and create an application that allows us to narrow down the results returned and browse them, allowing the user to select the audio file to play.

Browsing Audio in the MediaStore

Audio files, in particular music files, can be found by album, artist, and genre as well as directly in the `MediaStore`. Each of these has an Uri that can be used with a `managedQuery` to search with.

- *Album*: `android.provider.MediaStore.Audio.Albums.EXTERNAL_CONTENT_URI`
- *Artist*: `android.provider.MediaStore.Artists.EXTERNAL_CONTENT_URI`
- *Genre*: `android.provider.MediaStore.Genres.EXTERNAL_CONTENT_URI`

Here is how you would use the album Uri to query for all of the albums on the device:

```
String[] columns = { android.provider.MediaStore.Audio.Albums._ID,
          android.provider.MediaStore.Audio.Albums.ALBUM};
Cursor cursor = managedQuery(MediaStore.Audio.Albums.EXTERNAL_CONTENT_URI, columns,↵
null, null, null);
if (cursor != null) {
    while (cursor.moveToNext()) {
        Log.v("OUTPUT",
              cursor.getString(cursor.getColumnIndex(MediaStore.Audio.Albums.ALBUM)));
    }
}
```

In the foregoing code snippet, you see that we are asking the `MediaStore` to return the `_ID` and the `ALBUM` columns. The `ALBUM` constant indicates that we want the name of the album returned. Other columns available are listed in the `android.provider.MediaStore.Audio.Albums` class and are inherited from `android.provider.BaseColumns` and `android.provider.MediaStore.Audio.AlbumColumns`.

We are calling the `managedQuery` method giving just the Uri and the list of columns, leaving the other parameters as `null`. This will give us all of the albums available on the device.

Finally, we are outputting the list of albums. To iterate through the list returned inside the `Cursor` object, we first check that the `Cursor` contains results (`cursor != null`) and then use the `moveToNext` method.

Album Browsing App Example

What follows is an example that uses the foregoing as a starting point to allow the user to see the names of all of the albums. The user can indicate which album he or she would like to see the songs on. It will then present the list of songs, and if the user selects one of those, it will play that song.

```
package com.apress.proandroidmedia.ch5.audiobrowser;

import java.io.File;

import android.app.ListActivity;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.MediaStore;
import android.util.Log;
import android.view.View;
import android.widget.ListView;
import android.widget.SimpleCursorAdapter;
```

Instead of extending a generic `activity`, let's extend `ListActivity`. This allows us to present and manage a basic `ListView`.

```
public class AudioBrowser extends ListActivity {

    Cursor cursor;
```

Let's create a couple of constants that will help us keep track of where the user is in the application and respond appropriately when the user performs an action. This will be kept track of in the `currentState` variable that is initially set to `STATE_SELECT_ALBUM`.

```
public static int STATE_SELECT_ALBUM = 0;
public static int STATE_SELECT_SONG = 1;

int currentState = STATE_SELECT_ALBUM;
```

Just like a normal activity, we have an `onCreate` method where we can perform the initial commands.

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

After setting the layout (via the `main.xml` layout XML file), we create an array of Strings that represents the columns we want returned from the `MediaStore` when we run our query. In this case, it is the same as the foregoing snippet of code—we want the `_ID` and the name of the album, `ALBUM`. Both are constants in the `MediaStore.Audio.Albums` class.

```
        String[] columns = {
            android.provider.MediaStore.Audio.Albums._ID,
            android.provider.MediaStore.Audio.Albums.ALBUM
        };
```

We call the `managedQuery` method with only the Uri representing the album search and the columns, leaving everything else null. This should give us a list of all of the albums available.

```
        cursor = managedQuery(MediaStore.Audio.Albums.EXTERNAL_CONTENT_URI, columns,
null, null, null);
```

Once we do this, we are returned a `Cursor` object that contains the results of our query.

Since we are using a `ListActivity`, we have the ability to have it automagically manage the list of data for us. We can use the `setListAdapter` method to bind our `Cursor` object to `ListView`.

First we create an array of Strings that is the name of the columns in the `Cursor` that we want displayed. In our case, we just want the name of the album—`MediaStore.Audio.Albums.ALBUM` is our constant for that.

Next we list the View objects that will display the data from those columns. Since we just have one column, we need only one View object. It is `android.R.id.text1`. This View is available to us, as it is part of the `android.R.layout.simple_list_item_1` layout that we'll be using in the next step.

Last we call the `setListAdapter` method, passing in a `SimpleCursorAdapter`, which we are creating inline. The `SimpleCursorAdapter` is a simple adapter from a `Cursor` object containing data to a `ListActivity`. In creating the `SimpleCursoryAdapter`, we pass in our activity (`this`) as the Context, a standard `ListView` layout that is already defined for us (`android.R.layout.simple_list_item_1`), the `Cursor` object containing the data, and the two arrays we just defined.

```
        String[] displayFields = new String[] {MediaStore.Audio.Albums.ALBUM};
        int[] displayViews = new int[] {android.R.id.text1};
        setListAdapter(new SimpleCursorAdapter(this,↩
                android.R.layout.simple_list_item_1, cursor, displayFields,↩
displayViews));

    }
```

If we were to run this as is, we would get a simple list of the albums available on our device as shown in Figure 5–3. We are going to take it a step further, though, and allow the user to select an album.

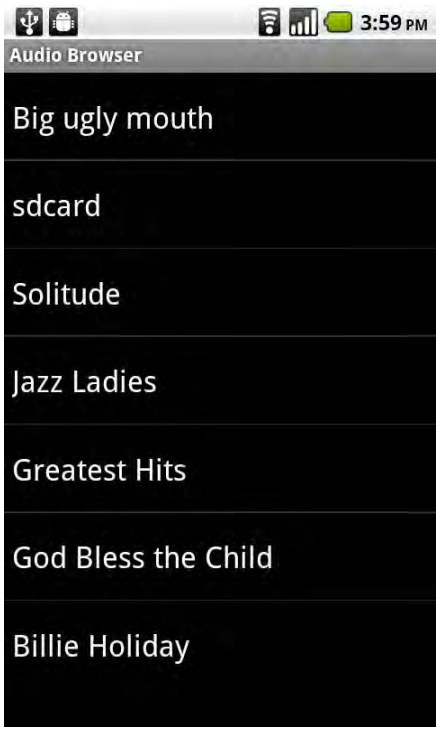


Figure 5–3. Albums listed in basic `ListView`

To allow the user to actually select one of the albums, we need to override the default `onListItemClick` method that is provided by our `ListActivity` parent class.

```
    protected void onListItemClick(ListView l, View v, int position, long id) {
        if (currentState == STATE_SELECT_ALBUM) {
```

When an album in the list is selected, this method will be called. Since our `currentState` variable starts off as `STATE_SELECT_ALBUM` the first time this method is called it should be `true`.

The position of the selected album in the list will be passed in and can be used with the `Cursor` object to get at the data about which album it was by calling the `moveToPosition` method.

```
if (cursor.moveToPosition(position)) {
```

Assuming the moveToPosition was successful, we are going to start all over again querying the MediaStore. This time, though, we are going to run our managedQuery on MediaStore.Audio.Media.EXTERNAL_CONTENT_URI as we want access to the individual media files.

First we choose the columns that we want returned.

```
String[] columns = {
        MediaStore.Audio.Media.DATA,
        MediaStore.Audio.Media._ID,
        MediaStore.Audio.Media.TITLE,
        MediaStore.Audio.Media.DISPLAY_NAME,
        MediaStore.Audio.Media.MIME_TYPE,
};
```

Next we need to construct an SQL WHERE clause for our query. Since we want only media files that belong to a specific album, our WHERE clause should indicate that.

In normal SQL, the WHERE clause would look like this:

```
WHERE album = 'album name'
```

Since we are working with a managedQuery, we don't need the word WHERE and we don't need to pass in what it should be equal to. Instead we substitute in a '?'. Therefore for the foregoing version, the String would be as follows:

```
album = ?
```

Since we don't know the actual name of the column as we are working with a constant. We'll use that to construct the WHERE clause.

```
String where = android.provider.MediaStore.Audio.Media.ALBUM + "=?";
```

Finishing off the WHERE clause, we need the data that will be substituted in for the ?s in the WHERE. This will be an array of Strings, one for each of the ?s used. In our case, we want to use the name of the album that was selected. Since we have the Cursor in the right position, we simply need to call the Cursor's getString method on the right column, which we get by calling the Cursor's getColumnIndex method on the column name.

```
String whereVal[] = ↩
{cursor.getString(cursor.getColumnIndex(MediaStore.Audio.Albums.ALBUM))};
```

Last we can specify that we want the results to be ordered by a specific column's value. For this let's create a String variable that will contain the name of the column that we want the results ordered by.

```
String orderBy = android.provider.MediaStore.Audio.Media.TITLE;
```

Finally, we can run our managedQuery method, passing in the Uri, the columns, the WHERE clause variable, the WHERE clause data, and the ORDER BY variable.

```
cursor = managedQuery(MediaStore.Audio.Media.EXTERNAL_CONTENT_URI,↩
columns, where, whereVal, orderBy);
```

Again, we'll use the `ListActivity` methods to manage the `Cursor` object and present the results in a list.

```
                String[] displayFields = new String[]↩
{MediaStore.Audio.Media.DISPLAY_NAME};
                int[] displayViews = new int[] {android.R.id.text1};
                setListAdapter(new SimpleCursorAdapter(this,↩
                        android.R.layout.simple_list_item_1, cursor, displayFields,↩
displayViews));
```

The last thing we'll do is change the `currentState` variable to be `STATE_SELECT_SONG` so that the next time through this method, we skip all of this as the user will be selecting a song and not an album.

```
                currentState = STATE_SELECT_SONG;
            }
        } else if (currentState == STATE_SELECT_SONG) {
```

When the user selects a song from the list after selecting an album, he or she will enter this part of the method as `currentState` will equal `STATE_SELECT_SONG`.

```
            if (cursor.moveToPosition(position)) {
```

Using the same `moveToPosition` call on the `Cursor` object as we did previously, we can get at the song that was actually selected. In this case, we are getting at the column that contains the path to the file and the MIME-type of that file. We are converting it to a `File` and creating an intent to start the built-in music player application.

```
                int fileColumn = cursor.getColumnIndex (MediaStore.Audio.Media.DATA);
                int mimeTypeColumn = cursor.getColumnIndex↩
(MediaStore.Audio.Media.MIME_TYPE);

                String audioFilePath = cursor.getString(fileColumn);
                String mimeType = cursor.getString(mimeTypeColumn);

                Intent intent = new Intent(android.content.Intent.ACTION_VIEW);

                File newFile = new File(audioFilePath);
                intent.setDataAndType(Uri.fromFile(newFile), mimeType);

                startActivity(intent);
            }
        }
    }
}
```

Here is the layout XML file that is being used by the foregoing code. You'll notice that it contains a ListView with the ID `list`. This is the default ID that needs to be used with the ListActivity that we are extending.

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

```
    <ListView android:id="@+android:id/list" android:layout_width="wrap_content"↩
     android:layout_height="wrap_content"></ListView>
</LinearLayout>
```

That wraps up our sample application, which allows us to browse and select songs to play by album using the `MediaStore`. Using very similar methods, we could build it out so that we could browse and select music based upon artist and genre as well.

Summary

As we have discovered throughout this chapter, Android provides a rich set of capabilities for working with audio files. The capabilities it offers are implemented in a manner similar to those we have used with the image capture capabilities; specifically, we can use the built-in applications through an intent or create our own custom playback application. Also, the `MediaStore` has special capabilities for audio beyond querying for individual audio files—we can use it to search and browse for audio based on artist, album, genre, and more.

In the next chapter, we'll take this a step further and look at the world opened up by harnessing audio not stored on the device, but rather available via the Internet.

Chapter 6

Background and Networked Audio

In the last chapter, we explored Android's basic audio playback capabilities. While those capabilities are fantastic, we need to push a bit further to make them generally useful. In this chapter, we'll look at how we can do things like play audio files in the background so that the application playing the audio doesn't need to be running. We'll take a look at how we can synthesize sound rather than just playing sound files, and we'll look at how to leverage streaming audio that is available on the Internet.

Background Audio Playback

So far we have concentrated on building applications that are centered around being in the foreground and have their user interface in front of the user. In the last chapter, we looked at how to add audio playback capabilities to those types of applications.

What happens, though, if we want to build an application that plays music or audio books, but we would like the user to be able to do other things with the phone while continuing to listen? We might have some trouble making that happen if we limit ourselves to just building *activities*. The Android operating system reserves the right to kill activities that aren't in the front and in use by the user. It does this in order to free up memory to make room for other applications to run. If the OS kills an activity that is playing audio, this would stop the audio from playing, making the user experience not so great.

Fortunately, there is a solution. Instead of playing our audio in an activity, we can use a Service.

Services

In order to ensure that the audio continues to play when the application is no longer in the front and its activity is not in use, we need to create a Service. A Service is a

component of an Android application that is meant to run tasks in the background without requiring any interaction from the user.

Local vs. Remote Services

There are a couple of different classes of Services in use by Android. The first and what we'll be exploring is called a Local Service. Local Services exist as part of a specific application and are accessed and controlled only by that application. Remote Services are the other type. They can communicate with, be accessed, and be controlled by other applications. As mentioned, we'll be concentrating on using a Local Service to provide audio playback capabilities. Developing Remote Services is a very large topic and is unfortunately out of the scope of this book.

Simple Local Service

To demonstrate a Service, let's go through this very simple example.

First we'll need an activity with an interface that allows us to start and stop the Service.

```
package com.apress.proandroidmedia.ch06.simpleservice;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class SimpleServiceActivity extends Activity implements OnClickListener {
```

Our activity will have two Buttons—one for starting the Service and one for stopping it.

```
    Button startServiceButton;
    Button stopServiceButton;
```

In order to start or stop the Service, we use a standard intent.

```
    Intent serviceIntent;

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

Our activity implements `OnClickListener`, so we set the `OnClickListener` for each of the Buttons to be "this".

```
        startServiceButton = (Button) this.findViewById(R.id.StartServiceButton);
        stopServiceButton = (Button) this.findViewById(R.id.StopServiceButton);

        startServiceButton.setOnClickListener(this);
        stopServiceButton.setOnClickListener(this);
```

When instantiating the intent that will be used to start and stop the Service, we pass in our activity as the Context followed by the Service's class.

```
        serviceIntent = new Intent(this, SimpleServiceService.class);
    }

    public void onClick(View v) {
        if (v == startServiceButton) {
```

When the `startServiceButton` is clicked, we call the `startService` method, passing in the intent just created to refer to our Service. The `startService` method is a part of the Context class of which activity is a child.

```
            startService(serviceIntent);
        }
        else if (v == stopServiceButton) {
```

When the `stopServiceButton` is clicked, we call the `stopService` method, passing in the same intent that refers to our Service. As with `startService`, `stopService` is part of the Context class.

```
            stopService(serviceIntent);
        }
    }
}
```

Here is our `main.xml` file that defines the layout for the foregoing activity. It contains the StartService and StopService Buttons as well as a TextView.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Simple Service"
    />
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
 android:id="@+id/StartServiceButton" android:text="Start Service"></Button>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
 android:text="Stop Service" android:id="@+id/StopServiceButton"></Button>
</LinearLayout>
```

Now we can move on to the code for the Service itself. In this example, we aren't accomplishing anything in the Service, just using Toast to tell us when the Service has started and stopped.

```
package com.apress.proandroidmedia.ch06.simpleservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
```

Services extend the `android.app.Service` class. The `Service` class is abstract, so in order to extend it, we have to at the very least implement the `onBind` method. In this very

simple example, we aren't going to be "binding" to the Service. Therefore we'll just return `null` in our `onBind` method.

```
public class SimpleServiceService extends Service {

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

The next three methods represent the Service's life cycle. The `onCreate` method, as in the *Activity,* is called when the Service is instantiated. It will be the first method called.

