MediaPlayer

In this document

Key classes

- `MediaPlayer`
- `AudioManager`
- `SoundPool`

See also

- MediaRecorder
- Supported Media Formats
- Data Storage

The Android multimedia framework includes support for playing variety of common media types, so that you can easily integrate audio, video and images into your applications. You can play audio or video from media files stored in your application's resources (raw resources), from standalone files in the filesystem, or from a data stream arriving over a network connection, all using `MediaPlayer` APIs.

This document shows you how to write a media-playing application that interacts with the user and the system in order to obtain good performance and a pleasant user experience.

> **Note:** You can play back the audio data only to the standard output device. Currently, that is the mobile device speaker or a Bluetooth headset. You cannot play sound files in the conversation audio during a call.

The basics

The following classes are used to play sound and video in the Android framework:

`MediaPlayer`

This class is the primary API for playing sound and video.

`AudioManager`

This class manages audio sources and audio output on a device.

Manifest declarations

Before starting development on your application using MediaPlayer, make sure your manifest has the appropriate declarations to allow use of related features.

- **Internet Permission** - If you are using MediaPlayer to stream network-based content, your application must request network access.

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

- **Wake Lock Permission** - If your player application needs to keep the screen from dimming or the processor from sleeping, or uses the `MediaPlayer.setScreenOnWhilePlaying()` or `MediaPlayer.setWakeMode()` methods, you must request this permission.

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

Using MediaPlayer

One of the most important components of the media framework is the `MediaPlayer` class. An object of this class can fetch, decode, and play both audio and video with minimal setup. It supports several different media sources such as:

- Local resources
- Internal URIs, such as one you might obtain from a Content Resolver
- External URLs (streaming)

For a list of media formats that Android supports, see the Supported Media Formats page.

Here is an example of how to play audio that's available as a local raw resource (saved in your application's `res/raw/` directory):

```
MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.sound_file_1);
mediaPlayer.start(); // no need to call prepare(); create() does that for you
```

In this case, a "raw" resource is a file that the system does not try to parse in any particular way. However, the content of this resource should not be raw audio. It should be a properly encoded and formatted media file in one of the supported formats.

And here is how you might play from a URI available locally in the system (that you obtained through a Content Resolver, for instance):

```
Uri myUri = ....; // initialize Uri here
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(getApplicationContext(), myUri);
mediaPlayer.prepare();
mediaPlayer.start();
```

Playing from a remote URL via HTTP streaming looks like this:

```
String url = "http://........"; // your URL here
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(url);
mediaPlayer.prepare(); // might take long! (for buffering, etc)
mediaPlayer.start();
```

> **Note:** If you're passing a URL to stream an online media file, the file must be capable of progressive download.

> **Caution:** You must either catch or pass `IllegalArgumentException` and `IOException` when using `setDataSource()`, because the file you are referencing might not exist.

Asynchronous preparation

Using `MediaPlayer` can be straightforward in principle. However, it's important to keep in mind that a few more things are necessary to integrate it correctly with a typical Android application. For example, the call to `prepare()` can take a long time to execute, because it might involve fetching and decoding media data. So, as is the case with any method that may take long to execute, you should **never call it from your application's UI thread**. Doing that will cause the UI to hang until the method returns, which is a very bad user experience and can cause an ANR (Application Not Responding) error. Even if you expect your resource to load quickly, remember that anything that takes more than a tenth of a second to respond in the UI will cause a noticeable pause and will give the user the impression that your application is slow.

To avoid hanging your UI thread, spawn another thread to prepare the `MediaPlayer` and notify the main thread when done. However, while you could write the threading logic yourself, this pattern is so common when using `MediaPlayer` that the framework supplies a convenient way to accomplish this task by using the `prepareAsync()` method. This method starts preparing the media in the background and returns immediately. When the media is done preparing, the `onPrepared()` method of the `MediaPlayer.OnPreparedListener`, configured through `setOnPreparedListener()` is called.

Managing state

Another aspect of a `MediaPlayer` that you should keep in mind is that it's state-based. That is, the `MediaPlayer` has an internal state that you must always be aware of when writing your code, because certain operations are only valid when then player is in specific states. If you perform an operation while in the wrong state, the system may throw an exception or cause other undesirable behaviors.

