public class

Summary: Nested Classes | Constants | Ctors | Methods | Protected Methods | Inherited Methods | [Expand All]

Added in API level 1

MediaPlayer

extends Object

java.lang.Object
↳android.media.MediaPlayer

Class Overview

MediaPlayer class can be used to control playback of audio/video files and streams. An example on how to use the methods in this class can be found in `VideoView (/reference/android/widget/VideoView.html)`.

Topics covered here are:

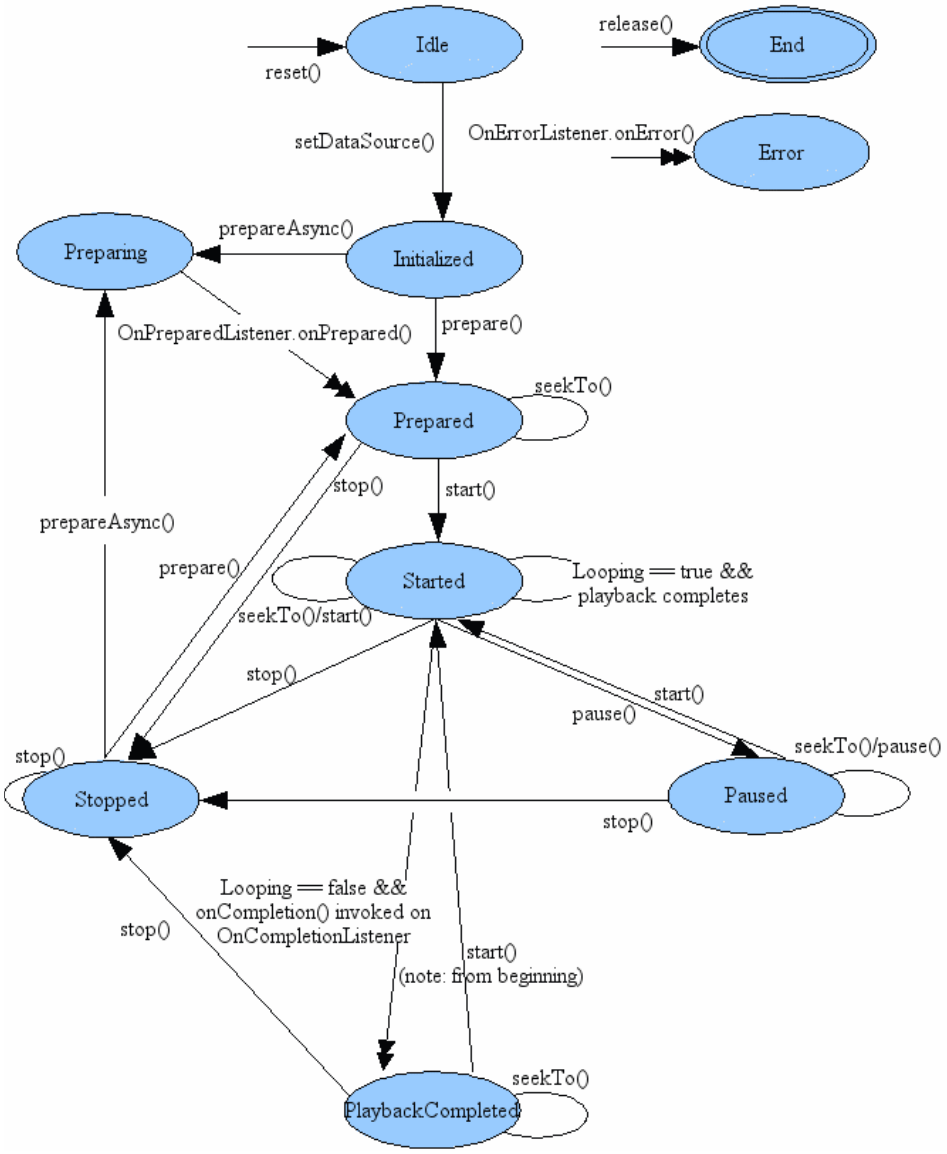
1. State Diagram
2. Valid and Invalid States
3. Permissions
4. Register informational and error callbacks

Developer Guides

For more information about how to use MediaPlayer, read the Media Playback (/guide/topics/media/mediaplayer.html) developer guide.

State Diagram

Playback control of audio/video files and streams is managed as a state machine. The following diagram shows the life cycle and the states of a MediaPlayer object driven by the supported playback control operations. The ovals represent the states a MediaPlayer object may reside in. The arcs represent the playback control operations that drive the object state transition. There are two types of arcs. The arcs with a single arrow head represent synchronous method calls, while those with a double arrow head represent asynchronous method calls.



From this state diagram, one can see that a MediaPlayer object has the following states:

- When a MediaPlayer object is just created using new or after `reset()` is called, it is in the *Idle* state; and after `release()` is called, it is in the *End* state. Between these two states is the life cycle of the MediaPlayer object.
  - There is a subtle but important difference between a newly constructed MediaPlayer object and the MediaPlayer object after `reset()` is called. It is a programming error to invoke methods such as `getCurrentPosition()`, `getDuration()`, `getVideoHeight()`, `getVideoWidth()`, `setAudioStreamType(int)`, `setLooping(boolean)`, `setVolume(float, float)`, `pause()`, `start()`, `stop()`, `seekTo(int)`, `prepare()` or `prepareAsync()` in the *Idle* state for both cases. If any of these methods is called right after a MediaPlayer object is constructed, the user supplied callback method

OnErrorListener.onError() won't be called by the internal player engine and the object state remains unchanged; but if these methods are called right after `reset()`, the user supplied callback method OnErrorListener.onError() will be invoked by the internal player engine and the object will be transfered to the *Error* state.

- It is also recommended that once a MediaPlayer object is no longer being used, call `release()` immediately so that resources used by the internal player engine associated with the MediaPlayer object can be released immediately. Resource may include singleton resources such as hardware acceleration components and failure to call `release()` may cause subsequent instances of MediaPlayer objects to fallback to software implementations or fail altogether. Once the MediaPlayer object is in the *End* state, it can no longer be used and there is no way to bring it back to any other state.
- Furthermore, the MediaPlayer objects created using new is in the *Idle* state, while those created with one of the overloaded convenient `create` methods are *NOT* in the *Idle* state. In fact, the objects are in the *Prepared* state if the creation using `create` method is successful.

- In general, some playback control operation may fail due to various reasons, such as unsupported audio/video format, poorly interleaved audio/video, resolution too high, streaming timeout, and the like. Thus, error reporting and recovery is an important concern under these circumstances. Sometimes, due to programming errors, invoking a playback control operation in an invalid state may also occur. Under all these error conditions, the internal player engine invokes a user supplied OnErrorListener.onError() method if an OnErrorListener has been registered beforehand via `setOnErrorListener(android.media.MediaPlayer.OnErrorListener)`.
- It is important to note that once an error occurs, the MediaPlayer object enters the *Error* state (except as noted above), even if an error listener has not been registered by the application.
- In order to reuse a MediaPlayer object that is in the *Error* state and recover from the error, `reset()` can be called to restore the object to its *Idle* state.
- It is good programming practice to have your application register a OnErrorListener to look out for error notifications from the internal player engine.
- IllegalStateException is thrown to prevent programming errors such as calling `prepare()`, `prepareAsync()`, or one of the overloaded `setDataSource` methods in an invalid state.

