

Sun Certified Mobile Application Developer (311-110)
Sun Microsystems, Inc.

Exam Notes
Sathya Srinivasan

02/14/2004

Table of Contents

MIDP 2.0 media support	5
DataSource.....	6
Manager	6
Player	7
UNREALIZED state.....	7
REALIZED state	7
PREFETCHED state	8
STARTED state.....	8
CLOSED state	8
Audio and Video Capture.....	8
System Properties	9
Class Hierarchy of media support in MIDP 2.0 and MMAPI 1.1	10

Chapter 10 Media using MIDP 2.1 and MMAPI 1.1

Given a set of requirements, develop code using MMAPI's support for Tone generation.

- Simple monotonic tones can be produced using the `Manager.playTone(int note, int duration, int volume)` method.
 - `note` can vary from 0 to 127
 - `duration` is expressed in milliseconds
 - `volume` can vary from 0 to 100 (percentage of the actual volume)
 - More complex tones can be played by getting a Tone sequence player.
 - `Player p = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR)`
 - A Tone Sequence player MUST implement `ToneControl` and SHOULD implement `VolumeControl` and `StopTimeControl` interfaces.
 - A monotonic tone sequence is a byte array and has the following structure.
 - Version definition (`ToneControl.VERSION, 1`)
 - Tempo definition (`ToneControl.TEMPO, {5-127}`)
 - Resolution definition (`ToneControl.RESOLUTION, {1-127}`)
 - Block definition (`ToneControl.BLOCK_START, {block number}, {tone values}, BLOCK_END, {same block number}`)
 - Sequence Events should be one of the following.
 - § Tone Event - A single tone of the form `{note value, tone duration}`.
 - § Volume Event - Indicates volume to be set from this point forward. Has two consecutive bytes in the form `{ToneControl.SET_VOLUME, volume value}`
 - § Repeat Event - Repeats a given tone. `{ToneControl.REPEAT, multiplier, note}`
 - `multiplier` should be between 2 and 127
 - § Block Event - Plays a pre-defined block. `{ToneControl.PLAY_BLOCK, block number}`
 - The byte sequence can be set using the `ToneControl.setSequence(byte[] sequence)` method.
 - The method can be called only if the `Player` is NOT in the `PREFETCHED` or `STARTED` state. Otherwise, an `IllegalStateException` will be thrown.
 - The constant `ToneControl.C4` defines the 'middle C' and has the value 60.
 - The constant `ToneControl.SILENCE` can be used to create a pause.
 - Multiplying the tempo by 4 gives the bpm (beats per minute).
-
- `VolumeControl` can be used to set the volume of a `Player`.
 - The volume level can be between 0 and 100, inclusive. If the set value is less than 0, it will be treated as 0 and if the set value is greater than 100, it will be treated as 100.
 - When a `Player` is unmuted, the volume will return to what was there previously before the `Player` was muted.
 - `StopTimeControl` can be used to stop a `Player` after a specific time. This is something like setting the sleep timer in a TV or stereo.
 - The `Player` is guaranteed to stop at most before 1 second after the specified time.
 - If the player is running while the stop time is being set and the player has already crossed the requested time, it will be stopped immediately.

- Once the `Player` is stopped, the stop time is reset. It can also be reset manually using the `RESET` constant.
- If the *media stop time* has been set on the `Player`, then calling the `setStopTime(int time)` will cause an `IllegalStateException` to be thrown.

Given a set of requirements, develop code that correctly uses MIDP support for sound including audio playback, tone generation, media flow controls (start, stop, etc.), media type controls (volume, tone), and media capabilities using "Manager", "Player", and "Control" objects, recognizing the difference between required and optional features.

MIDP 2.0 media support

- Since MIDP 2.0 might be supported by devices that might not have the need to provide extensive media support, the media requirements in MIDP 2.0 is a strict subset of the MMAPI 1.1 specification. Hence, whatever is under MIDP 2.0 is totally compatible with MMAPI 1.1.
- MIDP 2.0 media supports audio only. All graphic and video controls are excluded.
- Custom protocols via **DataSource** is not supported.
- MIDI playback is not mandated and so the `Player.MIDI_DEVICE_LOCATOR` is not present.