The documentation in the `MediaPlayer` class shows a complete state diagram, that clarifies which methods move the `MediaPlayer` from one state to another. For example, when you create a new `MediaPlayer`, it is in the *Idle* state. At that point, you should initialize it by calling `setDataSource()`, bringing it to the *Initialized* state. After that, you have to prepare it using either the `prepare()` or `prepareAsync()` method. When the `MediaPlayer` is done preparing, it will then enter the *Prepared* state, which means you can call `start()` to make it play the media. At that point, as the diagram illustrates, you can move between the *Started*, *Paused* and *PlaybackCompleted* states by calling such methods as `start()`, `pause()`, and `seekTo()`, amongst others. When you call `stop()`, however, notice that you cannot call `start()` again until you prepare the `MediaPlayer` again.

Always keep the state diagram in mind when writing code that interacts with a `MediaPlayer` object, because calling its methods from the wrong state is a common cause of bugs.

Releasing the MediaPlayer

A `MediaPlayer` can consume valuable system resources. Therefore, you should always take extra precautions to make sure you are not hanging on to a `MediaPlayer` instance longer than necessary. When you are done with it, you should always call `release()` to make sure any system resources allocated to it are properly released. For example, if you are using a `MediaPlayer` and your activity receives a call to `onStop()`, you must release the `MediaPlayer`, because it makes little sense to hold on to it while your activity is not interacting with the user (unless you are playing media in the background, which is discussed in the next section). When your activity is resumed or restarted, of course, you need to create a new `MediaPlayer` and prepare it again before resuming playback.

Here's how you should release and then nullify your `MediaPlayer`:

```
mediaPlayer.release();
mediaPlayer = null;
```

As an example, consider the problems that could happen if you forgot to release the `MediaPlayer` when your activity is stopped, but create a new one when the activity starts again. As you may know, when the user changes the screen orientation (or changes the device configuration in another way), the system handles that by restarting the activity (by default), so you might quickly consume all of the system resources as the user rotates the device back and forth between portrait and landscape, because at each orientation change, you create a new `MediaPlayer` that you never release. (For more information about runtime restarts, see Handling Runtime Changes.)

You may be wondering what happens if you want to continue playing "background media" even when the user leaves your activity, much in the same way that the built-in Music application behaves. In this case, what you need is a `MediaPlayer` controlled by a Service, as discussed in the next section

Using MediaPlayer in a service

If you want your media to play in the background even when your application is not onscreen—that is, you want it to continue playing while the user is interacting with other applications—then you must start a Service and control the `MediaPlayer` instance from there. You need to embed the MediaPlayer in a `MediaBrowserServiceCompat` service and have it interact with a `MediaBrowserCompat` in another activity.

You should be careful about this client/server setup. There are expectations about how a player running in a background service interacts with the rest of the system. If your application does not fulfill those expectations, the user may have a bad experience. Read Building an Audio App for the full details.

This section describes special instructions for managing a MediaPlayer when it is implemented inside a service.

Running asynchronously

First of all, like an `Activity`, all work in a `Service` is done in a single thread by default—in fact, if you're running an activity and a service from the same application, they use the same thread (the "main thread") by default. Therefore, services need to process incoming intents quickly and never perform lengthy computations when responding to them. If any heavy work or blocking calls are expected, you must do those tasks asynchronously: either from another thread you implement yourself, or using the framework's many facilities for asynchronous processing.

For instance, when using a `MediaPlayer` from your main thread, you should call `prepareAsync()` rather than `prepare()`, and implement a `MediaPlayer.OnPreparedListener` in order to be notified when the preparation is complete and you can start playing. For example:

```
public class MyService extends Service implements MediaPlayer.OnPreparedListener {
    private static final String ACTION_PLAY = "com.example.action.PLAY";
    MediaPlayer mMediaPlayer = null;

    public int onStartCommand(Intent intent, int flags, int startId) {
        ...
        if (intent.getAction().equals(ACTION_PLAY)) {
            mMediaPlayer = ... // initialize it here
            mMediaPlayer.setOnPreparedListener(this);
            mMediaPlayer.prepareAsync(); // prepare async to not block main thread
        }
    }

    /** Called when MediaPlayer is ready */
    public void onPrepared(MediaPlayer player) {
        player.start();
    }
}
```

Handling asynchronous errors

On synchronous operations, errors would normally be signaled with an exception or an error code, but whenever you use asynchronous resources, you should make sure your application is notified of errors appropriately. In the case of a `MediaPlayer`, you can accomplish this by implementing a `MediaPlayer.OnErrorListener` and setting it in your `MediaPlayer` instance:

```
public class MyService extends Service implements MediaPlayer.OnErrorListener {
    MediaPlayer mMediaPlayer;

    public void initMediaPlayer() {
        // ...initialize the MediaPlayer here...

        mMediaPlayer.setOnErrorListener(this);
    }

    @Override
    public boolean onError(MediaPlayer mp, int what, int extra) {
        // ... react appropriately ...
        // The MediaPlayer has moved to the Error state, must be reset!
    }
}
```

It's important to remember that when an error occurs, the `MediaPlayer` moves to the *Error* state (see the documentation for the `MediaPlayer` class for the full state diagram) and you must reset it before you can use it again.