- Calling `setDataSource(FileDescriptor)`, or `setDataSource(String)`, or `setDataSource(Context, Uri)`, or `setDataSource(FileDescriptor, long, long)` transfers a MediaPlayer object in the *Idle* state to the *Initialized* state.
- An IllegalStateException is thrown if setDataSource() is called in any other state.
- It is good programming practice to always look out for

`IllegalArgumentException` and `IOException` that may be thrown from the overloaded `setDataSource` methods.

- A MediaPlayer object must first enter the *Prepared* state before playback can be started.
  - There are two ways (synchronous vs. asynchronous) that the *Prepared* state can be reached: either a call to `prepare()` (synchronous) which transfers the object to the *Prepared* state once the method call returns, or a call to `prepareAsync()` (asynchronous) which first transfers the object to the *Preparing* state after the call returns (which occurs almost right way) while the internal player engine continues working on the rest of preparation work until the preparation work completes. When the preparation completes or when `prepare()` call returns, the internal player engine then calls a user supplied callback method, onPrepared() of the OnPreparedListener interface, if an OnPreparedListener is registered beforehand via `setOnPreparedListener(android.media.MediaPlayer.OnPreparedListener)`.
  - It is important to note that the *Preparing* state is a transient state, and the behavior of calling any method with side effect while a MediaPlayer object is in the *Preparing* state is undefined.
  - An IllegalStateException is thrown if `prepare()` or `prepareAsync()` is called in any other state.
  - While in the *Prepared* state, properties such as audio/sound volume, screenOnWhilePlaying, looping can be adjusted by invoking the corresponding set methods.
- To start the playback, `start()` must be called. After `start()` returns successfully, the MediaPlayer object is in the *Started* state. `isPlaying()` can be called to test whether the MediaPlayer object is in the *Started* state.
  - While in the *Started* state, the internal player engine calls a user supplied OnBufferingUpdateListener.onBufferingUpdate() callback method if a OnBufferingUpdateListener has been registered beforehand via `setOnBufferingUpdateListener(OnBufferingUpdateListener)`. This callback allows applications to keep track of the buffering status while streaming audio/video.
  - Calling `start()` has not effect on a MediaPlayer object that is already in the *Started* state.
- Playback can be paused and stopped, and the current playback position can be adjusted. Playback can be paused via `pause()`. When the call to `pause()` returns, the MediaPlayer object enters the *Paused* state. Note that the transition from the *Started* state to the *Paused* state and vice versa happens asynchronously in the player engine. It may take some time before the state is updated in calls to `isPlaying()`, and it can be a number of seconds in the case of streamed content.
  - Calling `start()` to resume playback for a paused MediaPlayer object, and the resumed playback position is the same as where it was paused. When the call to `start()` returns, the paused MediaPlayer object goes back to the *Started* state.

- Calling pause() has no effect on a MediaPlayer object that is already in the *Paused* state.
- Calling stop() stops playback and causes a MediaPlayer in the *Started*, *Paused*, *Prepared* or *PlaybackCompleted* state to enter the *Stopped* state.
- Once in the *Stopped* state, playback cannot be started until prepare() or prepareAsync() are called to set the MediaPlayer object to the *Prepared* state again.
- Calling stop() has no effect on a MediaPlayer object that is already in the *Stopped* state.
- The playback position can be adjusted with a call to seekTo(int).
- Although the asynchronuous seekTo(int) call returns right way, the actual seek operation may take a while to finish, especially for audio/video being streamed. When the actual seek operation completes, the internal player engine calls a user supplied OnSeekComplete.onSeekComplete() if an OnSeekCompleteListener has been registered beforehand via setOnSeekCompleteListener(OnSeekCompleteListener).
- Please note that seekTo(int) can also be called in the other states, such as *Prepared*, *Paused* and *PlaybackCompleted* state.
- Furthermore, the actual current playback position can be retrieved with a call to getCurrentPosition(), which is helpful for applications such as a Music player that need to keep track of the playback progress.
- When the playback reaches the end of stream, the playback completes.
- If the looping mode was being set to *true*with setLooping(boolean), the MediaPlayer object shall remain in the *Started* state.
- If the looping mode was set to *false* , the player engine calls a user supplied callback method, OnCompletion.onCompletion(), if a OnCompletionListener is registered beforehand via setOnCompletionListener(OnCompletionListener). The invoke of the callback signals that the object is now in the *PlaybackCompleted* state.
- While in the *PlaybackCompleted* state, calling start() can restart the playback from the beginning of the audio/video source.

Valid and invalid states

Method Name	Valid Sates	Invalid States	Comments
attachAuxEffect	{Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted}	{Idle, Error}	This method must b called after setDataSource. Calling it does not change the object state.

getAudioSessionId	any	{}	This method can be called in any state and calling it does not change the obje state.
getCurrentPosition	{Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted}	{Error}	Successful invoke o this method in a vali state does not change the state. Calling this method i an invalid state transfers the object to the *Error* state.
getDuration	{Prepared, Started, Paused, Stopped, PlaybackCompleted}	{Idle, Initialized, Error}	Successful invoke o this method in a vali state does not change the state. Calling this method i an invalid state transfers the object to the *Error* state.
getVideoHeight	{Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted}	{Error}	Successful invoke o this method in a vali state does not change the state. Calling this method i an invalid state transfers the object to the *Error* state.
getVideoWidth	{Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted}	{Error}	Successful invoke o this method in a vali state does not change the state. Calling this method i an invalid state transfers the object to the *Error* state.
isPlaying	{Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted}	{Error}	Successful invoke o this method in a vali state does not change the state. Calling this method i an invalid state transfers the object

			to the *Error* state.
pause	{Started, Paused, PlaybackCompleted}	{Idle, Initialized, Prepared, Stopped, Error}	Successful invoke o this method in a vali state transfers the object to the *Paused* state. Calling this method in an invalid state transfers the object to the *Error* state.
prepare	{Initialized, Stopped}	{Idle, Prepared, Started, Paused, PlaybackCompleted, Error}	Successful invoke o this method in a vali state transfers the object to the *Prepare* state. Calling this method in an invalid state throws an IllegalStateExceptio
prepareAsync	{Initialized, Stopped}	{Idle, Prepared, Started, Paused, PlaybackCompleted, Error}	Successful invoke o this method in a vali state transfers the object to the *Preparing* state. Calling this method i an invalid state throws an IllegalStateExceptio
release	any	{}	After `release()`, th object is no longer available.
reset	{Idle, Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted, Error}	{}	After `reset()`, the object is like being just created.
seekTo	{Prepared, Started, Paused, PlaybackCompleted}	{Idle, Initialized, Stopped, Error}	Successful invoke o this method in a vali state does not change the state. Calling this method i an invalid state transfers the object to the *Error* state.