```
    @Override
    public void onCreate() {
        Log.v("SIMPLESERVICE","onCreate");
    }
```

The `onStartCommand` method will be called whenever `startService` is called with an intent that matches this Service. Therefore it may be called more than once. The `onStartCommand` returns an integer value that represents what the OS should do if it kills the Service. `START_STICKY`, which we are using here, indicates that the Service will be restarted if killed.

```
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.v("SIMPLESERVICE","onStartCommand");
        return START_STICKY;
    }
```

onStartCommand vs. onStart

The `onStartCommand` method was introduced with Android 2.0 (API level 5). Previous to that, the method used was `onStart`. `onStart`'s parameters are an intent and an int for `startId`. It does not include the int `flags` parameter and doesn't have a return. If you are targeting a phone that is running something earlier than 2.0, you can use the `onStart` method.

```
    @Override
public void onStart(Intent intent, int startid) {
        Log.v("SIMPLESERVICE","onStart");
    }
```

The `onDestroy` method is called when the OS is destroying a Service. In this example, it is triggered when the `stopService` method is called by our activity. This method should be used to do any cleanup that needs to be done when a Service is being shut down.

```
    public void onDestroy() {
        Log.v("SIMPLESERVICE","onDestroy");
    }
}
```

Finally, in order to make this example work, we need to add an entry to our manifest file (`AndroidManifest.xml`) that specifies our Service.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.apress.proandroidmedia.ch06.simpleservice"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".SimpleServiceActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    <service android:name=".SimpleServiceService" />
    </application>
    <uses-sdk android:minSdkVersion="5" />
</manifest>
```

Of course, this example doesn't do anything other than output to the log indicating when the Service has been started and stopped. Let's move forward and have our Service actually do something.

Local Service plus MediaPlayer

Now that we have created an example Service, we can use it as a template to create an application to play audio files in the background. Here is a Service, and an activity to control the Service that does just that, allows us to play audio files in the background. It works in a similar manner to the custom audio player example from the last chapter, as it is using the same underlying `MediaPlayer` class that Android makes available to us.

```
package com.apress.proandroidmedia.ch06.backgroundaudio;

import android.app.Service;
import android.content.Intent;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;
import android.os.IBinder;
import android.util.Log;
```

The Service implements `OnCompletionListener` so that it can be notified when the `MediaPlayer` has finished playing an audio file.

```
public class BackgroundAudioService extends Service implements OnCompletionListener
{
```

We declare an object of type `MediaPlayer`. This object will handle the playback of the audio as shown in the custom audio player example in the last chapter.

```
    MediaPlayer mediaPlayer;

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

    @Override
```

```
public void onCreate() {
    Log.v("PLAYERSERVICE","onCreate");
```

In the `onCreate` method, we instantiate the `MediaPlayer`. We are passing it a specific reference to an audio file called `goodmorningandroid.mp3`, which should be placed in the raw resources (`res/raw`) directory of our project. If this directory doesn't exist, it should be created. Putting the audio file in that location allows us to refer to it by a constant in the generated `R` class, `R.raw.goodmorningandroid`. More detail about placing audio in the raw resources directory is available in Chapter 5 in the "Creating a Custom Audio-Playing Application" section.

```
    mediaPlayer = MediaPlayer.create(this, R.raw.goodmorningandroid);
```

We also set our Service, this class, to be the `OnCompletionListener` for the `MediaPlayer` object.

```
    mediaPlayer.setOnCompletionListener(this);
}
```

When the `startService` command is issued on this Service, the `onStartCommand` method will be triggered. In this method, we first check that the `MediaPlayer` object isn't already playing, as this method may be called multiple times, and if it isn't, we start it. Since we are using the `onStartCommand` method rather than the `onStart` method, this example runs only in Android 2.0 and above.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Log.v("PLAYERSERVICE","onStartCommand");

    if (!mediaPlayer.isPlaying()) {
        mediaPlayer.start();
    }
    return START_STICKY;
}
```

When the Service is destroyed, the `onDestroy` method is triggered. Since this doesn't guarantee that the `MediaPlayer` will stop playing, we invoke its `stop` method here if it is playing and also call its `release` method to get rid of any memory usage and or resource locks.

```
public void onDestroy() {
    if (mediaPlayer.isPlaying())
    {
        mediaPlayer.stop();
    }
    mediaPlayer.release();
    Log.v("SIMPLESERVICE","onDestroy");
}
```

Because we are implementing `OnCompletionListener` and the Service itself is set to be the `MediaPlayer`'s `OnCompletionListener`, the following `onCompletion` method is called when the `MediaPlayer` has finished playing an audio file. Since this Service is only meant to play one song and that's it, we call `stopSelf`, which is analogous to calling `stopService` in our activity.

```
    public void onCompletion(MediaPlayer _mediaPlayer) {
        stopSelf();
    }
}
```

Here is the activity that corresponds to the foregoing Service. It has a very simple interface with two buttons to control the starting and stopping of the Service. In each case, it calls finish directly after to illustrate that the Service is not dependent on the activity and runs independently.

```
package com.apress.proandroidmedia.ch06.backgroundaudio;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class BackgroundAudioActivity extends Activity implements OnClickListener {

    Button startPlaybackButton, stopPlaybackButton;
    Intent playbackServiceIntent;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        startPlaybackButton = (Button) this.findViewById(R.id.StartPlaybackButton);
        stopPlaybackButton = (Button) this.findViewById(R.id.StopPlaybackButton);

        startPlaybackButton.setOnClickListener(this);
        stopPlaybackButton.setOnClickListener(this);

        playbackServiceIntent = new Intent(this,BackgroundAudioService.class);
    }

    public void onClick(View v) {
        if (v == startPlaybackButton) {
            startService(playbackServiceIntent);
            finish();
        } else if (v == stopPlaybackButton) {
            stopService(playbackServiceIntent);
            finish();
        }
    }
}
```

Here is the `main.xml` layout XML file in use by the foregoing activity.

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

```
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Background Audio Player"
        />
    <Button android:text="Start Playback" android:id="@+id/StartPlaybackButton"↩
     android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
    <Button android:text="Stop Playback" android:id="@+id/StopPlaybackButton"↩
     android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
</LinearLayout>
```

Finally, here is the `AndroidManifest.xml` for this project.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.apress.proandroidmedia.ch06.backgroundaudio"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".BackgroundAudioActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".BackgroundAudioService" />
    </application>
    <uses-sdk android:minSdkVersion="5" />
</manifest>
```

As shown, simply using the `MediaPlayer` to start and stop audio playback within a Service is very straightforward. Let's now look at how we can take that a step further.

Controlling a MediaPlayer in a Service

Unfortunately, when using a Service, issuing commands to the MediaPlayer from the user-facing activity becomes more complicated.

In order to allow the `MediaPlayer` to be controlled, we need to bind the activity and Service together. Once we do that, since the activity and Service are running in the same process, we can call methods in the Service directly. If we were creating a remote Service, we would have to take further steps.

Let's add a Button labeled "Have Fun" to the foregoing activity. When this Button is pressed, we'll have the `MediaPlayer` seek back a few seconds and continuing playing the audio file.

We'll start by adding the Button to the layout XML:

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

```
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Background Audio Player"
        />
    <Button android:text="Start Playback" android:id="@+id/StartPlaybackButton"↩
     android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
    <Button android:text="Stop Playback" android:id="@+id/StopPlaybackButton"↩
     android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
    <Button android:text="Have Fun" android:id="@+id/HaveFunButton"↩
     android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
</LinearLayout>
```

Then in the activity, we'll get a reference to it and set its `onClickListener` to be the activity itself, just like the existing Buttons.

```
package com.apress.proandroidmedia.ch06.backgroundaudiobind;

import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class BackgroundAudioActivity extends Activity implements OnClickListener {

    Button startPlaybackButton, stopPlaybackButton;
    Button haveFunButton;
    Intent playbackServiceIntent;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        startPlaybackButton = (Button) this.findViewById(R.id.StartPlaybackButton);
        stopPlaybackButton = (Button) this.findViewById(R.id.StopPlaybackButton);
        haveFunButton = (Button) this.findViewById(R.id.HaveFunButton);

        startPlaybackButton.setOnClickListener(this);
        stopPlaybackButton.setOnClickListener(this);
        haveFunButton.setOnClickListener(this);

        playbackServiceIntent = new Intent(this,BackgroundAudioService.class);
    }
```

In order for us to have this Button interact with the `MediaPlayer` that is running in the Service, we have to bind to the Service. The means to do this is with the `bindService` method. This method takes in an intent; in fact, we can re-use the `playbackServiceIntent` that we are using to start the Service, a `ServiceConnection` object, and some flags for what to do when the Service isn't running.

In the following onClick method in our activity, we bind to the Service right after we start it, when the startPlaybackButton is pressed. We unbind from the Service when the stopPlaybackButton is called.

```
    public void onClick(View v) {
        if (v == startPlaybackButton) {
            startService(playbackServiceIntent);
            bindService(playbackServiceIntent, serviceConnection,
                        Context.BIND_AUTO_CREATE);
        } else if (v == stopPlaybackButton) {
            unbindService(serviceConnection);
            stopService(playbackServiceIntent);
        }
```

You'll probably notice that we are using a new object that we haven't defined, serviceConnection. This we'll take care of in a moment.

We also need to finish off the onClick method. Since our new Button has its onClickListener set to be the activity, we should handle that case as well and close out the onClick method.

```
        else if (v == haveFunButton) {
            baService.haveFun();
        }
    }
```

In the new section, we are using another new object, baService. baService is an object that is of type BackgroundAudioService. We'll declare it now and take care of creating when we create our ServiceConnection object.

```
    private BackgroundAudioService baService;
```

As mentioned, we are still missing the declaration and instantiation of an object called serviceConnection. serviceConnection will be an object of type ServiceConnection, which is an Interface for monitoring the state of a bound Service.

Let's take care of creating our serviceConnection now:

```
    private ServiceConnection serviceConnection = new ServiceConnection() {
```

The onServiceConnected method shown here will be called when a connection with the Service has been established through a bindService command that names this object as the ServiceConnection (such as we are doing in our bindService call).

One thing that is passed into this method is an IBinder object that is actually created and delivered from the Service itself. In our case, this IBinder object will be of type BackgroundAudioServiceBinder, which we'll create in our Service. It will have a method that returns our Service itself, called getService. The object returned by this can be operated directly on, as we are doing when the haveFunButton is clicked.

```
        public void onServiceConnected(ComponentName className, IBinder baBinder) {
            baService =
((BackgroundAudioService.BackgroundAudioServiceBinder)baBinder).getService();
        }
```

We also need an `onServiceDisconnected` method to handle cases when the Service goes away.

```
        public void onServiceDisconnected(ComponentName className) {
            baService = null;
        }
    };
}
```

.

Now we can turn our attention to what we need to change in the Service itself.

```
package com.apress.proandroidmedia.ch06.backgroundaudiobind;

import android.app.Service;
import android.content.Intent;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;
import android.os.Binder;
import android.os.IBinder;
import android.util.Log;

public class BackgroundAudioService extends Service implements OnCompletionListener
{
    MediaPlayer mediaPlayer;
```

The first change that we'll need to make in our Service is to create an inner class that extends `Binder` that can return our Service itself when asked.

```
    public class BackgroundAudioServiceBinder extends Binder {
        BackgroundAudioService getService() {
            return BackgroundAudioService.this;
        }
    }
```

Following that, we'll instantiate that as an object called `basBinder`.

```
    private final IBinder basBinder = new BackgroundAudioServiceBinder();
```

And override the implementation of `onBind` to return that.

```
    @Override
    public IBinder onBind(Intent intent) {
        // Return the BackgroundAudioServiceBinder object
        return basBinder;
    }
```

That's it for the binding. Now we just need to deal with "Having Fun."

As mentioned, when the `haveFunButton` is clicked, we want the `MediaPlayer` to seek back a few seconds. In this implementation, it will seek back 2,500 milliseconds or 2.5 seconds.

```
        public void haveFun() {
        if (mediaPlayer.isPlaying()) {
            mediaPlayer.seekTo(mediaPlayer.getCurrentPosition() - 2500);
        }
    }
```

That's it for updates on the Service. Here is the rest of the code for good measure.

```
    @Override
    public void onCreate() {
        Log.v("PLAYERSERVICE","onCreate");

        mediaPlayer = MediaPlayer.create(this, R.raw.goodmorningandroid);
        mediaPlayer.setOnCompletionListener(this);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.v("PLAYERSERVICE","onStartCommand");

        if (!mediaPlayer.isPlaying()) {
            mediaPlayer.start();
        }
        return START_STICKY;
    }

    public void onDestroy() {
        if (mediaPlayer.isPlaying())
        {
            mediaPlayer.stop();
        }
        mediaPlayer.release();
        Log.v("SIMPLESERVICE","onDestroy");
    }

    public void onCompletion(MediaPlayer _mediaPlayer) {
        stopSelf();
    }

}
```

Finally, here is the `AndroidManifest.xml` file that is required by the foregoing example.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      android:versionCode="1"
      android:versionName="1.0" package="com.apress.proandroidmedia↩
.ch06.backgroundaudiobind">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".BackgroundAudioActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".BackgroundAudioService" />
    </application>
    <uses-sdk android:minSdkVersion="5" />
</manifest>
```

Now that the foundation is in place, we can add whatever functionality we like into the Service and call the various methods such as `haveFun` directly from our activity by

binding to it. Without binding to the Service, we would be unable to do anything more than start and stop the Service.

The foregoing examples should give a good starting point for building an application that plays audio files in the background, allowing users to continue doing other tasks while the audio continues playing. The second example can be extended to build a full-featured audio playback application.

Networked Audio

Moving our attention forward, let's look at how we can further leverage Android's audio playback capabilities to harness media that lives elsewhere, in particular audio that lives online. With posting MP3 files, podcasting, and streaming all becoming more and more popular, it only makes sense that we would want to build audio playback applications that can leverage those services.

Fortunately, Android has rich capabilities for dealing with various types of audio available on the network.

Let's start with examining how to leverage web-based audio or audio delivered via HTTP.

HTTP Audio Playback

The simplest case to explore would simply be to play an audio file that lives online and is accessible via HTTP.

One such file would be this, which is available on my server:
`http://www.mobvcasting.com/android/audio/goodmorningandroid.mp3`

Here is an example activity that uses the `MediaPlayer` to illustrate how to play audio available via HTTP.

```
package com.apress.proandroidmedia.ch06.audiohttp;

import java.io.IOException;

import android.app.Activity;
import android.media.MediaPlayer;
import android.os.Bundle;
import android.util.Log;

public class AudioHTTPPlayer extends Activity {
    MediaPlayer mediaPlayer;

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

When our activity is created, we do a generic instantiation of a `MediaPlayer` object by calling the `MediaPlayer` constructor with no arguments. This is a different way of using

the MediaPlayer than we have previously seen and requires us to take some additional steps before we can play the audio.

```
        mediaPlayer = new MediaPlayer();
```

Specifically, we need to call the setDataSource method, passing in the HTTP location of the audio file we would like to play. This method can throw an IOException, so we have to catch and deal with that as well.

```
        try {
            mediaPlayer.setDataSource(
                "http://www.mobvcasting.com/android/audio/goodmorningandroid.mp3");
```

Following that we call the prepare method and then the start method, after which the audio should start playing.

```
            mediaPlayer.prepare();
            mediaPlayer.start();
        } catch (IOException e) {
            Log.v("AUDIOHTTPPLAYER",e.getMessage());
        }
    }
}
```

Running this example, you will probably notice a significant lag time from when the application loads to when the audio plays. The length of the delay is due to the speed of the data network that the phone is using for its Internet connection (among other variables).

If we add Log or Toast messages throughout the code, we would see that this delay happens between the call to the prepare method and the start method. During the running of the prepare method, the MediaPlayer is filling up a buffer so that the audio playback can run smoothly even if the network is slow.

