AudioTrack instances can operate under two modes: **static** and **streaming**. In the streaming mode, your application writes a continuous stream of data to an AudioTrack object. This is done by using one of this class's `.write()` methods. A streaming mode is useful when playing blocks of audio data that are too big to fit in memory because of the duration of the sample, or that too big to fit in memory because of the audio data characteristics (a high sampling rate or sample resolution, or both), or the digital audio sample is received (or synthesized) while previously queued digital audio samples are playing.

The static mode should be chosen when dealing with short sounds that fit into memory and that need to be played with the minimum amount of latency. This static mode should be the preferential mode for user interface feedback or game audio that is triggered frequently by the end user. It's important to note that a SoundPool class may do the same thing with far more memory efficiency, as well as with other features, such as pitch shifting.

Upon instantiation (at the time of creation), your AudioTrack object initializes its associated **audio buffer**. The size of this audio buffer is specified during object construction; it determines how long your AudioTrack can play before running out of memory allocation (space) to hold the audio sample data. For an AudioTrack that is using a static mode, this size is the maximum size of the sound that can be played from it. For the streaming mode, audio data is transferred to the audio sink using data chunks in sizes less than or equal to the total audio data buffer size specification.

Next, let's take a look at MediaPlayer and MediaRecorder classes for long-form audio playback and recording audio and video data streams.

The Android MediaPlayer Class: Digital Audio Playback

Also like Java and JavaFX, Android has a MediaPlayer class that can be used to play long-form audio and video media assets. The MediaPlayer is a complete player solution featuring a transport user interface and all controls necessary to play, stop, seek, reset, or pause new media assets such as digital audio or video.

This MediaPlayer class has a major amount of information attached to it in a programming scenario; the topic warrants an entire book. If you want to learn more about it, try the books *Android Apps for Absolute Beginners* (Apress, 2014) or *Pro Android UI* (Apress, 2014), or visit the Android Developer web site at `http://developer.android.com/reference/android/media/MediaPlayer.html`.

Let's take a look at some Java statements needed to create a URL `String`, instantiate a new **myMediaPlayer** object, set an audio stream type and data source, and go through the **states** associated with an digital audio asset **playback cycle**, including asset preparation, playback (start), pause, stop, reset, remove from memory (release), and nullification, as outlined here:

```
String url = "http://server-address/folder/file-name"; // Audio Asset URL
MediaPlayer myMediaPlayer = new MediaPlayer();
myMediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
myMediaPlayer.setDataSource(url);
```

```
myMediaPlayer.prepare();      // Prepare Audio Asset (buffer from server)
myMediaPlayer.start();       //  Start Playback
myMediaPlayer.pause();      //   Pause Playback
myMediaPlayer.stop();      //    Stop Playback
myMediaPlayer.reset();    //     Reset MediaPlayer object
myMediaPlayer.release(); //      Remove MediaPlayer object from memory
myMediaPlayer = null;   //       Nullify your MediaPlayer object
```

In Android Studio, this Java code creates a **String** object to hold the digital audio asset URL (line one). It declares and instantiates a **MediaPlayer** object (line two). It sets an **AudioStreamType** object using the **STREAM_MUSIC** constant from the **AudioManager** class (line three). It sets the **DataSource** object, the value of your String object (line four). Lines 5 through 11 go through the digital audio asset playback cycle states of preparations (loading into memory), playback start, playback pausing (if needed), playback halting (stop), playback reset, playback release (removing from memory), and object clearing to a null or unutilized state.

If you want to learn Java, check out the *Beginning Java 8 Games Development* (Apress, 2015); it covers Java programming in the context of new media assets, including digital audio.

MediaPlayer plays digital audio and digital video assets, usually in long form (several minutes), which were created and optimized outside of Java or Android using professional software like Audacity 2.1. For digital video editing, you can use the EditShare Lightworks open source software, which you can download at http://www.lwks.com.

The Android MediaRecorder Class: Digital Audio Recording

Unlike Java and JavaFX, Android has its own MediaRecorder class, probably because Android phones have built-in cameras. (Android also has its own Camera (and Camera2) API to control the camera hardware.) The MediaRecorder class can be used to record long-form audio and video, and thus to create new media assets.

The MediaRecorder is a complete media recorder solution, featuring all the controls necessary to start, stop, reset, and release multimedia recording hardware devices.

Let's take a look at some Java statements needed to declare a MediaRecorder object, named **myMediaRecorder**. Use a Java **new** keyword to instantiate this object using a constructor method called **MediaRecorder()**. Set your **AudioSource** constant to **MIC**. Set the **OutputFormat** and **AudioEncoder** constants to **AMR_NB**. Set the **OutputFile** object reference and go through the recording **states** associated with a digital audio data **recording lifecycle**, including memory preparation, start recording, stop recording, and removing a data capture area from memory. All of this is outlined here:

```
MediaRecorder myRecorder;
myRecorder = new MediaRecorder();
myRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
myRecorder.setOutputFormat(MediaRecorder.OutputFormat.AMR_NB);
myRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
myRecorder.setOutputFile(PATH_AND_FILE_NAME_REFERENCE);
```

```
myRecorder.prepare();     // This sets aside system memory for recording
myRecorder.start();      //  This starts the camera hardware recording
myRecorder.stop();      // This stops the camera hardware from recording
myRecorder.reset();    //  You can reuse a reset MediaRecorder object
myRecorder.release(); // Once released a MediaRecorder object can't be used
```

The MediaRecorder class has a huge amount of information attached to using it in a programming scenario; the topic could have an entire book written on it. If you want to learn more about it, visit the Android Developer web site at http://developer.android.com/reference/android/media/MediaRecorder.html.

Summary

In this chapter, you learned about topics that relate to digital audio programming and the programming languages used in digital editing software and app development software. The final chapter in this book covers publishing platforms.

CHAPTER 14

Publishing Digital Audio: Content Delivery Platforms

Now that you have an understanding of the fundamental concepts, terms, and principles behind digital audio editing, compositing, and programming, it is time to look at how digital audio is published on popular open source publishing platforms. I am going to delineate this chapter using **consumer electronics genres**, as these define the different types of applications.

For example, **e-book readers**, or e-readers, such as the Kindle Fire, use Kindle KF8 e-books. Smartwatches use Android Wear SDK under the Android Studio 1.4 and using the Android OS 5.4 API.

iTVs use the Android TV SDK in Android Studio 1.4 with the Android OS 5.4 API.

Automobile dashboards use the Android Auto SDK with the Android Studio 1.4 IDE and the Android OS 5.4 API.

Tablets and smartphones use Android SDK in Android Studio 1.4 IDE with the Android OS 5.4 API.

Laptops and netbooks use Java with JavaFX. These hardware devices support all the open industry-publishing standards, such as PDF, HTML5, and EPUB3, as well as "closed" (not open source) operating systems like Windows and iOS.

iOS runs on the Apple Watch and there is supposed to be Apple TV. iOS also runs on the iPhone and iPad.

Microsoft purchased Nokia, so these smartphones also run Windows. However, all the other major manufacturers in the world use open Android and HTML5 operating systems and platforms, so that is what I will focus on.

You'll continue to look at how to publish on electronic hardware devices using the software development platforms that these devices support, including Kindle KF8, EPUB3, Android Studio 1.4 (Android OS 5.4), Java, JavaFX, PDF, HTML5, CSS3, and JavaScript, many of which were covered in Chapter 13.

W. Jackson, *Digital Audio Editing Fundamentals*, DOI 10.1007/978-1-4842-1648-4_14