setAudioSessionId	{Idle}	{Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted, Error}	This method must b called in idle state a the audio session ID must be known before calling setDataSource. Calling it does not change the object state.
setAudioStreamType	{Idle, Initialized, Stopped, Prepared, Started, Paused, PlaybackCompleted}	{Error}	Successful invoke o this method does no change the state. In order for the target audio stream type to become effective, this method must be called before prepare() or prepareAsync().
setAuxEffectSendLevel	any	{}	Calling this method does not change the object state.
setDataSource	{Idle}	{Initialized, Prepared, Started, Paused, Stopped, PlaybackCompleted, Error}	Successful invoke o this method in a vali state transfers the object to the *Initialized* state. Calling this method i an invalid state throws an IllegalStateExceptio
setDisplay	any	{}	This method can be called in any state and calling it does not change the obje state.
setSurface	any	{}	This method can be called in any state and calling it does not change the obje state.
setVideoScalingMode	{Initialized, Prepared, Started, Paused, Stopped,	{Idle, Error}	Successful invoke o this method does no change the state.

	PlaybackCompleted}		
setLooping	{Idle, Initialized, Stopped, Prepared, Started, Paused, PlaybackCompleted}	{Error}	Successful invoke o this method in a vali state does not change the state. Calling this method i an invalid state transfers the object to the *Error* state.
isLooping	any	{}	This method can be called in any state and calling it does not change the obje state.
setOnBufferingUpdateListener	any	{}	This method can be called in any state and calling it does not change the obje state.
setOnCompletionListener	any	{}	This method can be called in any state and calling it does not change the obje state.
setOnErrorListener	any	{}	This method can be called in any state and calling it does not change the obje state.
setOnPreparedListener	any	{}	This method can be called in any state and calling it does not change the obje state.
setOnSeekCompleteListener	any	{}	This method can be called in any state and calling it does not change the obje state.
setScreenOnWhilePlaying	any	{}	This method can be called in any state and calling it does

			not change the obje state.
setVolume	{Idle, Initialized, Stopped, Prepared, Started, Paused, PlaybackCompleted}	{Error}	Successful invoke o this method does no change the state.
setWakeMode	any	{}	This method can be called in any state and calling it does not change the obje state.
start	{Prepared, Started, Paused, PlaybackCompleted}	{Idle, Initialized, Stopped, Error}	Successful invoke o this method in a vali state transfers the object to the *Started* state. Calling this method in an invalid state transfers the object to the *Error* state.
stop	{Prepared, Started, Stopped, Paused, PlaybackCompleted}	{Idle, Initialized, Error}	Successful invoke o this method in a vali state transfers the object to the *Stoppe* state. Calling this method in an invalid state transfers the object to the *Error* state.
getTrackInfo	{Prepared, Started, Stopped, Paused, PlaybackCompleted}	{Idle, Initialized, Error}	Successful invoke o this method does no change the state.
addTimedTextSource	{Prepared, Started, Stopped, Paused, PlaybackCompleted}	{Idle, Initialized, Error}	Successful invoke o this method does no change the state.
selectTrack	{Prepared, Started, Stopped, Paused, PlaybackCompleted}	{Idle, Initialized, Error}	Successful invoke o this method does no change the state.
deselectTrack	{Prepared, Started, Stopped, Paused, PlaybackCompleted}	{Idle, Initialized, Error}	Successful invoke o this method does no change the state.

Permissions

One may need to declare a corresponding WAKE_LOCK permission <uses-permission> (/reference/android/R.styleable.html#AndroidManifestUsesPermission) element.

This class requires the INTERNET (/reference/android/Manifest.permission.html#INTERNET) permission when used with network-based content. ()

Callbacks

Applications may want to register for informational and error events in order to be informed of some internal state update and possible runtime errors during playback or streaming. Registration for these events is done by properly setting the appropriate listeners (via calls to setOnPreparedListener(OnPreparedListener) (/reference/android/media/MediaPlayer.html#setOnPreparedListener(android.media.MediaPlayer.OnPreparedListener))setOnPreparedListener, setOnVideoSizeChangedListener(OnVideoSizeChangedListener) (/reference/android/media/MediaPlayer.html#setOnVideoSizeChangedListener(android.media.MediaPlayer.OnVideoSizeChangedListener))setOnVideoSizeChangedListener, setOnSeekCompleteListener(OnSeekCompleteListener) (/reference/android/media/MediaPlayer.html#setOnSeekCompleteListener(android.media.MediaPlayer.OnSeekCompleteListener))setOnSeekCompleteListener, setOnCompletionListener(OnCompletionListener) (/reference/android/media/MediaPlayer.html#setOnCompletionListener(android.media.MediaPlayer.OnCompletionListener))setOnCompletionListener, setOnBufferingUpdateListener(OnBufferingUpdateListener) (/reference/android/media/MediaPlayer.html#setOnBufferingUpdateListener(android.media.MediaPlayer.OnBufferingUpdateListener))setOnBufferingUpdateListener, setOnInfoListener(OnInfoListener) (/reference/android/media/MediaPlayer.html#setOnInfoListener(android.media.MediaPlayer.OnInfoListener))setOnInfoListener, setOnErrorListener(OnErrorListener) (/reference/android/media/MediaPlayer.html#setOnErrorListener(android.media.MediaPlayer.OnErrorListener))setOnErrorListener, etc). In order to receive the respective callback associated with these listeners, applications are required to create MediaPlayer objects on a thread with its own Looper running (main UI

thread by default has a Looper running).

Summary

Nested Classes	
interface MediaPlayer.OnBufferingUpdateListener	Interface definition of a callback to be invoked indicating buffering status of a media resource being streamed over the network.
interface MediaPlayer.OnCompletionListener	Interface definition for a callback to be invoked when playback of a media source has completed.
interface MediaPlayer.OnErrorListener	Interface definition of a callback to be invoked when there has been an error during an asynchronous operation (other errors will throw exceptions at method call time).
interface MediaPlayer.OnInfoListener	Interface definition of a callback to be invoked to communicate some info and/or warning about the media or its playback.
interface MediaPlayer.OnPreparedListener	Interface definition for a callback to be invoked when the media source is ready for

	playback.
interface MediaPlayer.OnSeekCompleteListener	Interface definition of a callback to be invoked indicating the completion of a seek operation.
interface MediaPlayer.OnTimedTextListener	Interface definition of a callback to be invoked when a timed text is available for display.
interface MediaPlayer.OnVideoSizeChangedListener	Interface definition of a callback to be invoked when the video size is first known or updated
class MediaPlayer.TrackInfo	Class for MediaPlayer to return each audio/video /subtitle track's metadata.