The prepare method actually blocks while it is doing this. This means that applications that use this method will likely become unresponsive until the prepare method is complete. Fortunately, there is a way around this, and that is to use the prepareAsync method. This method returns immediately and does the buffering and other work in the background, allowing the application to continue.

The issue then becomes one of paying attention to the state of the MediaPlayer object and implementing various callbacks that help us keep track of its state.

To get a handle on the various states that a MediaPlayer object may be in, it is helpful to look over the diagram from the MediaPlayer page on the Android API Reference, shown in Figure 6–1.

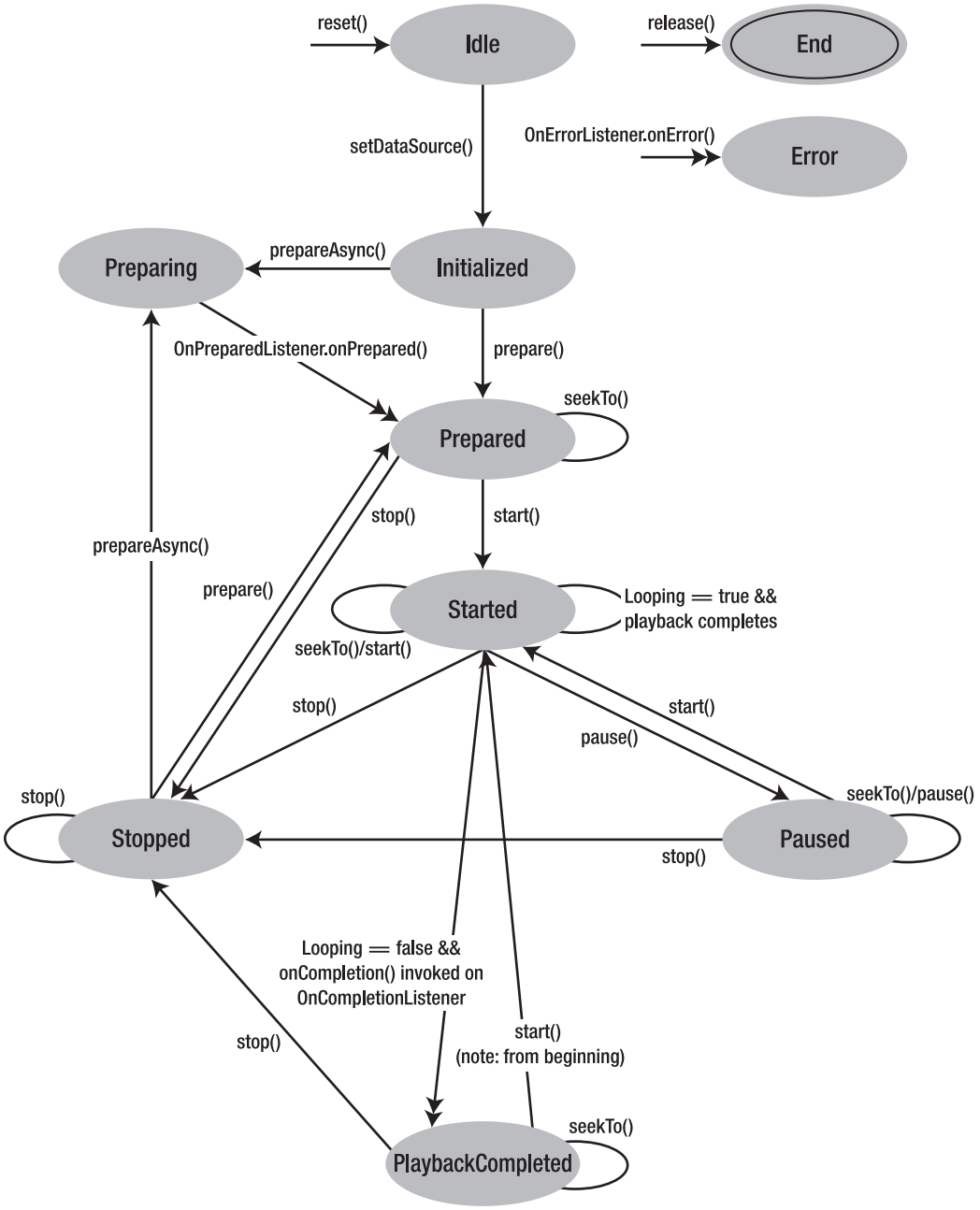


Figure 6–1. *MediaPlayer state diagram from Android API Reference*

Here is a full `MediaPlayer` example that uses `prepareAsync` and implements several listeners to keep track of its state.

```
package com.apress.proandroidmedia.ch06.audiohttpasync;

import java.io.IOException;
import android.app.Activity;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnBufferingUpdateListener;
import android.media.MediaPlayer.OnCompletionListener;
import android.media.MediaPlayer.OnErrorListener;
import android.media.MediaPlayer.OnInfoListener;
import android.media.MediaPlayer.OnPreparedListener;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
```

In this version of our HTTP audio player, we are implementing several interfaces. Two of them, `OnPreparedListener` and `OnCompletionListener`, will help us keep track of the state of the `MediaPlayer` so that we don't attempt to play or stop audio when we shouldn't.

```
public class AudioHTTPPlayer extends Activity
    implements OnClickListener, OnErrorListener, OnCompletionListener,
    OnBufferingUpdateListener, OnPreparedListener {

    MediaPlayer mediaPlayer;
```

The interface for this activity has start and stop Buttons, a TextView for displaying the status, and a TextView for displaying the percentage of the buffer that has been filled.

```
    Button stopButton, startButton;
    TextView statusTextView, bufferValueTextView;

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

In the `onCreate` method, we set the `stopButton` and `startButton` to be disabled. They'll be enabled or disabled throughout the running of the application. This is to illustrate when the methods they trigger are and aren't available.

```
        stopButton = (Button) this.findViewById(R.id.EndButton);
        startButton = (Button) this.findViewById(R.id.StartButton);
        stopButton.setOnClickListener(this);
        startButton.setOnClickListener(this);
        stopButton.setEnabled(false);
        startButton.setEnabled(false);

        bufferValueTextView = (TextView) this.findViewById(R.id.BufferValueTextView);
        statusTextView = (TextView) this.findViewById(R.id.StatusDisplayTextView);
        statusTextView.setText("onCreate");
```

After we instantiate the `MediaPlayer` object, we register the activity to be the `OnCompletionListener`, the `OnErrorListener`, the `OnBufferingUpdateListener`, and the `OnPreparedListener`.

```
        mediaPlayer = new MediaPlayer();

        mediaPlayer.setOnCompletionListener(this);
        mediaPlayer.setOnErrorListener(this);
        mediaPlayer.setOnBufferingUpdateListener(this);
        mediaPlayer.setOnPreparedListener(this);

        statusTextView.setText("MediaPlayer created");
```

Next we call `setDataSource` with the URL to the audio file.

```
        try {
            mediaPlayer.setDataSource(
                "http://www.mobvcasting.com/android/audio/goodmorningandroid.mp3");

            statusTextView.setText("setDataSource done");
            statusTextView.setText("calling prepareAsync");
```

Last, we'll call `prepareAsync`, which will start the buffering of the audio file in the background and return. When the preparation is complete, our activity's `onCompletion` method will be called due to our activity being registered as the `OnCompletionListener` for the `MediaPlayer`.

```
            mediaPlayer.prepareAsync();

        } catch (IOException e) {
            Log.v("AUDIOHTTPPLAYER",e.getMessage());
        }
    }
```

What follows is the implementation of the `onClick` method for the two Buttons. When the `stopButton` is pressed, the `MediaPlayer`'s `pause` method will be called. When the `startButton` is pressed, the `MediaPlayer`'s `start` method is called.

```
    public void onClick(View v) {
        if (v == stopButton) {
            mediaPlayer.pause();
            statusTextView.setText("pause called");
            startButton.setEnabled(true);
        } else if (v == startButton) {
            mediaPlayer.start();
            statusTextView.setText("start called");
            startButton.setEnabled(false);
            stopButton.setEnabled(true);
        }
    }
```

If the `MediaPlayer` enters into an error state, the `onError` method will be called on the object that is registered as the `MediaPlayer`'s `OnErrorListener`. The following `onError` method shows the various constants that are specified in the `MediaPlayer` class.

```
    public boolean onError(MediaPlayer mp, int what, int extra) {
        statusTextView.setText("onError called");

        switch (what) {
            case MediaPlayer.MEDIA_ERROR_NOT_VALID_FOR_PROGRESSIVE_PLAYBACK:
                statusTextView.setText(
                    "MEDIA ERROR NOT VALID FOR PROGRESSIVE PLAYBACK " + extra);
```

```
                Log.v(
                    "ERROR","MEDIA ERROR NOT VALID FOR PROGRESSIVE PLAYBACK " + extra);
                break;
            case MediaPlayer.MEDIA_ERROR_SERVER_DIED:
                statusTextView.setText("MEDIA ERROR SERVER DIED " + extra);
                Log.v("ERROR","MEDIA ERROR SERVER DIED " + extra);
                break;
            case MediaPlayer.MEDIA_ERROR_UNKNOWN:
                statusTextView.setText("MEDIA ERROR UNKNOWN " + extra);
                Log.v("ERROR","MEDIA ERROR UNKNOWN " + extra);
                break;
        }

        return false;
    }
```

When the `MediaPlayer` completes playback of an audio file, the `onCompletion` method of the object registered as the `OnCompletionListener` will be called. This indicates that we could start playback again.

```
    public void onCompletion(MediaPlayer mp) {
        statusTextView.setText("onCompletion called");
        stopButton.setEnabled(false);
        startButton.setEnabled(true);
    }
```

While the `MediaPlayer` is buffering, the `onBufferingUpdate` method of the object registered as the `MediaPlayer`'s `onBufferingUpdateListener` is called. The percentage of the buffer that is filled is passed in.

```
    public void onBufferingUpdate(MediaPlayer mp, int percent) {
        bufferValueTextView.setText("" + percent + "%");
    }
```

When the `prepareAsync` method finishes, the `onPrepared` method of the object registered as the `OnPreparedListener` will be called. This indicates that the audio is ready for playback, and therefore, in the following method, we are setting the `startButton` to be enabled.

```
    public void onPrepared(MediaPlayer mp) {
        statusTextView.setText("onPrepared called");
        startButton.setEnabled(true);
    }
}
```

Here is the Layout XML associated with the foregoing activity:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView android:text="Status" android:id="@+id/TextView01"↩
 android:layout_width="wrap_content" android:layout_height="wrap_content"></TextView>
    <TextView android:text="Unknown" android:id="@+id/StatusDisplayTextView"↩
 android:layout_width="wrap_content" android:layout_height="wrap_content"></TextView>
    <TextView android:text="0%" android:id="@+id/BufferValueTextView"↩
```

```
 android:layout_width="wrap_content" android:layout_height="wrap_content"></TextView>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
 android:text="Start" android:id="@+id/StartButton"></Button>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
 android:id="@+id/EndButton" android:text="Stop"></Button>
</LinearLayout>
```

As just shown, the `MediaPlayer` has a nice set of capabilities for handling audio files that are available online via HTTP.

Streaming Audio via HTTP

One way that live audio is commonly delivered online is via HTTP streaming. There are a variety of streaming methods that fall under the umbrella of HTTP streaming from server push, which has historically been used for displaying continually refreshing webcam images in browsers to a series of new methods being put forth by Apple, Adobe, and Microsoft for use by their respective media playback applications.

The main method for streaming live audio over HTTP is one developed in 1999 by a company called Nullsoft, which was subsequently purchased by AOL. Nullsoft was the creator of WinAMP, a popular MP3 player, and they developed a live audio streaming server that used HTTP, called SHOUTcast. SHOUTcast uses the ICY protocol, which extends HTTP. Currently, a large number of servers and playback software products support this protocol, so many, in fact, that it may be considered the de facto standard for online radio.

Fortunately, the `MediaPlayer` class on Android is capable of playing ICY streams without requiring us developers to jump through hoops.

Unfortunately for us, Internet radio stations don't typically advertise the direct URL to their streams. This is for good reason; unfortunately, browsers don't support ICY streams directly and require a helper application or plug-in to play the stream. In order to know to open a helper application, an intermediary file with a specific MIME-type is delivered by the Internet radio station, which contains a pointer to the actual live stream. In the case of ICY streams, this is typically either a PLS file or an M3U file.

- A PLS file is a multimedia playlist file and has the MIME-type "`audio/x-scpls`".
- An M3U file is also a file that stores multimedia playlists but in a more basic format. Its MIME-type is "`audio/x-mpegurl`".

The following illustrates the contents of an M3U file that points to a fake live stream.

```
#EXTM3U
#EXTINF:0,Live Stream Name
http://www.nostreamhere.org:8000/
```

The first line, `#EXTM3U`, is required and specifies that what follows is an Extended M3U file that can contain extra information. Extra information about a playlist entry is specified on the line above the entry and starts with `#EXTINF:`, followed by the duration in seconds, a comma, and then the name of the media.

An M3U file can have multiple entries as well, specifying one file or stream after another.

```
#EXTM3U
#EXTINF:0,Live Stream Name
http://www.nostreamhere.org:8000/
#EXTINF:0,Other Live Stream Name
http://www.nostreamthere.org/
```

Unfortunately, the `MediaPlayer` on Android doesn't handle the parsing of M3U files for us. Therefore, to create an HTTP streaming audio player on Android, we have to handle the parsing ourselves and use the `MediaPlayer` for the actual media playback.

Here is an example activity that parses and plays an M3U file delivered from an online radio station or any M3U file as entered in the URL field.

```
package  com.apress.proandroidmedia.ch06.httpaudioplaylistplayer;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Vector;

import org.apache.http.HttpResponse;
import org.apache.http.HttpStatus;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;

import android.app.Activity;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;
import android.media.MediaPlayer.OnPreparedListener;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
```

As in our previous example, this activity will extend `OnCompletionListener` and `OnPreparedListener` to track the state of the `MediaPlayer`.

```
public class HTTPAudioPlaylistPlayer extends Activity
  implements OnClickListener, OnCompletionListener, OnPreparedListener {
```

We'll use a `Vector` to hold the list of items in the playlist. Each item will be a `PlaylistFile` object that is defined in an inner class at the end of this class.

```
    Vector playlistItems;
```

We'll have a few Buttons on the interface as well as an `EditText` object, which will contain the URL to the M3U file.

```
    Button parseButton;
    Button playButton;
    Button stopButton;
```

```
EditText editTextUrl;
String baseURL = "";
MediaPlayer mediaPlayer;
```

The following integer is used keep track of which item from the `playlistItems Vector` we are currently on.

```
int currentPlaylistItemNumber = 0;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    parseButton = (Button) this.findViewById(R.id.ButtonParse);
    playButton = (Button) this.findViewById(R.id.PlayButton);
    stopButton = (Button) this.findViewById(R.id.StopButton);
```

We are setting the text of the `editTextUrl` object to be the URL of an M3U file from an online radio station. The first one, which is commented out, is the URL for KBOO, a community radio station in Portland, Oregon (`www.kboo.fm/`). The second, which is not commented out, is for KMFA, a classical station in Austin, Texas (`www.kmfa.org/`).

The user can edit this to be the URL to any M3U file available on the Internet.

```
    editTextUrl = (EditText) this.findViewById(R.id.EditTextURL);
    //editTextUrl.setText("http://live.kboo.fm:8000/high.m3u");
    editTextUrl.setText("http://pubint.ic.llnwd.net/stream/pubint_kmfa.m3u");

    parseButton.setOnClickListener(this);
    playButton.setOnClickListener(this);
    stopButton.setOnClickListener(this);
```

Initially the `playButton` and `stopButton` will not be enabled; the user will not be able to press them. The `parseButton`, on the other hand, will be enabled. After the M3U file is retrieved and parsed, the `playButton` will be enabled, and after the audio is playing, the `stopButton` will be enabled. This is how we'll guide the user through the flow of the application.