Using wake locks

When designing applications that play media in the background, the device may go to sleep while your service is running. Because the Android system tries to conserve battery while the device is sleeping, the system tries to shut off any of the phone's features that are not necessary, including the CPU and the WiFi hardware. However, if your service is playing or streaming music, you want to prevent the system from interfering with your playback.

In order to ensure that your service continues to run under those conditions, you have to use "wake locks." A wake lock is a way to signal to the system that your application is using some feature that should stay available even if the phone is idle.

> **Notice:** You should always use wake locks sparingly and hold them only for as long as truly necessary, because they significantly reduce the battery life of the device.

To ensure that the CPU continues running while your `MediaPlayer` is playing, call the `setWakeMode()` method when initializing your `MediaPlayer`. Once you do, the `MediaPlayer` holds the specified lock while playing and releases the lock when paused or stopped:

```
mMediaPlayer = new MediaPlayer();
// ... other initialization here ...
mMediaPlayer.setWakeMode(getApplicationContext(), PowerManager.PARTIAL_WAKE_LOCK);
```

However, the wake lock acquired in this example guarantees only that the CPU remains awake. If you are streaming media over the network and you are using Wi-Fi, you probably want to hold a `WifiLock` as well, which you must acquire and release manually. So, when you start preparing the `MediaPlayer` with the remote URL, you should create and acquire the Wi-Fi lock. For example:

```
WifiLock wifiLock = ((WifiManager) getSystemService(Context.WIFI_SERVICE))
    .createWifiLock(WifiManager.WIFI_MODE_FULL, "mylock");

wifiLock.acquire();
```

When you pause or stop your media, or when you no longer need the network, you should release the lock:

```
wifiLock.release();
```

Performing cleanup

As mentioned earlier, a `MediaPlayer` object can consume a significant amount of system resources, so you should keep it only for as long as you need and call `release()` when you are done with it. It's important to call this cleanup method explicitly rather than rely on system garbage collection because it might take some time before the garbage collector reclaims the `MediaPlayer`, as it's only sensitive to memory needs and not to shortage of other media-related resources. So, in the case when you're using a service, you should always override the `onDestroy()` method to make sure you are releasing the `MediaPlayer`:

```
public class MyService extends Service {
   MediaPlayer mMediaPlayer;
   // ...

   @Override
   public void onDestroy() {
       if (mMediaPlayer != null) mMediaPlayer.release();
   }
}
```

You should always look for other opportunities to release your `MediaPlayer` as well, apart from releasing it when being shut down. For example, if you expect not to be able to play media for an extended period of time (after losing audio focus, for example), you should definitely release your existing `MediaPlayer` and create it again later. On the other hand, if you only expect to stop playback for a very short time, you should probably hold on to your `MediaPlayer` to avoid the overhead of creating and preparing it again.

Retrieving media from a ContentResolver

Another feature that may be useful in a media player application is the ability to retrieve music that the user has on the device. You can do that by querying the `ContentResolver` for external media:

```
ContentResolver contentResolver = getContentResolver();
Uri uri = android.provider.MediaStore.Audio.Media.EXTERNAL_CONTENT_URI;
Cursor cursor = contentResolver.query(uri, null, null, null, null);
if (cursor == null) {
    // query failed, handle error.
} else if (!cursor.moveToFirst()) {
    // no media on the device
} else {
    int titleColumn = cursor.getColumnIndex(android.provider.MediaStore.Audio.Media.TITLE);
    int idColumn = cursor.getColumnIndex(android.provider.MediaStore.Audio.Media._ID);
    do {
       long thisId = cursor.getLong(idColumn);
       String thisTitle = cursor.getString(titleColumn);
       // ...process entry...
    } while (cursor.moveToNext());
}
```

To use this with the `MediaPlayer`, you can do this:

```
long id = /* retrieve it from somewhere */;
Uri contentUri = ContentUris.withAppendedId(
        android.provider.MediaStore.Audio.Media.EXTERNAL_CONTENT_URI, id);

mMediaPlayer = new MediaPlayer();
mMediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mMediaPlayer.setDataSource(getApplicationContext(), contentUri);

// ...prepare and start...
```