Constants

int MEDIA_ERROR_IO	File or network related operation errors.
int MEDIA_ERROR_MALFORMED	Bitstream is not conforming to the related coding standard or file spec.
int MEDIA_ERROR_NOT_VALID_FOR_PROGRESSIVE_PLAYBACK	The video is streamed and its container is not valid for

	progressive playback i.e the video's index (e.g moov atom) is not at the start of the file.
int MEDIA_ERROR_SERVER_DIED	Media server died.
int MEDIA_ERROR_TIMED_OUT	Some operation takes too long to complete, usually more than 3-5 seconds.
int MEDIA_ERROR_UNKNOWN	Unspecified media player error.
int MEDIA_ERROR_UNSUPPORTED	Bitstream is conforming to the related coding standard or file spec, but the media framework does not support the feature.
int MEDIA_INFO_BAD_INTERLEAVING	Bad interleaving means that a media has been improperly interleaved or not interleaved at all, e.g has all the video

	samples first then all the audio ones.
int MEDIA_INFO_BUFFERING_END	MediaPlayer is resuming playback after filling buffers.
int MEDIA_INFO_BUFFERING_START	MediaPlayer is temporarily pausing playback internally in order to buffer more data.
int MEDIA_INFO_METADATA_UPDATE	A new set of metadata is available.
int MEDIA_INFO_NOT_SEEKABLE	The media cannot be seeked (e.g live stream)
int MEDIA_INFO_SUBTITLE_TIMED_OUT	Reading the subtitle track takes too long.
int MEDIA_INFO_UNKNOWN	Unspecified media player info.
int MEDIA_INFO_UNSUPPORTED_SUBTITLE	Subtitle track was not supported by the media framework.
int MEDIA_INFO_VIDEO_RENDERING_START	The player just pushed the very first video frame for rendering.

int MEDIA_INFO_VIDEO_TRACK_LAGGING	The video is too complex for the decoder: it can't decode frames fast enough.
String MEDIA_MIMETYPE_TEXT_SUBRIP	MIME type for SubRip (SRT) container.
int VIDEO_SCALING_MODE_SCALE_TO_FIT	Specifies a video scaling mode.
int VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING	Specifies a video scaling mode.

Public Constructors

MediaPlayer()
Default constructor.

Public Methods

void	addTimedTextSource(Context context, Uri uri, String mimeType) Adds an external timed text source file (Uri).
void	addTimedTextSource(String path, String mimeType) Adds an external timed text source file.
void	addTimedTextSource(FileDescriptor fd, long offset, long length, String m Adds an external timed text file (FileDescriptor).
void	addTimedTextSource(FileDescriptor fd, String mimeType) Adds an external timed text source file (FileDescriptor).
void	attachAuxEffect(int effectId) Attaches an auxiliary effect to the player.
static MediaPlayer	create(Context context, Uri uri, SurfaceHolder holder) Convenience method to create a MediaPlayer for a given Uri.
static MediaPlayer	create(Context context, int resid) Convenience method to create a MediaPlayer for a given resource id.
static MediaPlayer	create(Context context, Uri uri) Convenience method to create a MediaPlayer for a given Uri.
void	deselectTrack(int index) Deselect a track.

int	getAudioSessionId () Returns the audio session ID.
int	getCurrentPosition () Gets the current playback position.
int	getDuration () Gets the duration of the file.
TrackInfo[]	getTrackInfo () Returns an array of track information.
int	getVideoHeight () Returns the height of the video.
int	getVideoWidth () Returns the width of the video.
boolean	isLooping () Checks whether the MediaPlayer is looping or non-looping.
boolean	isPlaying () Checks whether the MediaPlayer is playing.
void	pause () Pauses playback.
void	prepare () Prepares the player for playback, synchronously.
void	prepareAsync () Prepares the player for playback, asynchronously.
void	release () Releases resources associated with this MediaPlayer object.
void	reset () Resets the MediaPlayer to its uninitialized state.
void	seekTo (int msec) Seeks to specified time position.
void	selectTrack (int index) Selects a track.
void	setAudioSessionId (int sessionId) Sets the audio session ID.
void	setAudioStreamType (int streamtype) Sets the audio stream type for this MediaPlayer.
void	setAuxEffectSendLevel (float level) Sets the send level of the player to the attached auxiliary effect .
void	setDataSource (String path) Sets the data source (file-path or http/rtsp URL) to use.
void	setDataSource (Context context, Uri uri, Map<String, String> headers) Sets the data source as a content Uri.

void	setDataSource (Context context, Uri uri) Sets the data source as a content Uri.
void	setDataSource (FileDescriptor fd, long offset, long length) Sets the data source (FileDescriptor) to use.
void	setDataSource (FileDescriptor fd) Sets the data source (FileDescriptor) to use.
void	setDisplay (SurfaceHolder sh) Sets the `SurfaceHolder` to use for displaying the video portion of th
void	setLooping (boolean looping) Sets the player to be looping or non-looping.
void	setNextMediaPlayer (MediaPlayer next) Set the MediaPlayer to start when this MediaPlayer finishes playback
void	setOnBufferingUpdateListener (MediaPlayer.OnBufferingUpdateListener Register a callback to be invoked when the status of a network stream has changed.
void	setOnCompletionListener (MediaPlayer.OnCompletionListener listener) Register a callback to be invoked when the end of a media source has reached during playback.
void	setOnErrorListener (MediaPlayer.OnErrorListener listener) Register a callback to be invoked when an error has happened during a asynchronous operation.
void	setOnInfoListener (MediaPlayer.OnInfoListener listener) Register a callback to be invoked when an info/warning is available.
void	setOnPreparedListener (MediaPlayer.OnPreparedListener listener) Register a callback to be invoked when the media source is ready for p
void	setOnSeekCompleteListener (MediaPlayer.OnSeekCompleteListener liste Register a callback to be invoked when a seek operation has been com
void	setOnTimedTextListener (MediaPlayer.OnTimedTextListener listener) Register a callback to be invoked when a timed text is available for dis
void	setOnVideoSizeChangedListener (MediaPlayer.OnVideoSizeChangedList Register a callback to be invoked when the video size is known or upd
void	setScreenOnWhilePlaying (boolean screenOn) Control whether we should use the attached SurfaceHolder to keep the on while video playback is occurring.
void	setSurface (Surface surface) Sets the `Surface` to be used as the sink for the video portion of the m
void	setVideoScalingMode (int mode) Sets video scaling mode.
void	setVolume (float leftVolume, float rightVolume) Sets the volume on this player.

void setWakeMode(Context context, int mode)
Set the low-level power management behavior for this MediaPlayer.

void start()
Starts or resumes playback.

void stop()
Stops playback after playback has been stopped or paused.

Protected Methods

finalize()
void Invoked when the garbage collector has detected that this instance is no longer reachable.

Inherited Methods [Expand]

▸ From class java.lang.Object

Constants

public static final int **MEDIA_ERROR_IO** Added in API level 17

File or network related operation errors.

Constant Value: -1004 (0xfffffc14)

public static final int **MEDIA_ERROR_MALFORMED** Added in API level 17

Bitstream is not conforming to the related coding standard or file spec.