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

    mediaPlayer = new MediaPlayer();
    mediaPlayer.setOnCompletionListener(this);
    mediaPlayer.setOnPreparedListener(this);
}
```

Each of the Buttons has their `OnClickListener` set to be this activity. Therefore the following `onClick` method will be called when any of these are clicked. This drives most of the application's flow.

When the `parseButton` is pressed, the `parsePlaylistFile` method is called. When the `playButton` is pressed, the `playPlaylistItems` method is called.

```
public void onClick(View view) {
    if (view == parseButton) {
        parsePlaylistFile();
    } else if (view == playButton) {
        playPlaylistItems();
    } else if (view == stopButton) {
        stop();
    }
}
```

The first method that will be triggered is `parsePlaylistFile`. This method downloads the M3U file that is specified by the URL in the `editTextUrl` object and parses it. The act of parsing it picks out any lines that represent files to play and creates a `PlaylistItem` object, which is added to the `playlistItems Vector`.

```
private void parsePlaylistFile() {
```

We'll start out with an empty `Vector`. If a new M3U file is parsed, anything previously in here will be thrown away.

```
    playlistItems = new Vector();
```

To retrieve the M3U file off of the Web, we can use the Apache Software Foundation's `HttpClient` library, which is included with Android.

First we create an `HttpClient` object, which represents something along the lines of a web browser, and then an `HttpGet` object, which represents the specific request for a file. The `HttpClient` will execute the `HttpGet` and return an `HttpResponse`.

```
    HttpClient httpClient = new DefaultHttpClient();
    HttpGet getRequest = new HttpGet(editTextUrl.getText().toString());

    Log.v("URI",getRequest.getURI().toString());

    try {
        HttpResponse httpResponse = httpClient.execute(getRequest);
        if (httpResponse.getStatusLine().getStatusCode() != HttpStatus.SC_OK) {
            // ERROR MESSAGE
            Log.v("HTTP ERROR",httpResponse.getStatusLine().getReasonPhrase());
        } else {
```

After we make the request, we can retrieve an `InputStream` from the `HttpResponse`. This `InputStream` contains the contents of the file requested.

```
            InputStream inputStream = httpResponse.getEntity().getContent();
            BufferedReader bufferedReader =
              new BufferedReader(new InputStreamReader(inputStream));
```

With the aid of a `BufferedReader`, we can go through the file line by line.

```
            String line;
            while ((line = bufferedReader.readLine()) != null) {
                Log.v("PLAYLISTLINE","ORIG: " + line);
```

If the line starts with a "#", we'll ignore it for now. As described earlier, these lines are metadata.

```
                if (line.startsWith("#")) {
                    // Metadata
                    // Could do more with this but not fo now
```

Otherwise, if it isn't a blank line, it has a length greater than 0, and we'll assume that it is a playlist item.

```
                } else if (line.length() > 0) {
```

If the line starts with "http://", we treat it as a full URL to the stream, otherwise we treat it as a relative URL and tack on the URL of the original request for the M3U file.

```
                    String filePath = "";

                    if (line.startsWith("http://")) {
                        // Assume it's a full URL
                        filePath = line;
                    } else {
                        // Assume it's relative
                        filePath = getRequest.getURI().resolve(line).toString();
                    }
```

We then add it to our `Vector` of playlist items.

```
                    PlaylistFile playlistFile = new PlaylistFile(filePath);
                    playlistItems.add(playlistFile);
                }
            }
            inputStream.close();
        }
    } catch (ClientProtocolException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
```

Last, now that we are done parsing the file, we enable the `playButton`.

```
    playButton.setEnabled(true);
}
```

When the user clicks the `playButton`, the `playPlaylistItems` method is called. This method takes the first item from the `playlistItems` `Vector` and hands it to the `MediaPlayer` object for preparation.

```
private void playPlaylistItems() {
    playButton.setEnabled(false);

    currentPlaylistItemNumber = 0;
    if (playlistItems.size() > 0)
    {
        String path =
         ((PlaylistFile)playlistItems.get(currentPlaylistItemNumber)).getFilePath();
        try {
```

After extracting the path to the file or stream, we use that in a `setDataSource` method call on the `MediaPlayer`.

```
            mediaPlayer.setDataSource(path);
```

Then we call prepareAsync, which allows the MediaPlayer to buffer and prepare to play the audio in the background. When the buffering and other preparation is done, the activity's onPrepared method will be called since the activity is registered as the OnPreparedListener.

```
                mediaPlayer.prepareAsync();

            } catch (IllegalArgumentException e) {
                e.printStackTrace();
            } catch (IllegalStateException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Once the onPrepared method has been called, the stopButton is enabled and the MediaPlayer object is triggered to start playing the audio.

```
    public void onPrepared(MediaPlayer _mediaPlayer) {
        stopButton.setEnabled(true);
        Log.v("HTTPAUDIOPLAYLIST","Playing");
        mediaPlayer.start();
    }
```

When the audio playback is complete, the onCompletion method is triggered in this activity since the activity extends and is registered as the MediaPlayer's OnCompletionListener.

The onCompletion method cues up the next item in the playlistItems Vector.

```
    public void onCompletion(MediaPlayer mediaPlayer) {
        Log.v("ONCOMPLETE","called");
        mediaPlayer.stop();
        mediaPlayer.reset();
        if (playlistItems.size() > currentPlaylistItemNumber + 1) {
            currentPlaylistItemNumber++;
            String path =
             ((PlaylistFile)playlistItems.get(currentPlaylistItemNumber)).getFilePath();
            try {
                mediaPlayer.setDataSource(path);
                mediaPlayer.prepareAsync();
            } catch (IllegalArgumentException e) {
                e.printStackTrace();
            } catch (IllegalStateException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

The stop method is called when the user presses the stopButton. This method causes the MediaPlayer to pause rather than stop. The MediaPlayer has a stop method, but that puts the MediaPlayer in an unprepared state. The pause method just pauses playback.

```
    private void stop() {
        mediaPlayer.pause();
        playButton.setEnabled(true);
        stopButton.setEnabled(false);
    }
```

Last, we have an inner class called `PlaylistFile`. One `PlaylistFile` object is created for each file represented in the M3U file.

```
    class PlaylistFile {
        String filePath;

        public PlaylistFile(String _filePath) {
            filePath = _filePath;
        }

        public void setFilePath(String _filePath) {
            filePath = _filePath;
        }

        public String getFilePath() {
            return filePath;
        }
    }
}
```

Here is the layout XML file (`main.xml`) for the foregoing activity.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView android:layout_width="wrap_content" android:layout_height="wrap_content"↩
     android:text="Enter URL" android:id="@+id/EnterURLTextView"></TextView>
    <EditText android:layout_width="wrap_content" android:layout_height="wrap_content"↩
     android:id="@+id/EditTextURL" android:text="http://www.mobvcasting.com/android/↩
    audio/test.m3u"></EditText>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
     android:id="@+id/ButtonParse" android:text="Parse"></Button>
    <TextView android:layout_width="wrap_content" android:layout_height="wrap_content"↩
     android:id="@+id/PlaylistTextView"></TextView>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
     android:id="@+id/PlayButton" android:text="Play"></Button>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
     android:id="@+id/StopButton" android:text="Stop"></Button>
</LinearLayout>
```

This example requires that the following permission be added to the `AndroidManifest.xml` file

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

As you can see in the foregoing example, working with a live audio stream via HTTP is as straightforward as working with a file delivered via HTTP. Figure 6–2 shows the example in action.

Figure 6–2. *HTTP Audio Playlist Player example shown after audio started playback*

RTSP Audio Streaming

Android supports one more protocol for streaming audio through the `MediaPlayer`. This is called the Real Time Streaming Protocol or RTSP. RTSP has been in use for quite some time and was made popular in the mid- to late 1990s by RealNetworks, as it is the protocol they used in their audio and video streaming software.

The same code in use for the preceding HTTP streaming example works with an RTSP audio stream. We'll get into more RTSP specifics in Chapter 10.

Summary

As we have seen throughout this chapter, Android's rich advanced audio capabilities help it to move beyond just being a playback device. Out of the box, it has capabilities that allow us as developers to take advantage of the wide variety of audio available online, from individual MP3 files to live radio streams.

In the next chapter, we'll look at using Android as an audio production device as well.

Chapter 7

Audio Capture

Developing audio playback applications isn't the only way to work with audio on Android. We can also write applications that involve capturing audio. In this chapter, we'll explore three different methods that can be used for audio capture. Each method has relative strengths and weaknesses. The first method, using an intent, is the easiest but least flexible, followed by a method using the `MediaRecorder` class, which is a bit harder to use but offers more flexibility. The final method uses the `AudioRecord` class, and offers the most flexibility but does the least amount of work for us.

Audio Capture with an Intent

The easiest way to simply allow audio recording capabilities in an application is to leverage an existing application through an intent that provides recording capabilities. In the case of audio, Android ships with a sound recorder application that can be triggered in this manner.

The action used to create the intent is available as a constant called `RECORD_SOUND_ACTION` within the `MediaStore.Audio.Media` class. Here is the basic code to trigger the built-in sound recorder.

```
Intent intent = new Intent(MediaStore.Audio.Media.RECORD_SOUND_ACTION);
startActivity(intent);
```

Figure 7–1 shows the built-in audio recording application as triggered by an intent.

Figure 7–1. *Android's built-in sound recorder triggered with an intent*

Of course, in order to retrieve the recording that the user creates, we'll want to use `startActivityForResult` rather than just `startActivity`.

Here is an example that triggers the built-in sound recorder via an intent. When the user finishes, the audio file is played back using the `MediaPlayer`.

```
package  com.apress.proandroidmedia.ch07.intentaudiorecord;

import android.app.Activity;
import android.content.Intent;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;
import android.net.Uri;
import android.os.Bundle;
import android.provider.MediaStore;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
```

Our activity implements `OnClickListener` so that it can respond to Button presses, and `OnCompletionListener` so that it can be notified when audio is finished playing in the `MediaPlayer`.

```
public class IntentAudioRecorder extends Activity implements OnClickListener,↩
 OnCompletionListener {
```

We'll create a constant called RECORD_REQUEST that we can pass into the startActivityForResult function so that we can identify the source of any calls to onActivityResult, which is called when the sound recorder is complete.

The onActivityResult method would be called by any returning activity when triggered by a startActivityForResult function. Passing in a unique constant along with the intent allows us to differentiate between them within the onActivityResult method.

```
public static int RECORD_REQUEST = 0;

Button createRecording, playRecording;
```

We need a Uri object that we will use to contain the Uri to the audio file that is recorded by the sound recorder activity.

```
Uri audioFileUri;

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

After we set the content view, we can obtain references to the Button objects. Each one's click listener is set to this so that our activity's onClick method will be called. Also, we'll set the playRecording button to not be enabled until we have an audio file to play (get a result from the sound recorder activity).

```
        createRecording = (Button) this.findViewById(R.id.RecordButton);
        createRecording.setOnClickListener(this);

        playRecording = (Button) this.findViewById(R.id.PlayButton);
        playRecording.setOnClickListener(this);
        playRecording.setEnabled(false);
    }
```

When we click either of the Buttons, our onClick method is triggered. If the createRecording Button was clicked, we trigger the sound recorder activity through the MediaStore.Audio.Media.RECORD_SOUND_ACTION action in an intent passed into startActivityForResult.

```
    public void onClick(View v) {
        if (v == createRecording) {
            Intent intent =
            new Intent(MediaStore.Audio.Media.RECORD_SOUND_ACTION);
            startActivityForResult(intent, RECORD_REQUEST);
        } else if (v == playRecording) {
```

If the playRecording Button was pressed, we create a MediaPlayer and set it to play the audio file represented by the Uri that was returned from the sound recorder activity and saved in our audioFileUri object.

We also set the OnCompletionListener of the MediaPlayer to be ourselves so that our OnCompletion method is called when it is done playing, and we disable the playRecording Button so that the user cannot trigger the playback to happen again until we are ready.

```
            MediaPlayer mediaPlayer = MediaPlayer.create(this, audioFileUri);
            mediaPlayer.setOnCompletionListener(this);
            mediaPlayer.start();
            playRecording.setEnabled(false);
        }
    }
```

Our `onActivityResult` method is triggered when the sound recorder activity is complete. The `resultCode` should equal the `RESULT_OK` constant, and the `requestCode` should equal the value we passed into the `startActivityForResult` method, `RECORD_REQUEST`. If both of those are true, then we can retrieve the Uri of the recorded audio file by getting it out of the intent that is passed back to us through its `getData` method. Once all of that is done, we enable the `playRecording` Button so that we can play the returned audio file.

```
    protected void onActivityResult (int requestCode, int resultCode, Intent data) {
        if (resultCode == RESULT_OK && requestCode == RECORD_REQUEST) {
            audioFileUri = data.getData();
            playRecording.setEnabled(true);
        }
    }
```

Finally, in our `onCompletion` method, which is called when the `MediaPlayer` is done playing a file, we re-enable the `playRecording` Button so the user may listen to the audio file again if he or she so chooses.

```
    public void onCompletion(MediaPlayer mp) {
        playRecording.setEnabled(true);
    }
}
```

Here is the layout XML that is in use by our activity.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button android:text="Record Audio" android:id="@+id/RecordButton"↩
     android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
    <Button android:text="Play Recording" android:id="@+id/PlayButton"↩
     android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
</LinearLayout>
```

As you can see, simply adding audio recording capabilities is straightforward. It doesn't allow much in the way of control over the user interface or other aspects of the recording, but it does give a no-frills usable interface to the user without much work on our part.

Custom Audio Capture

Of course, using an intent to trigger the sound recorder isn't the only way we can capture audio. The Android SDK includes a `MediaRecorder` class, which we can leverage

to build our own audio recording functionality. Doing so enables a lot more flexibility, such as controlling the length of time audio is recorded for.

The `MediaRecorder` class is used for both audio and video capture. After constructing a `MediaRecorder` object, to capture audio, the `setAudioEncoder` and `setAudioSource` methods must be called. If these methods are not called, audio will not be recorded. (The same goes for video. If `setVideoEncoder` and `setVideoSource` methods are not called, video will not be recorded. We won't be dealing with video in this chapter; therefore we won't use either of these methods.)

Additionally, two other methods are generally called before having the `MediaRecorder` prepare to record. These are `setOutputFormat` and `setOutputFile`. `setOutputFormat` allows us to choose what file format should be used for the recording and `setOutputFile` allows us to specify the file that we will record to. It is important to note that the order of each of these calls matters quite a bit.

MediaRecorder Audio Sources

The first method that should be called after the `MediaRecorder` is instantiated is `setAudioSource`. `setAudioSource` takes in a constant that is defined in the `AudioSource` inner class. Generally we will want to use `MediaRecorder.AudioSource.MIC`, but it is interesting to note that `MediaRecorder.AudioSource` also contains constants for `VOICE_CALL`, `VOICE_DOWNLINK`, and `VOICE_UPLINK`. Unfortunately, it appears as though there aren't any handsets or versions of Android where recording audio from the call actually works. Also of note, as of Froyo, Android version 2.2, there are constants for `CAMCORDER` and `VOICE_RECOGNITION`. These may be used if the device has more than one microphone.

```
MediaRecorder recorder = new MediaRecorder();
recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
```

MediaRecorder Output Formats

The next method to be called in sequence is `setOutputFormat`. The values this takes in are specified as constants in the `MediaRecorder.OutputFormat` inner class.