Constant Value: -1007 (0xfffffc11)

public static final int **MEDIA_ERROR_NOT_VALID_FOR_PROGRESSIVE_PLAYBACK** Added in API level 3

The video is streamed and its container is not valid for progressive playback i.e the video's index (e.g moov atom) is not at the start of the file.

See Also

`MediaPlayer.OnErrorListener`

Constant Value: 200 (0x000000c8)

public static final int **MEDIA_ERROR_SERVER_DIED** Added in API level 1

Media server died. In this case, the application must release the MediaPlayer object and instantiate a new one.

See Also

MediaPlayer.OnErrorListener

Constant Value: 100 (0x00000064)

public static final int **MEDIA_ERROR_TIMED_OUT** Added in API level 17

Some operation takes too long to complete, usually more than 3-5 seconds.

Constant Value: -110 (0xffffff92)

public static final int **MEDIA_ERROR_UNKNOWN** Added in API level 1

Unspecified media player error.

See Also

MediaPlayer.OnErrorListener

Constant Value: 1 (0x00000001)

public static final int **MEDIA_ERROR_UNSUPPORTED** Added in API level 17

Bitstream is conforming to the related coding standard or file spec, but the media framework does not support the feature.

Constant Value: -1010 (0xfffffc0e)

public static final int **MEDIA_INFO_BAD_INTERLEAVING** Added in API level 3

Bad interleaving means that a media has been improperly interleaved or not interleaved at all, e.g has all the video samples first then all the audio ones. Video is playing but a lot of disk seeks may be happening.

See Also

MediaPlayer.OnInfoListener

Constant Value: 800 (0x00000320)

public static final int **MEDIA_INFO_BUFFERING_END** Added in API level 9

MediaPlayer is resuming playback after filling buffers.

See Also

MediaPlayer.OnInfoListener

Constant Value: 702 (0x000002be)

public static final int
MEDIA_INFO_BUFFERING_START Added in API level 9

MediaPlayer is temporarily pausing playback internally in order to buffer more data.

See Also
MediaPlayer.OnInfoListener

Constant Value: 701 (0x000002bd)

public static final int
MEDIA_INFO_METADATA_UPDATE Added in API level 5

A new set of metadata is available.

See Also
MediaPlayer.OnInfoListener

Constant Value: 802 (0x00000322)

public static final int **MEDIA_INFO_NOT_SEEKABLE** Added in API level 3

The media cannot be seeked (e.g live stream)

See Also
MediaPlayer.OnInfoListener

Constant Value: 801 (0x00000321)

public static final int
MEDIA_INFO_SUBTITLE_TIMED_OUT Added in API level 19

Reading the subtitle track takes too long.

See Also
MediaPlayer.OnInfoListener

Constant Value: 902 (0x00000386)

public static final int **MEDIA_INFO_UNKNOWN** Added in API level 3

Unspecified media player info.

See Also
MediaPlayer.OnInfoListener

Constant Value: 1 (0x00000001)

public static final int

MEDIA_INFO_UNSUPPORTED_SUBTITLE Added in API level 19

Subtitle track was not supported by the media framework.

See Also

MediaPlayer.OnInfoListener

Constant Value: 901 (0x00000385)

public static final int
MEDIA_INFO_VIDEO_RENDERING_START Added in API level 17

The player just pushed the very first video frame for rendering.

See Also

MediaPlayer.OnInfoListener

Constant Value: 3 (0x00000003)

public static final int
MEDIA_INFO_VIDEO_TRACK_LAGGING Added in API level 3

The video is too complex for the decoder: it can't decode frames fast enough. Possibly only the audio plays fine at this stage.

See Also

MediaPlayer.OnInfoListener

Constant Value: 700 (0x000002bc)

public static final String
MEDIA_MIMETYPE_TEXT_SUBRIP Added in API level 16

MIME type for SubRip (SRT) container. Used in addTimedTextSource APIs.

Constant Value: "application/x-subrip"

public static final int
VIDEO_SCALING_MODE_SCALE_TO_FIT Added in API level 16

Specifies a video scaling mode. The content is stretched to the surface rendering area. When the surface has the same aspect ratio as the content, the aspect ratio of the content is maintained; otherwise, the aspect ratio of the content is not maintained when video is being rendered. Unlike VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING (/reference/android/media/MediaPlayer.html#VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING), there is no content cropping with this video scaling mode.

Constant Value: 1 (0x00000001)

public static final int
VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING Added in API level 16

Specifies a video scaling mode. The content is scaled, maintaining its aspect ratio. The whole surface area is always used. When the aspect ratio of the content is the same as the surface, no content is cropped; otherwise, content is cropped to fit the surface.

Constant Value: 2 (0x00000002)

Public Constructors

public **MediaPlayer** () Added in API level 1

Default constructor. Consider using one of the create() methods for synchronously instantiating a MediaPlayer from a Uri or resource.

When done with the MediaPlayer, you should call `release()` (/reference/android/media/MediaPlayer.html#release()), to free the resources. If not released, too many MediaPlayer instances may result in an exception.

Public Methods

public void **addTimedTextSource** (Context context, Uri uri, String mimeType) Added in API level 16

Adds an external timed text source file (Uri). Currently supported format is SubRip with the file extension .srt, case insensitive. Note that a single external timed text source may contain multiple tracks in it. One can find the total number of available tracks using `getTrackInfo()` (/reference/android/media/MediaPlayer.html#getTrackInfo()) to see what additional tracks become available after this method call.

Parameters

context	the Context to use when resolving the Uri
uri	the Content URI of the data you want to play
mimeType	The mime type of the file. Must be one of the mime types listed above.

Throws

IOException	if the file cannot be accessed or is corrupted.
IllegalArgumentException	if the mimeType is not supported.
IllegalStateException	if called in an invalid state.

public void **addTimedTextSource** (String path, String mimeType) Added in API level 16

Adds an external timed text source file. Currently supported format is SubRip with the file extension .srt, case insensitive. Note that a single external timed text source may contain multiple tracks in it. One can find the total number of available tracks using `getTrackInfo()` (/reference/android/media/MediaPlayer.html#getTrackInfo()) to see what additional tracks become available after this method call.

Parameters

path	The file path of external timed text source file.
mimeType	The mime type of the file. Must be one of the mime types listed above.

Throws

IOException	if the file cannot be accessed or is corrupted.
IllegalArgumentException	if the mimeType is not supported.
IllegalStateException	if called in an invalid state.

public void **addTimedTextSource** (FileDescriptor fd, long offset, long length, String mimeType) Added in API level 16

Adds an external timed text file (FileDescriptor). It is the caller's responsibility to close the file descriptor. It is safe to do so as soon as this call returns. Currently supported format is SubRip. Note that a single external timed text source may contain multiple tracks in it. One can find the total number of available tracks using `getTrackInfo()` (/reference/android/media/MediaPlayer.html#getTrackInfo()) to see what additional tracks become available after this method call.