- *`MediaRecorder.OutputFormat.MPEG_4`*: This specifies that the file written will be an MPEG-4 file. It may contain both audio and video tracks.
- *`MediaRecorder.OutputFormat.RAW_AMR`*: This represents a raw file without any type of container. This should be used only when capturing audio without video and when the audio encoder is AMR_NB.
- *`MediaRecorder.OutputFormat.THREE_GPP`*: This specifies that the file written will be a 3GPP file (extension `.3gp`). It may contain both audio and video tracks.

```
recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
```

MediaRecorder Audio Encoders

Following the setting of the output format, we can call `setAudioEncoder` to set the codec that should be used. The possible values are specified as constants in the `MediaRecorder.AudioEncoder` class and other than `DEFAULT`, only one other value exists: `MediaRecorder.AudioEncoder.AMR_NB`, which is the Adaptive Multi-Rate Narrow Band codec. This codec is tuned for speech and is therefore not a great choice for anything other than speech. By default it has a sample rate of 8 kHz and a bitrate between 4.75 and 12.2 kbps, both of which are very low for recoding anything but speech. Unfortunately, this is our only choice for use with the `MediaRecorder` at the moment.

```
recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
```

MediaRecorder Output and Recording

Last, we'll want to call `setOutputFile` with the location of the file we want to record to. The following snippet of code creates a file using `File.createTempFile` in the preferred file location for applications that need to store files on the SD card.

```
File path = new File(Environment.getExternalStorageDirectory().getAbsolutePath() +↵
 "/Android/data/com.apress.proandroidmedia.ch07.customrecorder/files/");
path.mkdirs();
audioFile = File.createTempFile("recording", ".3gp", path);
recorder.setOutputFile(audioFile.getAbsolutePath());
```

Now we can actually call `prepare`, which signals the end of the configuration stage and tells the `MediaRecorder` to get ready to start recording. We call the `start` method to actually start recording.

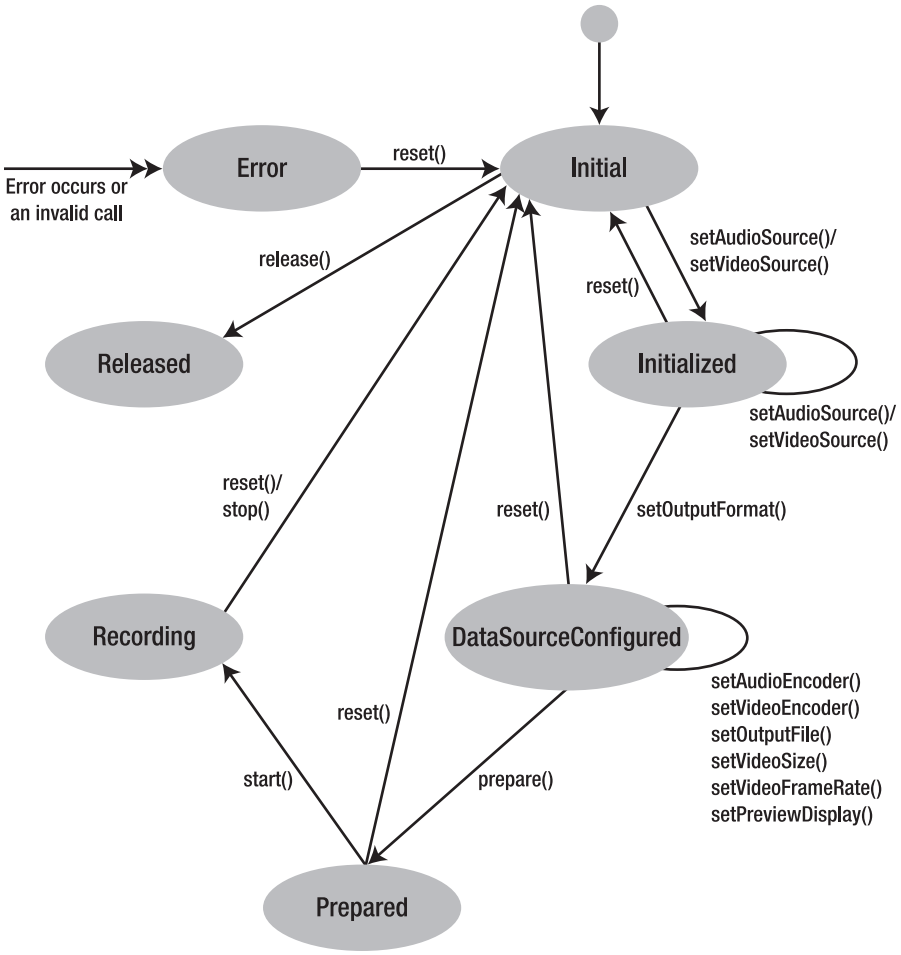
```
recorder.prepare();
recorder.start();
```

To stop recording, we call the `stop` method.

```
recorder.stop();
```

MediaRecorder State Machine

The `MediaRecorder`, similar to the `MediaPlayer`, operates as a state machine. Figure 7–2 shows a diagram from the Android API reference page for `MediaRecorder`, which describes the various states and the methods that may be called from each state.



MediaRecorder state diagram

Figure 7–2. *`MediaRecorder` state diagram from Android API Reference*

MediaRecorder Example

Here is the code for a full custom audio capture and playback example using the `MediaRecorder` class.

```
package com.apress.proandroidmedia.ch07.customrecorder;

import java.io.File;
import java.io.IOException;

import android.app.Activity;
import android.media.MediaPlayer;
import android.media.MediaRecorder;
import android.media.MediaPlayer.OnCompletionListener;
```

```
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
```

Our `CustomRecorder` activity implements `OnClickListener` so that it may be notified when Buttons are pressed, and `OnCompletionListener` so that it can respond when `MediaPlayer` has completed playing audio.

```
public class CustomRecorder extends Activity implements OnClickListener,↩
 OnCompletionListener {
```

We'll have a series of user interface components. The first, a `TextView` called `statusTextView`, will report the status of the application to the user: "Recording," "Ready to Play," and so on.

```
    TextView statusTextView;
```

A series of buttons will be used for controlling various aspects. The names of the Buttons describe their use.

```
    Button startRecording, stopRecording, playRecording, finishButton;
```

We'll have a `MediaRecorder` for recording the audio and a `MediaPlayer` for playing it back.

```
    MediaRecorder recorder;
    MediaPlayer player;
```

Finally, we have a `File` object called `audioFile`, which will reference the file that is recorded to.

```
    File audioFile;

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

When the activity starts up, we'll set the text of the `statusTextView` to be "Ready."

```
        statusTextView = (TextView) this.findViewById(R.id.StatusTextView);
        statusTextView.setText("Ready");

        stopRecording = (Button) this.findViewById(R.id.StopRecording);
        startRecording = (Button) this.findViewById(R.id.StartRecording);
        playRecording = (Button) this.findViewById(R.id.PlayRecording);
        finishButton = (Button) this.findViewById(R.id.FinishButton);
```

We'll set all of the Buttons' `onClickListeners` to be `this` so that our `onClick` method is called when any of them are pressed.

```
        startRecording.setOnClickListener(this);
        stopRecording.setOnClickListener(this);
        playRecording.setOnClickListener(this);
        finishButton.setOnClickListener(this);
```

Finally, in the onCreate method, we'll disable the stopRecording and playRecording Buttons since they won't work until we either start recording or finish recording respectively.

```
        stopRecording.setEnabled(false);
        playRecording.setEnabled(false);
    }
```

In the following onClick method, we handle all of the Button presses.

```
    public void onClick(View v) {
        if (v == finishButton) {
```

If the finishButton is pressed, we finish the activity.

```
            finish();
        } else if (v == stopRecording) {
```

If the stopRecording Button is pressed, we call stop and release on the MediaRecorder object.

```
            recorder.stop();
            recorder.release();
```

We then construct a MediaPlayer object and have it prepare to play back the audio file that we just recorded.

```
            player = new MediaPlayer();
            player.setOnCompletionListener(this);
```

The following two methods that we are using on the MediaPlayer, setDataSource and prepare, may throw a variety of exceptions. In the following code, we are simply throwing them. In your application development, you will probably want to catch and deal with them more elegantly, such as alerting the user when a file doesn't exist.

```
            try {
                player.setDataSource(audioFile.getAbsolutePath());
            } catch (IllegalArgumentException e) {
                throw new RuntimeException(
                    "Illegal Argument to MediaPlayer.setDataSource", e);
            } catch (IllegalStateException e) {
                throw new RuntimeException(
                    "Illegal State in MediaPlayer.setDataSource", e);
            } catch (IOException e) {
                throw new RuntimeException(
                    "IOException in MediaPalyer.setDataSource", e);
            }

            try {
                player.prepare();
            } catch (IllegalStateException e) {
                throw new RuntimeException(
                    "IllegalStateException in MediaPlayer.prepare", e);
            } catch (IOException e) {
                throw new RuntimeException("IOException in MediaPlayer.prepare", e);
            }
```

We set the statusTextView to indicate to the user that we are ready to play the audio file.

```
            statusTextView.setText("Ready to Play");
```

We then set the playRecording and startRecording Buttons to be enabled and disable the stopRecording Button, as we are not currently recording.

```
            playRecording.setEnabled(true);
            stopRecording.setEnabled(false);
            startRecording.setEnabled(true);
        } else if (v == startRecording) {
```

When the startRecording Button is pressed, we construct a new MediaRecorder and call setAudioSource, setOutputFormat, and setAudioEncoder.

```
            recorder = new MediaRecorder();

            recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
            recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
            recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
```

We then create a new File on the SD card and call setOutputFile on the MediaRecorder object.

```
            File path = new File(Environment.getExternalStorageDirectory()↩
.getAbsolutePath() + "/Android/data/com.apress.proandroidmedia.ch07↩
.customrecorder/files/");
            path.mkdirs();

            try {
                audioFile = File.createTempFile("recording", ".3gp", path);
            } catch (IOException e) {
                throw new RuntimeException("Couldn't create recording audio file",e);
            }

            recorder.setOutputFile(audioFile.getAbsolutePath());
```

We call prepare on the MediaRecorder and start to begin the recording.

```
            try {
                recorder.prepare();
            } catch (IllegalStateException e) {
                throw new RuntimeException(
                    "IllegalStateException on MediaRecorder.prepare", e);
            } catch (IOException e) {
                throw new RuntimeException("IOException on MediaRecorder.prepare",e);
            }

            recorder.start();
```

Last, we update the statusTextView and change which Buttons are enabled and disabled.

```
            statusTextView.setText("Recording");

            playRecording.setEnabled(false);
            stopRecording.setEnabled(true);
            startRecording.setEnabled(false);
        } else if (v == playRecording) {
```

The last Button that we need to respond to is playRecording. When the stopRecording Button is pressed, the MediaPlayer object, player, is constructed and configured. All that we need to do when the playRecording Button is pushed is to start the playback, set the status message, and change which Buttons are enabled.

```
            player.start();
            statusTextView.setText("Playing");
            playRecording.setEnabled(false);
            stopRecording.setEnabled(false);
            startRecording.setEnabled(false);
        }
    }
```

The onCompletion method is called when the MediaPlayer object has completed playback of a recording. We use it to change the status message and set which Buttons are enabled.

```
    public void onCompletion(MediaPlayer mp) {
        playRecording.setEnabled(true);
        stopRecording.setEnabled(false);
        startRecording.setEnabled(true);
        statusTextView.setText("Ready");
    }
}
```

Here is the layout XML file, main.xml, for the foregoing activity.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView android:layout_width="wrap_content" android:layout_height=↩
"wrap_content" android:id="@+id/StatusTextView" android:text="Status"↩
 android:textSize="35dip"></TextView>

    <Button android:text="Start Recording" android:id="@+id/StartRecording"↩
 android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
    <Button android:text="Stop Recording" android:id="@+id/StopRecording"↩
 android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
    <Button android:text="Play Recording" android:id="@+id/PlayRecording"↩
 android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
 android:id="@+id/FinishButton" android:text="Finish"></Button>
</LinearLayout>
```

We'll also need to add the following permissions to the AndroidManifest.xml file.

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

As we have seen, developing a custom audio capture application using MediaRecorder is not too cumbersome. Now let's look at how we can use the MediaRecorder's other methods to add other features.

Other MediaRecorder Methods

MediaRecorder has a variety of other methods available that we can use in relation to audio capture.

- *getMaxAmplitude*: Allows us to request the maximum amplitude of audio that has been recorded by the MediaPlayer. The value is reset each time the method is called, so each call will return the maximum amplitude from the last time it is called. An audio level meter may be implemented by calling this method periodically.
- *setMaxDuration*: Allows us to specify a maximum recording duration in milliseconds. This method must be called after the setOutputFormat method but before the prepare method.
- *setMaxFileSize*: Allows us to specify a maximum file size for the recording in bytes. As with setMaxDuration, this method must be called after the setOutputFormat method but before the prepare method.

Here is an update to the custom recorder application we went through previously that includes a display of the current amplitude.

```
package com.apress.proandroidmedia.ch07.customrecorder;

import java.io.File;
import java.io.IOException;

import android.app.Activity;
import android.media.MediaPlayer;
import android.media.MediaRecorder;
import android.media.MediaPlayer.OnCompletionListener;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;

public class CustomRecorder extends Activity implements OnClickListener,
        OnCompletionListener {
```

In this version, we have added a TextView called amplitudeTextView. This will display the numeric amplitude of the audio input.

```
    TextView statusTextView, amplitudeTextView;
    Button startRecording, stopRecording, playRecording, finishButton;
    MediaRecorder recorder;
    MediaPlayer player;
    File audioFile;
```

We'll need an instance of a new class called RecordAmplitude. This class is an inner class that is defined toward the end of this source code listing. It uses a Boolean called isRecording that will be set to true when we start the MediaRecorder.

```
RecordAmplitude recordAmplitude;
boolean isRecording = false;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    statusTextView = (TextView) this.findViewById(R.id.StatusTextView);
    statusTextView.setText("Ready");
```

We'll use a `TextView` to display the current amplitude of the audio as it is captured.

```
    amplitudeTextView = (TextView) this
            .findViewById(R.id.AmplitudeTextView);
    amplitudeTextView.setText("0");

    stopRecording = (Button) this.findViewById(R.id.StopRecording);
    startRecording = (Button) this.findViewById(R.id.StartRecording);
    playRecording = (Button) this.findViewById(R.id.PlayRecording);
    finishButton = (Button) this.findViewById(R.id.FinishButton);

    startRecording.setOnClickListener(this);
    stopRecording.setOnClickListener(this);
    playRecording.setOnClickListener(this);
    finishButton.setOnClickListener(this);

    stopRecording.setEnabled(false);
    playRecording.setEnabled(false);
}

public void onClick(View v) {
    if (v == finishButton) {
        finish();
    } else if (v == stopRecording) {
```

When we finish the recording, we set the `isRecording` Boolean to `false` and call cancel on our `RecordAmplitude` class. Since `RecordAmplitude` extends `AsyncTask`, calling `cancel` with `true` as the parameter will interrupt its thread if necessary.

```
        isRecording = false;
        recordAmplitude.cancel(true);

        recorder.stop();
        recorder.release();

        player = new MediaPlayer();
        player.setOnCompletionListener(this);
        try {
            player.setDataSource(audioFile.getAbsolutePath());
        } catch (IllegalArgumentException e) {
            throw new RuntimeException(
                    "Illegal Argument to MediaPlayer.setDataSource", e);
        } catch (IllegalStateException e) {
            throw new RuntimeException(
                    "Illegal State in MediaPlayer.setDataSource", e);
        } catch (IOException e) {
            throw new RuntimeException(
```

```
                    "IOException in MediaPalyer.setDataSource", e);
            }

            try {
                player.prepare();
            } catch (IllegalStateException e) {
                throw new RuntimeException(
                        "IllegalStateException in MediaPlayer.prepare", e);
            } catch (IOException e) {
                throw new RuntimeException(
                        "IOException in MediaPlayer.prepare", e);
            }

            statusTextView.setText("Ready to Play");

            playRecording.setEnabled(true);
            stopRecording.setEnabled(false);
            startRecording.setEnabled(true);

        } else if (v == startRecording) {
            recorder = new MediaRecorder();

            recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
            recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
            recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);

            File path = new File(Environment.getExternalStorageDirectory()↩
                    .getAbsolutePath() + "/Android/data/com.apress.proandroidmedia.ch07↩
.customrecorder/files/");
            path.mkdirs();
            try {
                audioFile = File.createTempFile("recording", ".3gp", path);
            } catch (IOException e) {
                throw new RuntimeException(
                        "Couldn't create recording audio file", e);
            }
            recorder.setOutputFile(audioFile.getAbsolutePath());

            try {
                recorder.prepare();
            } catch (IllegalStateException e) {
                throw new RuntimeException(
                        "IllegalStateException on MediaRecorder.prepare", e);
            } catch (IOException e) {
                throw new RuntimeException(
                        "IOException on MediaRecorder.prepare", e);
            }
            recorder.start();
```

After we start the recording, we set the `isRecording` Boolean to `true` and create a new instance of `RecordAmplitude`. Since `RecordAmplitude` extends `AsyncTask`, we'll call the `execute` method to start the `RecordAmplitude`'s task running.

```
            isRecording = true;
            recordAmplitude = new RecordAmplitude();
            recordAmplitude.execute();

            statusTextView.setText("Recording");

            playRecording.setEnabled(false);
            stopRecording.setEnabled(true);
            startRecording.setEnabled(false);
        } else if (v == playRecording) {
            player.start();
            statusTextView.setText("Playing");
            playRecording.setEnabled(false);
            stopRecording.setEnabled(false);
            startRecording.setEnabled(false);
        }
    }

    public void onCompletion(MediaPlayer mp) {
        playRecording.setEnabled(true);
        stopRecording.setEnabled(false);
        startRecording.setEnabled(true);
        statusTextView.setText("Ready");
    }
```

Here is the definition of `RecordAmplitude`. It extends `AsyncTask`, which is a nice utility class in Android that provides a thread to run long-running tasks without tying up the user interface or making an application unresponsive.

```
    private class RecordAmplitude extends AsyncTask<Void, Integer, Void> {
```

The `doInBackground` method runs on a separate thread and is run when the `execute` method is called on the object. This method loops as long as `isRecording` is `true` and calls `Thread.sleep(500)`, which causes it to not do anything for half a second. Once that is complete, it calls `publishProgress` and passes in the result of `getMaxAmplitude` on the `MediaRecorder` object.

```
        @Override
        protected Void doInBackground(Void... params) {
            while (isRecording) {

                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                publishProgress(recorder.getMaxAmplitude());
            }
            return null;
        }
```

The preceding call to `publishProgress` calls the `onProgressUpdate` method defined here, which runs on the main thread so it can interact with the user interface. In this case, it is updating the `amplitudeTextView` with the value that is passed in from the `publishProgress` method call.

```
        protected void onProgressUpdate(Integer... progress) {
            amplitudeTextView.setText(progress[0].toString());
        }
    }
}
```

Of course, we'll need to update the layout XML to include the TextView for displaying the amplitude.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
        <TextView android:layout_width="wrap_content" android:layout_height=↩
"wrap_content" android:id="@+id/StatusTextView" android:text="Status"↩
 android:textSize="35dip"></TextView>
        <TextView android:layout_width="wrap_content" android:layout_height=↩
"wrap_content" android:id="@+id/AmplitudeTextView" android:textSize="35dip"↩
 android:text="0"></TextView>
        <Button android:text="Start Recording" android:id="@+id/StartRecording"↩
 android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
        <Button android:text="Stop Recording" android:id="@+id/StopRecording"↩
 android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
        <Button android:text="Play Recording" android:id="@+id/PlayRecording"↩
 android:layout_width="wrap_content" android:layout_height="wrap_content"></Button>
        <Button android:layout_width="wrap_content" android:layout_height=↩
"wrap_content" android:id="@+id/FinishButton" android:text="Finish"></Button>
</LinearLayout>
```

As we can see, using an `AsyncTask` to do something periodically is a nice way to provide automatically updating information to the user while something else is in progress. This provides a nicer user experience for our `MediaRecorder` example. Using the `getMaxAmplitude` method provides the user with some feedback about the recording that is currently happening.

In Android 2.2, Froyo, the following methods were made available:

- *`setAudioChannels`*: Allows us to specify the number of audio channels that will be recorded. Typically this will be either one channel (mono) or two channels (stereo). This method must be called prior to the `prepare` method.
- *`setAudioEncodingBitRate`*: Allows us to specify the number of bits per second that will be used by the encoder when compressing the audio. This method must be called prior to the `prepare` method.
- *`setAudioSamplingRate`*: Allows us to specify the sampling rate of the audio as it is captured and encoded. The applicable rates are determined by the hardware and codec being used. This method must be called prior to the `prepare` method.

Inserting Audio into the MediaStore

Audio recordings may be put into the MediaStore content provider so they are available to other applications. The process is very similar to the process we used earlier to add images to the `MediaStore`. In this case though, we'll add them after they are created.

We create a `ContentValues` object to hold the data that we'll insert into the MediaStore. A `ContentValues` object is made up of a series of key/value pairs. The keys that may be used are defined as constants in the `MediaStore.Audio.Media` class (and those classes it inherits from).

The `MediaStore.Audio.Media.DATA` constant is the key for the path to the recorded file. It is the only required pair in order to insert the file into the MediaStore.

To do the actual insert into the MediaStore, we use the `insert` method on a `ContentResolver` object with the Uri to the table for audio files on the SD card and the `ContentValues` object containing the data. The Uri is defined as a constant in `MediaStore.Audio.Media` named `EXTERNAL_CONTENT_URI`.

Here is a snippet that may be plugged into the `CustomRecorder` example just after the `release` method is called on the `MediaRecorder` (`recorder.release()`). It will cause the recording to be inserted into the MediaStore and made available to other applications that use the MediaStore for finding audio to play back.

```
ContentValues contentValues = new ContentValues();
contentValues.put(MediaStore.MediaColumns.TITLE, "This Isn't Music");
contentValues.put(MediaStore.MediaColumns.DATE_ADDED, System.currentTimeMillis());
contentValues.put(MediaStore.Audio.Media.DATA, audioFile.getAbsolutePath());
Uri newUri =
  getContentResolver().insert(
    MediaStore.Audio.Media.EXTERNAL_CONTENT_URI, contentValues);
```

Of course, in order to use the foregoing snippet, we'll need to add these imports:

```
import android.content.ContentValues;
import android.net.Uri;
import android.provider.MediaStore;
```

Raw Audio Recording with AudioRecord

Aside from using an intent to launch the sound recorder and using the `MediaRecorder`, Android offers a third method to capture audio, using a class called `AudioRecord`. `AudioRecord` is the most flexible of the three methods in that it allows us access to the raw audio stream but has the least number of built-in capabilities, such as not automatically compressing the audio.

The basics for using `AudioRecord` are straightforward. We simply need to construct an object of type `AudioRecord`, passing in various configuration parameters.

The first value we'll need to specify is the audio source. The values for use here are the same as we used for the `MediaRecorder` and are defined in `MediaRecorder.AudioSource`. Essentially this means that we have `MediaRecorder.AudioSource.MIC` available to us.

```
int audioSource = MediaRecorder.AudioSource.MIC;
```

The next value that we'll need to specify is the sample rate of the recording. This should be specified in Hz. As we know, the MediaRecorder samples audio at 8 kHz or 8,000 Hz. CD quality audio is typically 44.1 kHz or 44,100 Hz. Hz or hertz is the number of samples per second. Different Android handset hardware will be able to sample at different sample rates. For our example application, we'll sample at 11,025 Hz, which is another commonly used sample rate.

```
int sampleRateInHz = 11025;
```

Next, we need to specify the number of channels of audio to capture. The constants for this parameter are specified in the `AudioFormat` class and are self-explanatory.

```
AudioFormat.CHANNEL_CONFIGURATION_MONO
AudioFormat.CHANNEL_CONFIGURATION_STEREO
AudioFormat.CHANNEL_CONFIGURATION_INVALID
AudioFormat.CHANNEL_CONFIGURATION_DEFAULT
```

We'll use a mono configuration for now.

```
int channelConfig = AudioFormat.CHANNEL_CONFIGURATION_MONO;
```

Following that, we need to specify the audio format. The possibilities here are also specified in the `AudioFormat` class.

```
AudioFormat.ENCODING_DEFAULT
AudioFormat.ENCODING_INVALID
AudioFormat.ENCODING_PCM_16BIT
AudioFormat.ENCODING_PCM_8BIT
```

Among these four, our choices boil down to PCM 16-bit and PCM 8-bit. PCM stands for Pulse Code Modulation, which is essentially the raw audio samples. We can therefore set the resolution of each sample to be 16 bits or 8 bits. Sixteen bits will take up more space and processing power, while the representation of the audio will be closer to reality.

For our example, we'll use the 16-bit version.

```
int audioFormat = AudioFormat.ENCODING_PCM_16BIT;
```

Last, we'll need to specify the buffer size. We can actually ask the `AudioRecord` class what the minimum buffer size should be with a static method call, `getMinBufferSize`, passing in the sample rate, channel configuration, and audio format.

```
int bufferSizeInBytes = AudioRecord.getMinBufferSize(sampleRateInHz, channelConfig,↩
 audioFormat);
```

Now we can construct the actual `AudioRecord` object.

```
AudioRecord audioRecord = new AudioRecord(audioSource, sampleRateInHz, channelConfig,↩
 audioFormat, bufferSizeInBytes);
```

The AudioRecord class doesn't actually save the captured audio anywhere. We need to do that manually as the audio comes in. The first thing we'll probably want to do is record it to a file.

To do that, we'll need to create a file.

```
File recordingFile;
File path = new File(Environment.getExternalStorageDirectory()↩
                .getAbsolutePath() + "/Android/data/com.apress.proandroidmedia.ch07↩
.altaudiorecorder /files/");
path.mkdirs();
try {
    recordingFile = File.createTempFile("recording", ".pcm", path);
} catch (IOException e1) {
    throw new RuntimeException("Couldn't create file on SD card", e);
}
```

Next we create an OutputStream to that file, specifically one wrapped in a BufferedOutputStream and a DataOutputStream for performance and convenience reasons.

```
DataOutputStream dos = new DataOutputStream(new BufferedOutputStream(new↩
 FileOutputStream(recordingFile)));
```

Now we can start the capture and write the audio samples to the file. We'll use an array of shorts to hold the audio we read from the AudioRecord object. We'll make the array smaller than the buffer that the AudioRecord object has so that buffer won't fill up before we read it out.

To make sure this array is smaller than the buffer size, we divide by 4. The size of the buffer is in bytes and each short takes up 2 bytes, so dividing by 2 won't be enough. Dividing by 4 will make it so that this array is half the size of the AudioRecord object's internal buffer.

```
short[] buffer = new short[bufferSize/4];
```

We simply call the startRecording method on the AudioRecord object to kick things off.

```
audioRecord.startRecording();
```

After recording has started, we can construct a loop to continuously read from the AudioRecord object into our array of shorts and write that to the DataOutputStream for the file.

```
while (true) {
    int bufferReadResult = audioRecord.read(buffer, 0, bufferSize/4);
    for (int i = 0; i < bufferReadResult; i++) {
        dos.writeShort(buffer[i]);
    }
}
audioRecord.stop();
dos.close();
```

When we are done, we call stop on the AudioRecord object and close on the DataOutputStream.

Of course, in the real world, we wouldn't put this in a `while (true)` loop as it will never complete. We also probably want to run this in some kind of thread so that it doesn't tie up the user interface and anything else we might want the application to do while recording.

Before going through a full example, let's look at how we can play back audio as it is captured using the `AudioRecord` class.

Raw Audio Playback with AudioTrack

`AudioTrack` is a class in Android that allows us to play raw audio samples. This allows for the playback of audio captured with `AudioRecord` that otherwise wouldn't be playable using a `MediaPlayer` object.

To construct an `AudioTrack` object, we need to pass in a series of configuration variables describing the audio to be played.

- The first argument is the stream type. The possible values are defined as constants in the `AudioManager` class. We'll be using `AudioManager.STREAM_MUSIC`, which is the audio stream used for normal music playback.
- The second argument is the sample rate in hertz of the audio data that will be played back. In our example, we'll be capturing audio at 11,025 Hz, and therefore, to play it back, we need to specify the same value.
- The third argument is the channel configuration. The possible values, the same as those used when constructing an `AudioRecord` object, are defined as constants in the `AudioFormat` class. Their names are self-explanatory.
  - `AudioFormat.CHANNEL_CONFIGURATION_MONO`
  - `AudioFormat.CHANNEL_CONFIGURATION_STEREO`
  - `AudioFormat.CHANNEL_CONFIGURATION_INVALID`
  - `AudioFormat.CHANNEL_CONFIGURATION_DEFAULT`
- The fourth argument is the format of the audio. The possible values are the same as those used when constructing an `AudioRecord` object, and they are defined in `AudioFormat` as constants. The value used should match the value of the audio that will be passed in.
  - `AudioFormat.ENCODING_DEFAULT`
  - `AudioFormat.ENCODING_INVALID`
  - `AudioFormat.ENCODING_PCM_16BIT`
  - `AudioFormat.ENCODING_PCM_8BIT`

- The fifth argument is the size of the buffer that will be used in the object to store the audio. To determine the smallest buffer size to use, we can call `getMinBufferSize`, passing in the sample rate, the channel configuration, and audio format.

```
int frequency = 11025;
int channelConfiguration = AudioFormat.CHANNEL_CONFIGURATION_MONO;
int audioEncoding = AudioFormat.ENCODING_PCM_16BIT;

int bufferSize = AudioTrack.getMinBufferSize(frequency, channelConfiguration,↵
 audioEncoding);
```

- The last argument is the mode. The possible values are defined as constants in the `AudioTrack` class.
  - *`AudioTrack.MODE_STATIC`*: The audio data will all be transferred to the `AudioTrack` object before playback occurs.
  - *`AudioTrack.MODE_STREAM`*: The audio data will continue to be transferred to the `AudioTrack` object while playback is in progress.

Here is our `AudioTrack` configuration:

```
AudioTrack audioTrack = new AudioTrack(AudioManager.STREAM_MUSIC, frequency,
                channelConfiguration, audioEncoding, bufferSize,
                AudioTrack.MODE_STREAM);
```

Once the `AudioTrack` is constructed, we need to open an audio source, read the audio data into a buffer, and pass it to the `AudioTrack` object.

We'll construct a `DataInputStream` from a file containing raw PCM data in the right format (11,025 Hz, 16 bit, mono).

```
DataInputStream dis = new DataInputStream(
            new BufferedInputStream(new FileInputStream(recordingFile)));
```

We can then call `play` on the `AudioTrack` and start writing audio in from the `DataInputStream`.

```
audioTrack.play();

while (isPlaying && dis.available() > 0) {
    int i = 0;
    while (dis.available() > 0 && i < audiodata.length) {
        audiodata[i] = dis.readShort();
        i++;
    }
    audioTrack.write(audiodata, 0, audiodata.length);
}

dis.close();
```

That covers the basics of using `AudioTrack` to play back audio from a file as it is recorded from an `AudioRecorder`.

Raw Audio Capture and Playback Example

Here is a full example that records using `AudioRecord` and plays back using `AudioTrack`. Each of these operations lives in their own thread through the use of `AsyncTask`, so that they don't make the application become unresponsive by running in the main thread.

```
package com.apress.proandroidmedia.ch07.altaudiorecorder;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

import android.app.Activity;
import android.media.AudioFormat;
import android.media.AudioManager;
import android.media.AudioRecord;
import android.media.AudioTrack;
import android.media.MediaRecorder;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;

public class AltAudioRecorder extends Activity implements OnClickListener {
```

We have two inner classes defined—one for the recording and one for the playback. Each one extends `AsyncTask`.

```
    RecordAudio recordTask;
    PlayAudio playTask;

    Button startRecordingButton, stopRecordingButton, startPlaybackButton,
            stopPlaybackButton;
    TextView statusText;

    File recordingFile;
```

We'll use Booleans to keep track of whether we should be recording and playing. These will be used in the loops in recording and playback tasks.