Parameters

fd	the FileDescriptor for the file you want to play
offset	the offset into the file where the data to be played starts, in bytes
length	the length in bytes of the data to be played

mimeType	The mime type of the file. Must be one of the mime types listed above.

Throws

IllegalArgumentException	if the mimeType is not supported.
IllegalStateException	if called in an invalid state.

public void **addTimedTextSource** (FileDescriptor fd, String mimeType)

Added in API level 16

Adds an external timed text source file (FileDescriptor). It is the caller's responsibility to close the file descriptor. It is safe to do so as soon as this call returns. Currently supported format is SubRip. Note that a single external timed text source may contain multiple tracks in it. One can find the total number of available tracks using `getTrackInfo()` (/reference/android/media/MediaPlayer.html#getTrackInfo()) to see what additional tracks become available after this method call.

Parameters

fd	the FileDescriptor for the file you want to play
mimeType	The mime type of the file. Must be one of the mime types listed above.

Throws

IllegalArgumentException	if the mimeType is not supported.
IllegalStateException	if called in an invalid state.

public void **attachAuxEffect** (int effectId)

Added in API level 9

Attaches an auxiliary effect to the player. A typical auxiliary effect is a reverberation effect which can be applied on any sound source that directs a certain amount of its energy to this effect. This amount is defined by setAuxEffectSendLevel(). .

After creating an auxiliary effect (e.g. `EnvironmentalReverb` (/reference/android/media/audiofx/EnvironmentalReverb.html)), retrieve its ID with `getId()` (/reference/android/media/audiofx/AudioEffect.html#getId()) and use it when calling this method to attach the player to the effect.

To detach the effect from the player, call this method with a null effect id.

This method must be called after one of the overloaded `setDataSource` methods.

Parameters

effectId	system wide unique id of the effect to attach

See Also

`setAuxEffectSendLevel(float)`

public static MediaPlayer **create** (Context context, Uri uri, SurfaceHolder holder)

Added in API level 1

Convenience method to create a MediaPlayer for a given Uri. On success, `prepare()` will already have been called and must not be called again.

When done with the MediaPlayer, you should call `release()`, to free the resources. If not released, too many MediaPlayer instances will result in an exception.

Parameters

context	the Context to use
uri	the Uri from which to get the datasource
holder	the SurfaceHolder to use for displaying the video

Returns

a MediaPlayer object, or null if creation failed

public static MediaPlayer **create** (Context context, int resid)

Added in API level 1

Convenience method to create a MediaPlayer for a given resource id. On success, `prepare()` will already have been called and must not be called again.

When done with the MediaPlayer, you should call `release()`, to free the resources. If not released, too many MediaPlayer instances will result in an exception.

Parameters

context	the Context to use
resid	the raw resource id (*R.raw.<something>*) for the resource to use as the datasource

Returns

a MediaPlayer object, or null if creation failed

public static MediaPlayer **create** (Context context, Uri uri)

Added in API level 1

Convenience method to create a MediaPlayer for a given Uri. On success, `prepare()` (/reference/android/media/MediaPlayer.html#prepare()) will already have been called and must not be called again.

When done with the MediaPlayer, you should call `release()` (/reference/android/media/MediaPlayer.html#release()), to free the resources. If not released, too many MediaPlayer instances will result in an exception.

Parameters

context	the Context to use
uri	the Uri from which to get the datasource

Returns

a MediaPlayer object, or null if creation failed

public void **deselectTrack** (int index)

Added in API level 16

Deselect a track.

Currently, the track must be a timed text track and no audio or video tracks can be deselected. If the timed text track identified by index has not been selected before, it throws an exception.

Parameters

index	the index of the track to be deselected. The valid range of the index is 0..total number of tracks - 1. The total number of tracks as well as the type of each individual track can be found by calling `getTrackInfo()` method.

Throws

IllegalStateException	if called in an invalid state.

See Also

`getTrackInfo()`

public int **getAudioSessionId** ()

Added in API level 9

Returns the audio session ID.

Returns

the audio session ID. Note that the audio session ID is 0 only if a problem occured when the MediaPlayer was contructed.

public int **getCurrentPosition** ()

Added in API level 1

Gets the current playback position.

Returns

the current position in milliseconds

public int **getDuration** ()

Added in API level 1

Gets the duration of the file.

Returns

the duration in milliseconds, if no duration is available (for example, if streaming live content), -1 is returned.

public TrackInfo[] **getTrackInfo** ()

Added in API level 16

Returns an array of track information.

Returns

Array of track info. The total number of tracks is the array length. Must be called again if an external timed text source has been added after any of the addTimedTextSource methods are called.

Throws

IllegalStateException	if it is called in an invalid state.

public int **getVideoHeight** ()

Added in API level 1

Returns the height of the video.

Returns

the height of the video, or 0 if there is no video, no display surface was set, or the height has not been determined yet. The OnVideoSizeChangedListener can be registered via `setOnVideoSizeChangedListener(OnVideoSizeChangedListener)` to provide a notification when the height is available.

public int **getVideoWidth** ()

Added in API level 1

Returns the width of the video.

Returns

the width of the video, or 0 if there is no video, no display surface was set, or the width has not been determined yet. The

OnVideoSizeChangedListener can be registered via
`setOnVideoSizeChangedListener(OnVideoSizeChangedListener)`
to provide a notification when the width is available.

public boolean **isLooping** ()

Added in API level 3

Checks whether the MediaPlayer is looping or non-looping.

Returns

true if the MediaPlayer is currently looping, false otherwise

public boolean **isPlaying** ()

Added in API level 1

Checks whether the MediaPlayer is playing.

Returns

true if currently playing, false otherwise

Throws

IllegalStateException	if the internal player engine has not been initialized or has been released.

public void **pause** ()

Added in API level 1

Pauses playback. Call start() to resume.

Throws

IllegalStateException	if the internal player engine has not been initialized.

public void **prepare** ()

Added in API level 1

Prepares the player for playback, synchronously. After setting the datasource and the display surface, you need to either call prepare() or prepareAsync(). For files, it is OK to call prepare(), which blocks until MediaPlayer is ready for playback.

Throws

IllegalStateException	if it is called in an invalid state
IOException	

public void **prepareAsync** ()

Added in API level 1

Prepares the player for playback, asynchronously. After setting the datasource and the display surface, you need to either call prepare() or prepareAsync(). For streams, you should call prepareAsync(), which returns immediately, rather than blocking until enough data

has been buffered.

Throws

IllegalStateException	if it is called in an invalid state

public void **release** ()

Added in API level 1

Releases resources associated with this MediaPlayer object. It is considered good practice to call this method when you're done using the MediaPlayer. In particular, whenever an Activity of an application is paused (its onPause() method is called), or stopped (its onStop() method is called), this method should be invoked to release the MediaPlayer object, unless the application has a special need to keep the object around. In addition to unnecessary resources (such as memory and instances of codecs) being held, failure to call this method immediately if a MediaPlayer object is no longer needed may also lead to continuous battery consumption for mobile devices, and playback failure for other applications if no multiple instances of the same codec are supported on a device. Even if multiple instances of the same codec are supported, some performance degradation may be expected when unnecessary multiple instances are used at the same time.

public void **reset** ()

Added in API level 1

Resets the MediaPlayer to its uninitialized state. After calling this method, you will have to initialize it again by setting the data source and calling prepare().

public void **seekTo** (int msec)

Added in API level 1

Seeks to specified time position.