```
    boolean isRecording = false;
    boolean isPlaying = false;
```

Here are the variables that we'll use to define the configuration of both the `AudioRecord` and `AudioTrack` objects.

```
    // These should really be constants themselves
    int frequency = 11025;
```

```
    int channelConfiguration = AudioFormat.CHANNEL_CONFIGURATION_MONO;
    int audioEncoding = AudioFormat.ENCODING_PCM_16BIT;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        statusText = (TextView) this.findViewById(R.id.StatusTextView);

        startRecordingButton = (Button) this .findViewById(R.id.StartRecordingButton);
        stopRecordingButton = (Button) this .findViewById(R.id.StopRecordingButton);
        startPlaybackButton = (Button) this .findViewById(R.id.StartPlaybackButton);
        stopPlaybackButton = (Button) this.findViewById(R.id.StopPlaybackButton);

        startRecordingButton.setOnClickListener(this);
        stopRecordingButton.setOnClickListener(this);
        startPlaybackButton.setOnClickListener(this);
        stopPlaybackButton.setOnClickListener(this);

        stopRecordingButton.setEnabled(false);
        startPlaybackButton.setEnabled(false);
        stopPlaybackButton.setEnabled(false);
```

The last thing we'll do in the constructor is create the file that we'll record to and play back from. In this case, we are creating the file in the preferred location for files associated with an application on the SD card.

```
        File path = new File(Environment.getExternalStorageDirectory()↵
                .getAbsolutePath() + "/Android/data/com.apress.proandroidmedia.ch07↵
.altaudiorecorder/files/");
        path.mkdirs();
        try {
            recordingFile = File.createTempFile("recording", ".pcm", path);
        } catch (IOException e) {
            throw new RuntimeException("Couldn't create file on SD card", e);
        }
    }
```

The `onClick` method handles the Button presses generated by the user. Each one corresponds to a specific method.

```
    public void onClick(View v) {
        if (v == startRecordingButton) {
            record();
        } else if (v == stopRecordingButton) {
            stopRecording();
        } else if (v == startPlaybackButton) {
            play();
        } else if (v == stopPlaybackButton) {
            stopPlaying();
        }
    }
```

To start playback, we construct a new `PlayAudio` object and call its `execute` method, which is inherited from `AsyncTask`.

```
    public void play() {
```

```
        startPlaybackButton.setEnabled(true);

        playTask = new PlayAudio();
        playTask.execute();

        stopPlaybackButton.setEnabled(true);
    }
```

To stop playback, we set the `isPlaying` Boolean to `false` and that's it. This will cause the `PlayAudio` object's loop to finish.

```
    public void stopPlaying() {
        isPlaying = false;
        stopPlaybackButton.setEnabled(false);
        startPlaybackButton.setEnabled(true);
    }
```

To start recording, we construct a `RecordAudio` object and call its `execute` method.

```
    public void record() {
        startRecordingButton.setEnabled(false);
        stopRecordingButton.setEnabled(true);

        // For Fun
        startPlaybackButton.setEnabled(true);

        recordTask = new RecordAudio();
        recordTask.execute();
    }
```

To stop recording, we simply set the `isRecording` Boolean to `false`. This allows the `RecordAudio` object to stop looping and perform any cleanup.

```
    public void stopRecording() {
        isRecording = false;
    }
```

Here is our `PlayAudio` inner class. This class extends `AsyncTask` and uses an `AudioTrack` object to play back the audio.

```
    private class PlayAudio extends AsyncTask<Void, Integer, Void> {
        @Override
        protected Void doInBackground(Void... params) {
            isPlaying = true;

            int bufferSize = AudioTrack.getMinBufferSize(frequency,
                    channelConfiguration, audioEncoding);
            short[] audiodata = new short[bufferSize/4];

            try {
                DataInputStream dis = new DataInputStream(
                        new BufferedInputStream(new FileInputStream(
                                recordingFile)));

                AudioTrack audioTrack = new AudioTrack(
                        AudioManager.STREAM_MUSIC, frequency,
                        channelConfiguration, audioEncoding, bufferSize,
                        AudioTrack.MODE_STREAM);
```

```
                audioTrack.play();

                while (isPlaying && dis.available() > 0) {
                    int i = 0;
                    while (dis.available() > 0 && i < audiodata.length) {
                        audiodata[i] = dis.readShort();
                        i++;
                    }
                    audioTrack.write(audiodata, 0, audiodata.length);
                }

                dis.close();

                startPlaybackButton.setEnabled(false);
                stopPlaybackButton.setEnabled(true);

            } catch (Throwable t) {
                Log.e("AudioTrack", "Playback Failed");
            }

            return null;
        }
    }
```

Last is our `RecordAudio` class, which extends `AsyncTask`. This class runs an `AudioRecord` object in the background and calls `publishProgress` to update the UI with an indication of recording progress.

```
    private class RecordAudio extends AsyncTask<Void, Integer, Void> {
        @Override
        protected Void doInBackground(Void... params) {
            isRecording = true;

            try {
                DataOutputStream dos = new DataOutputStream(
                        new BufferedOutputStream(new FileOutputStream(
                                recordingFile)));

                int bufferSize = AudioRecord.getMinBufferSize(frequency,
                        channelConfiguration, audioEncoding);

                AudioRecord audioRecord = new AudioRecord(
                        MediaRecorder.AudioSource.MIC, frequency,
                        channelConfiguration, audioEncoding, bufferSize);

                short[] buffer = new short[bufferSize];
                audioRecord.startRecording();

                int r = 0;
                while (isRecording) {
                    int bufferReadResult = audioRecord.read(buffer, 0,
                            bufferSize);
                    for (int i = 0; i < bufferReadResult; i++) {
                        dos.writeShort(buffer[i]);
                    }

                    publishProgress(new Integer(r));
```

```
                    r++;
                }

                audioRecord.stop();
                dos.close();
            } catch (Throwable t) {
                Log.e("AudioRecord", "Recording Failed");
            }

            return null;
        }
```

When `publishProgress` is called, `onProgressUpdate` is the method called.

```
        protected void onProgressUpdate(Integer... progress) {
            statusText.setText(progress[0].toString());
        }
```

When the `doInBackground` method completes, the following `onPostExecute` method is called.

```
        protected void onPostExecute(Void result) {
            startRecordingButton.setEnabled(true);
            stopRecordingButton.setEnabled(false);
            startPlaybackButton.setEnabled(true);
        }
    }
}
```

Here is the layout XML for the foregoing example:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:text="Status" android:id=↩
    "@+id/StatusTextView"/>

    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
     android:text="Start Recording" android:id="@+id/StartRecordingButton"></Button>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
     android:text="Stop Recording" android:id="@+id/StopRecordingButton"></Button>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
     android:text="Start Playback" android:id="@+id/StartPlaybackButton"></Button>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↩
     android:text="Stop Playback" android:id="@+id/StopPlaybakButton" ></Button>
</LinearLayout>
```

And, we'll need to add these permissions to `AndroidManifest.xml`.

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

As we have seen, using the `AudioRecord` and `AudioTrack` classes to create a capture and playback application is much more cumbersome than working with the `MediaRecorder` and `MediaPlayer` classes. But as we'll see in the next chapter, it is worth the effort when we need to do any type of audio processing or want to synthesize audio.

Summary

In this chapter, we looked at three different methods for recording audio on Android. Each of them comes with their own plusses and minuses. Using the built-in sound recorder is great for no-fuss audio recordings, where little or no programmatic control is needed. Using the `MediaRecorder` allows us to take it a step further, allowing control over the length of time media is recorded and other aspects but leaving the interface up to us. Last we investigated the ability to record raw samples with `AudioRecord`. Using this we have the most control and flexibility but have to do the most work in order to capture and work with the audio.

In the next chapter, we'll look more at audio possibilities, investigating audio processing and synthesis.

Chapter 8

Audio Synthesis and Analysis

At the end of the last chapter, we looked at a way to capture raw PCM audio and play it back using the `AudioRecord` and `AudioTrack` classes. In this chapter, we'll continue using those classes to both algorithmically synthesize audio and analyze recorded audio.

Digital Audio Synthesis

Digital audio synthesis is a very broad topic with a great deal of theory, mathematics, engineering, and history behind it. Unfortunately, most of the topic overall is out of the scope of what can be covered in this book. What we will do is look at some basic examples on how we can harness a few built-in classes on Android to create audio from scratch.

As you probably know, sound is formed by a repetitive change in pressure in air (or other substance) in the form of a wave. Certain frequencies of these oscillations, otherwise known as sound waves, are audible, meaning our ears are sensitive to that number of repetitions in a period of time. This range is somewhere between 12 Hz (12 cycles per second), which is a very low sound such as a rumble, and 20 kHz (20,000 cycles per second), which is a very high-pitched sound.

To create audio, we need to cause the air to vibrate at the frequency desired for the sound we want. In the digital realm, this is generally done with a speaker that is driven by an analog electric signal. Digital audio systems contain a chip or board that performs a digital-to-analog conversion (DAC). A DAC will take in data in the form of a series of numbers that represent audio samples and convert that into an electrical voltage, which is translated into sound by the speaker.

In order to synthesize audio, we simply need to synthesize the audio samples and feed them to the appropriate mechanism. In the case of Android, that mechanism is the `AudioTrack` class.

As we learned in the last chapter, the AudioTrack class allows us to play raw audio samples (such as those captured by the AudioRecord class).

Playing a Synthesized Sound

Here is a quick example showing how to construct an AudioTrack class and pass in data to play. For a full discussion of the parameters used to construct the AudioTrack object, please see the "Raw Audio Playback with AudioTrack" section of Chapter 7.

This example uses an inner class that extends AsyncTask, AudioSynthesisTask. AsyncTask defines a method called doInBackground, which runs any code that is placed inside it in a thread that is separate from the main thread of the activity. This allows the activity and its UI to be responsive, as the loop that feeds the write method of our AudioTrack object would otherwise tie it up.

```
package com.apress.proandroidmedia.ch08.audiosynthesis;

import android.app.Activity;
import android.media.AudioFormat;
import android.media.AudioManager;
import android.media.AudioTrack;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class AudioSynthesis extends Activity implements OnClickListener {

    Button startSound;
    Button endSound;

    AudioSynthesisTask audioSynth;

    boolean keepGoing = false;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        startSound = (Button) this.findViewById(R.id.StartSound);
        startSound.setOnClickListener(this);

        endSound = (Button) this.findViewById(R.id.EndSound);
        endSound.setOnClickListener(this);

        endSound.setEnabled(false);
    }

    @Override
    public void onPause() {
        super.onPause();
        keepGoing = false;
```

```
        endSound.setEnabled(false);
        startSound.setEnabled(true);
    }

    public void onClick(View v) {
        if (v == startSound) {
            keepGoing = true;

            audioSynth = new AudioSynthesisTask();
            audioSynth.execute();

            endSound.setEnabled(true);
            startSound.setEnabled(false);
        } else if (v == endSound) {
            keepGoing = false;

            endSound.setEnabled(false);
            startSound.setEnabled(true);
        }
    }

    private class AudioSynthesisTask extends AsyncTask<Void, Void, Void>
    {
        @Override
        protected Void doInBackground(Void... params) {
            final int SAMPLE_RATE = 11025;

            int minSize = AudioTrack.getMinBufferSize(SAMPLE_RATE,
                    AudioFormat.CHANNEL_CONFIGURATION_MONO,
                    AudioFormat.ENCODING_PCM_16BIT);

            AudioTrack audioTrack = new AudioTrack(
                    AudioManager.STREAM_MUSIC, SAMPLE_RATE,
                    AudioFormat.CHANNEL_CONFIGURATION_MONO,
                    AudioFormat.ENCODING_PCM_16BIT,
                    minSize,
                    AudioTrack.MODE_STREAM);

            audioTrack.play();

            short[] buffer = {
                    8130,15752,22389,27625,31134,32695,32210,29711,25354,19410,12253,
                    4329,-3865,-11818,-19032,-25055,-29511,-32121,-32722,-31276,-27874,
                    -22728,-16160,-8582,-466
            };

            while (keepGoing) {
                    audioTrack.write(buffer, 0, buffer.length);
            }

            return null;
        }
    }
}
```

Here is the layout XML in use by the preceding activity.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↵
     android:id="@+id/StartSound" android:text="Start Sound"></Button>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↵
     android:id="@+id/EndSound" android:text="End Sound"></Button>
</LinearLayout>
```

The key to the foregoing code is the array of shorts. These are the audio samples that are continuously being passed into the `AudioTrack` object through the `write` method. In this case, the samples oscillate from 8,130 to 32,695, down to -32,121 and back up to -466. If we plotted these values on a graph, these samples taken together will construct a waveform. Since sound is created with oscillating pressure, and each of the samples represents a pressure value, having these samples represent a waveform is required to create sound. Varying this waveform allows us to create different kinds of audio. The following set of samples describes a short waveform, only ten samples, and therefore represents a high-frequency sound, one that has many oscillations per second. Low-frequency sounds would have a waveform that spans many more samples at a fixed sample rate.

```
short[] buffer = {
    8130,15752,32695,12253,4329,
    -3865,-19032,-32722,-16160,-466
};
```

Generating Samples

Using a little bit of math, we can algorithmically create these samples. The classic sine wave can be reproduced. This example produces a sine wave at 440 Hz.

```
package com.apress.proandroidmedia.ch08.audiosynthesis;

import android.app.Activity;
import android.media.AudioFormat;
import android.media.AudioManager;
import android.media.AudioTrack;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class AudioSynthesis extends Activity implements OnClickListener {

    Button startSound;
    Button endSound;
```

```
AudioSynthesisTask audioSynth;

boolean keepGoing = false;

float synth_frequency = 440; // 440 Hz, Middle A

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    startSound = (Button) this.findViewById(R.id.StartSound);
    startSound.setOnClickListener(this);

    endSound = (Button) this.findViewById(R.id.EndSound);
    endSound.setOnClickListener(this);

    endSound.setEnabled(false);
}

@Override
public void onPause() {
    super.onPause();
    keepGoing = false;

    endSound.setEnabled(false);
    startSound.setEnabled(true);
}

public void onClick(View v) {
    if (v == startSound) {
        keepGoing = true;

        audioSynth = new AudioSynthesisTask();
        audioSynth.execute();

        endSound.setEnabled(true);
        startSound.setEnabled(false);
    } else if (v == endSound) {
        keepGoing = false;

        endSound.setEnabled(false);
        startSound.setEnabled(true);
    }
}

private class AudioSynthesisTask extends AsyncTask<Void, Void, Void>
{
    @Override
    protected Void doInBackground(Void... params) {
        final int SAMPLE_RATE= 11025;

        int minSize = AudioTrack.getMinBufferSize(SAMPLE_RATE,
                AudioFormat.CHANNEL_CONFIGURATION_MONO,
                AudioFormat.ENCODING_PCM_16BIT);

        AudioTrack audioTrack = new AudioTrack(AudioManager.STREAM_MUSIC,
```

```
                    SAMPLE_RATE,
                    AudioFormat.CHANNEL_CONFIGURATION_MONO,
                    AudioFormat.ENCODING_PCM_16BIT,
                    minSize,
                    AudioTrack.MODE_STREAM);

            audioTrack.play();

            short[] buffer = new short[minSize];


            float angular_frequency =
               (float)(2*Math.PI) * synth_frequency / SAMPLE_RATE;
            float angle = 0;

            while (keepGoing) {
                 for (int i = 0; i < buffer.length; i++)
                 {

                     buffer[i] = (short)(Short.MAX_VALUE * ((float) Math.sin(angle)));
                     angle += angular_frequency;
                 }
                 audioTrack.write(buffer, 0, buffer.length);
            }

            return null;
        }
    }
}
```

Here is the layout XML file for the foregoing activity:

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

    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↵
     android:id="@+id/StartSound" android:text="Start Sound"></Button>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"↵
     android:id="@+id/EndSound" android:text="End Sound"></Button>
</LinearLayout>
```

Changing the `synth_frequency` would allow us to reproduce any other frequency we would like. Of course, changing the function used to generate the values would change the sound as well. You may want to try clamping the samples to `Short.MAX_VALUE` or `Short.MIN_VALUE` to do a quick and dirty square wave example.