Parameters

msec	the offset in milliseconds from the start to seek to

Throws

IllegalStateException	if the internal player engine has not been initialized

public void **selectTrack** (int index)

Added in API level 16

Selects a track.

If a MediaPlayer is in invalid state, it throws an IllegalStateException exception. If a MediaPlayer is in *Started* state, the selected track is presented immediately. If a MediaPlayer is not in Started state, it just

marks the track to be played.

In any valid state, if it is called multiple times on the same type of track (ie. Video, Audio, Timed Text), the most recent one will be chosen.

The first audio and video tracks are selected by default if available, even though this method is not called. However, no timed text track will be selected until this function is called.

Currently, only timed text tracks or audio tracks can be selected via this method. In addition, the support for selecting an audio track at runtime is pretty limited in that an audio track can only be selected in the *Prepared* state.

Parameters

index	the index of the track to be selected. The valid range of the index is 0..total number of track - 1. The total number of tracks as well as the type of each individual track can be found by calling `getTrackInfo()` method.

Throws

IllegalStateException	if called in an invalid state.

See Also

`getTrackInfo()`

public void **setAudioSessionId** (int sessionId) Added in API level 9

Sets the audio session ID.

Parameters

sessionId	the audio session ID. The audio session ID is a system wide unique identifier for the audio stream played by this MediaPlayer instance. The primary use of the audio session ID is to associate audio effects to a particular instance of MediaPlayer: if an audio session ID is provided when creating an audio effect, this effect will be applied only to the audio content of media players within the same audio session and not to the output mix. When created, a MediaPlayer instance automatically generates its own audio session ID. However, it is possible to force this player to be part of an already existing audio session by calling this method. This method must be called before one of the overloaded `setDataSource` methods.

Throws

IllegalStateException	if it is called in an invalid state
IllegalArgumentException	

public void **setAudioStreamType** (int streamtype) Added in API level 1

Sets the audio stream type for this MediaPlayer. See `AudioManager` `(/reference/android/media/AudioManager.html)` for a list of stream types. Must call this method before prepare() or prepareAsync() in order for the target stream type to become effective thereafter.

Parameters

streamtype	the audio stream type

See Also

`AudioManager`

public void **setAuxEffectSendLevel** (float level) Added in API level 9

Sets the send level of the player to the attached auxiliary effect . The level value range is 0 to 1.0.

By default the send level is 0, so even if an effect is attached to the player this method must be called for the effect to be applied.

Note that the passed level value is a raw scalar. UI controls should be scaled logarithmically: the gain applied by audio framework ranges from -72dB to 0dB, so an appropriate conversion from linear UI input x to level is: x == 0 -> level = 0 0 < x <= R -> level = 10^(72*(x-R)/20/R)

Parameters

level	send level scalar

See Also

`attachAuxEffect(int)`

public void **setDataSource** (String path) Added in API level 1

Sets the data source (file-path or http/rtsp URL) to use.

Parameters

path	the path of the file, or the http/rtsp URL of the stream you want to play

Throws

IllegalStateException	if it is called in an invalid state

When `path` refers to a local file, the file may actually be opened by a process other than the calling application. This implies that the pathname should be an absolute path (as any other process runs with unspecified current working directory), and that the pathname should reference a world-readable file. As an alternative, the application could first open the file for reading, and then use the file descriptor form `setDataSource(FileDescriptor)` (/reference/android/media/MediaPlayer.html#setDataSource(java.io.FileDescriptor)).

IOException	
IllegalArgumentException	
SecurityException	

public void **setDataSource** (Context context, Uri uri, Map<String, String> headers) Added in API level 14

Sets the data source as a content Uri.

Parameters

context	the Context to use when resolving the Uri
uri	the Content URI of the data you want to play
headers	the headers to be sent together with the request for the data

Throws

IllegalStateException	if it is called in an invalid state
IOException	
IllegalArgumentException	
SecurityException	

public void **setDataSource** (Context context, Uri uri) Added in API level 1

Sets the data source as a content Uri.

Parameters

context	the Context to use when resolving the Uri
uri	the Content URI of the data you want to play

Throws

IllegalStateException	if it is called in an invalid state

IOException

IllegalArgumentException

SecurityException

public void **setDataSource** (FileDescriptor fd, long offset, long length)

Added in API level 1

Sets the data source (FileDescriptor) to use. The FileDescriptor must be seekable (N.B. a LocalSocket is not seekable). It is the caller's responsibility to close the file descriptor. It is safe to do so as soon as this call returns.

Parameters

fd	the FileDescriptor for the file you want to play
offset	the offset into the file where the data to be played starts, in bytes
length	the length in bytes of the data to be played

Throws

IllegalStateException	if it is called in an invalid state
IOException	
IllegalArgumentException	

public void **setDataSource** (FileDescriptor fd)

Added in API level 1

Sets the data source (FileDescriptor) to use. It is the caller's responsibility to close the file descriptor. It is safe to do so as soon as this call returns.

Parameters

fd	the FileDescriptor for the file you want to play

Throws

IllegalStateException	if it is called in an invalid state
IOException	
IllegalArgumentException	

public void **setDisplay** (SurfaceHolder sh)

Added in API level 1

Sets the `SurfaceHolder (/reference/android/view/SurfaceHolder.html)` to use for displaying the video portion of the media. Either a surface holder or surface must be set if a display or video sink is needed. Not calling this method or `setSurface(Surface) (/reference/android/media`

/MediaPlayer.html#setSurface(android.view.Surface)) when playing back a video will result in only the audio track being played. A null surface holder or surface will result in only the audio track being played.

Parameters

sh the SurfaceHolder to use for video display

public void **setLooping** (boolean looping) Added in API level 1

Sets the player to be looping or non-looping.

Parameters

looping whether to loop or not

public void **setNextMediaPlayer** (MediaPlayer next) Added in API level 16

Set the MediaPlayer to start when this MediaPlayer finishes playback (i.e. reaches the end of the stream). The media framework will attempt to transition from this player to the next as seamlessly as possible. The next player can be set at any time before completion. The next player must be prepared by the app, and the application should not call start() on it. The next MediaPlayer must be different from 'this'. An exception will be thrown if next == this. The application may call setNextMediaPlayer(null) to indicate no next player should be started at the end of playback. If the current player is looping, it will keep looping and the next player will not be started.

Parameters

next the player to start after this one completes playback.

public void **setOnBufferingUpdateListener** (MediaPlayer.OnBufferingUpdateListener listener) Added in API level 1

Register a callback to be invoked when the status of a network stream's buffer has changed.