Of course, this just scratches the surface of what can be done with audio synthesis on Android. Given `AudioTrack` allows us to play raw PCM samples, almost any technique that can be used to generate digital audio can be utilized on Android, taking into account processor speed and memory limitations.

What follows is an example application that takes some techniques from Chapter 4 for tracking finger position on the touchscreen and the foregoing example code for

generating audio. In this application, we'll generate audio and choose the frequency based upon the location of the user's finger on the x axis of the touchscreen.

```
package com.apress.proandroidmedia.ch08.fingersynthesis;

import android.app.Activity;
import android.media.AudioFormat;
import android.media.AudioManager;
import android.media.AudioTrack;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;
```

Our activity will implement `OnTouchListener` so that we can track the touch locations.

```
public class FingerSynthesis extends Activity implements OnTouchListener {
```

Just like the previous example, we'll use an `AsyncTask` to provide a thread for generating and playing the audio samples.

```
    AudioSynthesisTask audioSynth;
```

We need a base audio frequency that will be played when the finger is at the 0 position on the x axis. This will be lowest frequency played.

```
    static final float BASE_FREQUENCY = 440;
```

We'll be varying the `synth_frequency` float as the finger moves. When we start the app, we'll set it to the `BASE_FREQUENCY`.

```
    float synth_frequency = BASE_FREQUENCY;
```

We'll use the `play` Boolean to determine when we should actually being playing audio or not. It will be controlled by the touch events.

```
    boolean play = false;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.main);
```

In our layout, we have only one item, a `LinearLayout` with the ID of `MainView`. We'll get a reference to this and register the `OnTouchListener` to be our activity. This way our activity's `onTouch` method will be called when the user touches the screen.

```
        View mainView = this.findViewById(R.id.MainView);
        mainView.setOnTouchListener(this);

        audioSynth = new AudioSynthesisTask();
        audioSynth.execute();
    }

    @Override
    public void onPause() {
```

```
        super.onPause();
        play = false;

        finish();
    }
```

Our onTouch method, called when the user touches, stops touching, or drags a finger on the screen, will set the play Boolean to true or false depending on the action of the user. This will control whether audio samples are generated. It will also track the location of the user's finger on the x axis of the touchscreen and adjust the synth_frequency variable accordingly.

```
    public boolean onTouch(View v, MotionEvent event) {
        int action = event.getAction();
        switch (action)
        {
            case MotionEvent.ACTION_DOWN:
                play = true;
                synth_frequency = event.getX() + BASE_FREQUENCY;
                Log.v("FREQUENCY",""+synth_frequency);
                break;
            case MotionEvent.ACTION_MOVE:
                play = true;
                synth_frequency = event.getX() + BASE_FREQUENCY;
                Log.v("FREQUENCY",""+synth_frequency);
                break;
            case MotionEvent.ACTION_UP:
                play = false;
                break;
            case MotionEvent.ACTION_CANCEL:
                break;
            default:
                break;
        }
        return true;
    }

    private class AudioSynthesisTask extends AsyncTask<Void, Void, Void>
    {
        @Override
        protected Void doInBackground(Void... params) {
            final int SAMPLE_RATE= 11025;

            int minSize = AudioTrack.getMinBufferSize(SAMPLE_RATE,
                    AudioFormat.CHANNEL_CONFIGURATION_MONO,
                    AudioFormat.ENCODING_PCM_16BIT);

            AudioTrack audioTrack = new AudioTrack(AudioManager.STREAM_MUSIC,
                    SAMPLE_RATE,
                    AudioFormat.CHANNEL_CONFIGURATION_MONO,
                    AudioFormat.ENCODING_PCM_16BIT,
                    minSize,
                    AudioTrack.MODE_STREAM);

            audioTrack.play();

            short[] buffer = new short[minSize];
```

```
            float angle = 0;
```

Finally, in the `AudioSynthesisTask`, in the loop that generates the audio, we'll check the `play` Boolean and do the calculations to generate the audio samples based on the `synth_frequency` variable, which we are changing based upon the user's finger position.

```
            while (true) {

                if (play)
                {
                    for (int i = 0; i < buffer.length; i++)
                    {
                        float angular_frequency =
                           (float)(2*Math.PI) * synth_frequency / SAMPLE_RATE;

                        buffer[i] =
                           (short)(Short.MAX_VALUE * ((float) Math.sin(angle)));
                        angle += angular_frequency;
                    }
                    audioTrack.write(buffer, 0, buffer.length);
                } else {
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

Here is the layout XML:

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

This example shows some of the power and flexibility of the `AudioTrack` class. Since we can algorithmically generate audio, we can use just about any method we would like to determine its features (its pitch or frequency in this example).

Audio Analysis

Now that we have gone over more advanced ways that `AudioTrack` may be used, how about looking at what else we might do with audio as it comes in through an `AudioRecord` object?

Capturing Sound for Analysis

As previously described, sound is vibration traveling through a substance. These vibrations can be captured by a microphone. Microphones convert the vibrations that travel through air into a constantly varying electrical current. When a microphone is used to capture sound by a computer, that sound is digitized. Specifically, amplitude samples of a specific size (sample size) are taken many times a second (sample rate). This stream of data is called a PCM (pulse code modulation) stream, which forms the foundation for digital audio. Taken all together, the samples represented in the PCM stream digitally represent the audio waveform that is captured. The higher the sample rate, the more accurate the representation and the higher the frequency of audio that can be captured.

As we learned in the previous chapter, when we started working with the `AudioRecord` class, these parameters may be passed into the constructor of the `AudioRecord` class when creating an object. To revisit what each of the parameters means, please see the "Raw Audio Recording with AudioRecord" section in Chapter 7.

> **NOTE:** The Nyquist sampling theorem, named after Harry Nyquist, who was an engineer for Bell Labs in the early to mid-twentieth century, explains that the highest frequency that may be captured by a digitizing system is one half of the sample rate used. Therefore, in order to capture audio at 440 Hz (middle A), our system needs to capture samples at 880 Hz or higher.

Here is a quick recap of the steps required to capture audio using an object of type `AudioRecord`.

```
int frequency = 8000;
int channelConfiguration = AudioFormat.CHANNEL_CONFIGURATION_MONO;
int audioEncoding = AudioFormat.ENCODING_PCM_16BIT;

int bufferSize = AudioRecord.getMinBufferSize(frequency,
                 channelConfiguration, audioEncoding);

AudioRecord audioRecord = new AudioRecord(
                    MediaRecorder.AudioSource.MIC, frequency,
                    channelConfiguration, audioEncoding, bufferSize);

short[] buffer = new short[blockSize];
audioRecord.startRecording();

while (started) {
    int bufferReadResult = audioRecord.read(buffer, 0, blockSize);
}

audioRecord.stop();
```

The foregoing code doesn't actually do anything with the audio that is captured. Normally we would want to write it to a file or to analyze it in some other manner.

Visualizing Frequencies

One common way that people typically use to analyze audio is to visualize the frequencies that exist within it. Commonly these types of visuals are employed with equalizers that allow the adjustment of the levels of various frequency ranges.

The technique used to break an audio signal down into component frequencies employs a mathematic transformation called a discrete Fourier transform (DFT). A DFT is commonly used to translate data from a time base to a frequency base. One algorithm used to perform DFT is a fast Fourier transform (FFT), which is very efficient but unfortunately complex.

Fortunately, many implementations of FFT algorithms exist that are in the public domain or are open source and that we may employ. One such version is a Java port of the FFTPACK library, originally developed by Paul Swarztrauber of the National Center for Atmospheric Research. The Java port was performed by Baoshe Zhang of the University of Lethbridge in Alberta, Canada. Various implementations are available online at `www.netlib.org/fftpack/`. The one we'll be using is archived in a file called `jfftpack.tgz` linked off of that page. It is directly downloadable via `www.netlib.org/fftpack/jfftpack.tgz`.

To use this or any other package containing Java source code in an Eclipse Android project, we need to import the source into our project. This archive contains the correct directory structure for the package, so we just drag the top-level folder in the `javasource` directory (`ca`) into the `src` directory of our project.

Here is an example that draws the graphic portion of a graphic equalizer.

```
package com.apress.proandroidmedia.ch08.audioprocessing;

import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.media.AudioFormat;
import android.media.AudioRecord;
import android.media.MediaRecorder;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.ImageView;
```

We'll import the `RealDoubleFFT` class in the `fftpack` package.

```
import ca.uol.aig.fftpack.RealDoubleFFT;

public class AudioProcessing extends Activity implements OnClickListener {
```

We'll use a frequency of 8 kHz, one audio channel, and 16 bit samples in the `AudioRecord` object.

```
int frequency = 8000;
int channelConfiguration = AudioFormat.CHANNEL_CONFIGURATION_MONO;
int audioEncoding = AudioFormat.ENCODING_PCM_16BIT;
```

`transformer` will be our FFT object, and we'll be dealing with 256 samples at a time from the `AudioRecord` object through the FFT object. The number of samples we use will correspond to the number of component frequencies we will get after we run them through the FFT object. We are free to choose a different size, but we do need concern ourselves with memory and performance issues as the math required to the calculation is processor-intensive.

```
private RealDoubleFFT transformer;
int blockSize = 256;

Button startStopButton;
boolean started = false;
```

`RecordAudio` is an inner class defined here that extends `AsyncTask`.

```
RecordAudio recordTask;
```

We'll be using an `ImageView` to display a `Bitmap` image. This image will represent the levels of the various frequencies that are in the current audio stream. To draw these levels, we'll use `Canvas` and `Paint` objects constructed from the `Bitmap`.

```
ImageView imageView;
Bitmap bitmap;
Canvas canvas;
Paint paint;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    startStopButton = (Button) this.findViewById(R.id.StartStopButton);
    startStopButton.setOnClickListener(this);
```

The `RealDoubleFFT` class constructor takes in the number of samples that we'll deal with at a time. This also represents the number of distinct ranges of frequencies that will be output.

```
    transformer = new RealDoubleFFT(blockSize);
```

Here is the setup of the `ImageView` and related object for drawing.

```
    imageView = (ImageView) this.findViewById(R.id.ImageView01);
    bitmap = Bitmap.createBitmap((int)256,(int)100,Bitmap.Config.ARGB_8888);
    canvas = new Canvas(bitmap);
    paint = new Paint();
    paint.setColor(Color.GREEN);
    imageView.setImageBitmap(bitmap);
}
```

Most of the work in this activity is done in the following class, called `RecordAudio`, which extends `AsyncTask`. Using `AsyncTask`, we run the methods that will tie up the user

interface on a separate thread. Anything that is placed in the doInBackground method will be run in this manner.

```
private class RecordAudio extends AsyncTask<Void, double[], Void> {
    @Override
    protected Void doInBackground(Void... params) {
        try {
```

We'll set up and use AudioRecord in the normal manner.

```
int bufferSize = AudioRecord.getMinBufferSize(frequency,
        channelConfiguration, audioEncoding);

AudioRecord audioRecord = new AudioRecord(
        MediaRecorder.AudioSource.MIC, frequency,
        channelConfiguration, audioEncoding, bufferSize);
```

The short array, buffer, will take in the raw PCM samples from the AudioRecord object. The double array, toTransform, will hold the same data but in the form of doubles, as that is what the FFT class requires.

```
short[] buffer = new short[blockSize];
double[] toTransform = new double[blockSize];

audioRecord.startRecording();

while (started) {
    int bufferReadResult = audioRecord.read(buffer, 0, blockSize);
```

After we read the data from the AudioRecord object, we loop through and translate it from short values to double values. We can't do this directly by casting, as the values expected should be between -1.0 and 1.0 rather than the full range. Dividing the short by 32,768.0 will do that, as that value is the maximum value of short.

> **NOTE:** There is a constant Short.MAX_VALUE that could be used instead.

```
    for (int i = 0; i < blockSize && i < bufferReadResult; i++) {
        toTransform[i] = (double) buffer[i] / 32768.0; // signed 16 bit
    }
```

Next we'll pass the array of double values to the FFT object. The FFT object re-uses the same array to hold the output values. The data contained will be in the frequency domain rather than the time domain. This means that the first element in the array will not represent the first sample in time—rather, it will represent the levels of the first set of frequencies.

Since we are using 256 values (or ranges) and our sample rate is 8,000, we can determine that each element in the array will cover approximately 15.625 Hz. We come up with this figure by dividing the sample rate in half (as the highest frequency we can capture is half the sample rate) and then dividing by 256. Therefore the data represented in the first element of the array will represent the level of audio that is between 0 and 15.625 Hz.

```
    transformer.ft(toTransform);
```

Calling publishProgress calls onProgressUpdate.

```
                publishProgress(toTransform);
            }

            audioRecord.stop();
        } catch (Throwable t) {
            Log.e("AudioRecord", "Recording Failed");
        }

        return null;
    }
```

onProgressUpdate runs on the main thread in our activity and can therefore interact with the user interface without problems. In this implementation, we are passing in the data after it has been run through the FFT object. This method takes care of drawing the data on the screen as a series of lines at most 100 pixels tall. Each line represents one of the elements in the array and therefore a range of 15.625 Hz. The first line represents frequencies ranging from 0 to 15.625 Hz, and the last line represents frequencies ranging from 3,984.375 to 4,000 Hz. Figure 8–1 shows what this looks like in action.

```
    protected void onProgressUpdate(double[]... toTransform) {
        canvas.drawColor(Color.BLACK);

        for (int i = 0; i < toTransform[0].length; i++) {
            int x = i;
            int downy = (int) (100 - (toTransform[0][i] * 10));
            int upy = 100;

            canvas.drawLine(x, downy, x, upy, paint);
        }
        imageView.invalidate();
    }
}

public void onClick(View v) {
    if (started) {
        started = false;
        startStopButton.setText("Start");
        recordTask.cancel(true);
    } else {
        started = true;
        startStopButton.setText("Stop");
        recordTask = new RecordAudio();
        recordTask.execute();
    }
}
}
```

Here is the layout XML file used by the AudioProcessing activity just defined.

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

```
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
    />
    <ImageView android:id="@+id/ImageViewO1" android:layout_width="wrap_content"↩
 android:layout_height="wrap_content"></ImageView><Button android:text="Start"↩
 android:id="@+id/StartStopButton" android:layout_width="wrap_content"↩
 android:layout_height="wrap_content"></Button>
</LinearLayout>
```

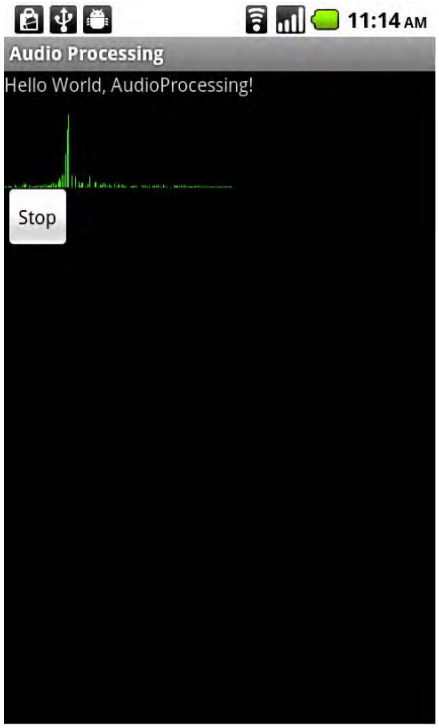


Figure 8–1. *AudioProcessing* ***activity running***

Summary

With this chapter, we have concluded our coverage of audio on Android and have done so by showing how flexible it can be. Although we only scratched the surface of both audio synthesis and analysis, doing so shows the potential of what can be done and how flexible the `AudioTrack` and `AudioRecord` classes in Android are.

Next we'll turn our attention to video.