Parameters

listener the callback that will be run.

public void **setOnCompletionListener** (MediaPlayer.OnCompletionListener listener) Added in API level 1

Register a callback to be invoked when the end of a media source has been reached during playback.

Parameters

listener the callback that will be run

public void **setOnErrorListener**
(MediaPlayer.OnErrorListener listener) Added in API level 1

Register a callback to be invoked when an error has happened during an asynchronous operation.

Parameters

listener the callback that will be run

public void **setOnInfoListener**
(MediaPlayer.OnInfoListener listener) Added in API level 3

Register a callback to be invoked when an info/warning is available.

Parameters

listener the callback that will be run

public void **setOnPreparedListener**
(MediaPlayer.OnPreparedListener listener) Added in API level 1

Register a callback to be invoked when the media source is ready for playback.

Parameters

listener the callback that will be run

public void **setOnSeekCompleteListener**
(MediaPlayer.OnSeekCompleteListener listener) Added in API level 1

Register a callback to be invoked when a seek operation has been completed.

Parameters

listener the callback that will be run

public void **setOnTimedTextListener**
(MediaPlayer.OnTimedTextListener listener) Added in API level 16

Register a callback to be invoked when a timed text is available for display.

Parameters

listener the callback that will be run

public void **setOnVideoSizeChangedListener** (MediaPlayer.OnVideoSizeChangedListener listener) Added in API level 3

Register a callback to be invoked when the video size is known or updated.

Parameters

listener the callback that will be run

public void **setScreenOnWhilePlaying** (boolean screenOn) Added in API level 1

Control whether we should use the attached SurfaceHolder to keep the screen on while video playback is occurring. This is the preferred method over `setWakeMode(Context, int)` (/reference/android/media/MediaPlayer.html#setWakeMode(android.content.Context, int)) where possible, since it doesn't require that the application have permission for low-level wake lock access.

Parameters

screenOn Supply true to keep the screen on, false to allow it to turn off.

public void **setSurface** (Surface surface) Added in API level 14

Sets the `Surface` (/reference/android/view/Surface.html) to be used as the sink for the video portion of the media. This is similar to `setDisplay(SurfaceHolder)` (/reference/android/media/MediaPlayer.html#setDisplay(android.view.SurfaceHolder)), but does not support `setScreenOnWhilePlaying(boolean)` (/reference/android/media/MediaPlayer.html#setScreenOnWhilePlaying(boolean)). Setting a Surface will un-set any Surface or SurfaceHolder that was previously set. A null surface will result in only the audio track being played. If the Surface sends frames to a `SurfaceTexture` (/reference/android/graphics/SurfaceTexture.html), the timestamps returned from `getTimestamp()` (/reference/android/graphics/SurfaceTexture.html#getTimestamp()) will have an unspecified zero point. These timestamps cannot be directly compared between different media sources, different instances of the same media source, or multiple runs of the same program. The timestamp is normally monotonically increasing and is unaffected by time-of-day adjustments, but it is reset when the position is set.

Parameters

surface	The `Surface` to be used for the video portion of the media.

public void **setVideoScalingMode** (int mode)

Added in API level 16

Sets video scaling mode. To make the target video scaling mode effective during playback, this method must be called after data source is set. If not called, the default video scaling mode is `VIDEO_SCALING_MODE_SCALE_TO_FIT` (/reference/android/media/MediaPlayer.html#VIDEO_SCALING_MODE_SCALE_TO_FIT).

The supported video scaling modes are:

- `VIDEO_SCALING_MODE_SCALE_TO_FIT`
- `VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING`

Parameters

mode	target video scaling mode. Most be one of the supported video scaling modes; otherwise, IllegalArgumentException will be thrown.

See Also

`VIDEO_SCALING_MODE_SCALE_TO_FIT`
`VIDEO_SCALING_MODE_SCALE_TO_FIT_WITH_CROPPING`

public void **setVolume** (float leftVolume, float rightVolume)

Added in API level 1

Sets the volume on this player. This API is recommended for balancing the output of audio streams within an application. Unless you are writing an application to control user settings, this API should be used in preference to `setStreamVolume(int, int, int)` (/reference/android/media/AudioManager.html#setStreamVolume(int, int, int)) which sets the volume of ALL streams of a particular type. Note that the passed volume values are raw scalars in range 0.0 to 1.0. UI controls should be scaled logarithmically.

Parameters

leftVolume	left volume scalar
rightVolume	right volume scalar

public void **setWakeMode** (Context context, int mode)

Added in API level 1

Set the low-level power management behavior for this MediaPlayer.

This can be used when the MediaPlayer is not playing through a SurfaceHolder set with setDisplay(SurfaceHolder) (/reference/android/media/MediaPlayer.html#setDisplay(android.view.SurfaceHolder)) and thus can use the high-level setScreenOnWhilePlaying(boolean) (/reference/android/media/MediaPlayer.html#setScreenOnWhilePlaying(boolean)) feature.

This function has the MediaPlayer access the low-level power manager service to control the device's power usage while playing is occurring. The parameter is a combination of PowerManager (/reference/android/os/PowerManager.html) wake flags. Use of this method requires WAKE_LOCK (/reference/android/Manifest.permission.html#WAKE_LOCK) permission. By default, no attempt is made to keep the device awake during playback.

Parameters

context	the Context to use
mode	the power/wake mode to set

See Also

PowerManager

public void **start** ()

Added in API level 1

Starts or resumes playback. If playback had previously been paused, playback will continue from where it was paused. If playback had been stopped, or never started before, playback will start at the beginning.

Throws

IllegalStateException	if it is called in an invalid state

public void **stop** ()

Added in API level 1

Stops playback after playback has been stopped or paused.

Throws

IllegalStateException	if the internal player engine has not been initialized.

Protected Methods

protected void **finalize** ()

Added in API level 1

Invoked when the garbage collector has detected that this instance is no longer reachable. The default implementation does nothing, but this method can be overridden to free resources.

Note that objects that override `finalize` are significantly more expensive than objects that don't. Finalizers may be run a long time after the object is no longer reachable, depending on memory pressure, so it's a bad idea to rely on them for cleanup. Note also that finalizers are run on a single VM-wide finalizer thread, so doing blocking work in a finalizer is a bad idea. A finalizer is usually only necessary for a class that has a native peer and needs to call a native method to destroy that peer. Even then, it's better to provide an explicit `close` method (and implement `Closeable (/reference/java/io/Closeable.html)`), and insist that callers manually dispose of instances. This works well for something like files, but less well for something like a `BigInteger` where typical calling code would have to deal with lots of temporaries. Unfortunately, code that creates lots of temporaries is the worst kind of code from the point of view of the single finalizer thread.

If you *must* use finalizers, consider at least providing your own `ReferenceQueue (/reference/java/lang/ref/ReferenceQueue.html)` and having your own thread process that queue.

Unlike constructors, finalizers are not automatically chained. You are responsible for calling `super.finalize()` yourself.

Uncaught exceptions thrown by finalizers are ignored and do not terminate the finalizer thread. See *Effective Java* Item 7, "Avoid finalizers" for more.