Class	Function	Difference w.r. to MMAPI 1.1
<code>Manager</code>	Factory to create <code>Players</code> .	No support for <code>DataSource</code> .
<code>Player</code>	Plays media data.	Subset of features. No MIDI support.
<code>Control</code>	Interface for media controls.	
<code>ToneControl</code>	Controls Tone Sequence Format	
<code>VolumeControl</code>	Controls the volume	
<code>Controllable</code>	Super-interface for <code>Player</code> .	No <code>SourceStream</code> sub-interface.
<code>PlayerListener</code>	Listener for <code>Player</code> events.	Subset of features.

- Operations and functions are exactly the same as that of MMAPI 1.1. See other sections for details.

Develop code that correctly uses MMAPI support for playback and recording of media, including the use of the "DataSource", "Player", and "Manager" objects, support for audio and video capture and playback, system properties queries, recognizing the difference between required and optional features.

DataSource

- `DataSource` is an abstract class that can be extended to create custom streaming content.
- The `MediaPlayer.createPlayer(DataSource ds)` method can be used to create a `Player` object from a specified `DataSource`.
- `DataSource` extensions are preferable over the standard `InputStream` objects since these support random seeking capabilities. That is, you can go to a specified time in the content using this class, which is not possible if you use `InputStream`.
- `DataSource` also supports the concept of a transfer size, which is more suitable for framed content like video.
- The typical sequence of operations in a `DataSource` object is
 - `DataSource(String locator)` to create an object for a given locator URL.
 - `DataSource.connect()` to connect to the source.
 - `DataSource.getStreams():SourceStream[]` to get one or more streams associated with this `DataSource`. Typically there will be only one stream.
 - `DataSource.start()` to start the transfer of data.
 - `DataSource.stop()` to stop the transfer of data.
 - `DataSource.disconnect()` to disconnect from the source.
- The `SourceStream.getTransferSize()` can be used to find the size of a block of data transferred. This information can be used to set appropriate buffer size while calling the `SourceStream.read(byte[] byte, int offset, int length)` method.

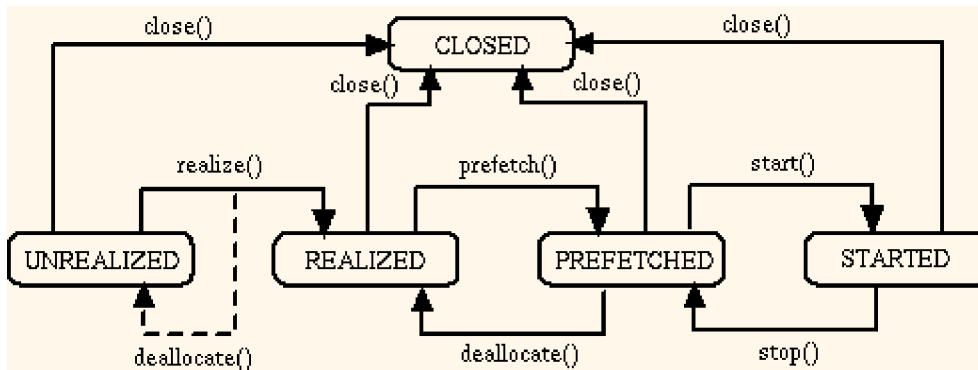
Manager

- The `Manager` is a factory class that creates `Players` based on the content type of the associated stream.
- The `Manager.createPlayer()` method can take in a locator URL (a `String`), a `DataSource` object, or an `InputStream` object which will have information about the source.
- The `Manager` also has a simple method `Manager.playTone()` to play a tone.
- Convenience constants `MIDI_DEVICE_LOCATOR` and `TONE_DEVICE_LOCATOR` can be used to get a MIDI (audio/midi) or Tone Sequence (audio/x-tone-seq) `Player`.
- `Manager.getSupportedContentTypes(null)` will give ALL the content types supported by this `Manager`. If a valid `String` is passed, it gives the content types supported for the requested protocol.
- `Manager.getSupportedProtocols(null)` will give ALL the protocols supported by this `Manager`. If a valid `String` is passed, it gives the protocols supported for the requested content type.
- The locator URL is of the format `<scheme>://<scheme-specific-part>`.
- The `Manager.getSystemTimeBase()` gets a implementation-independent time base that can be used by the `Players`.

Locator URL	Content Type
device://tone	Tone Sequence (TONE_DEVICE_LOCATOR)
device://midi	MIDI (MIDI_DEVICE_LOCATOR)
rtp://host[:port][/type]	RTP (type can be audio, video, or text)

Player

- Renders a time-based media data (audio, video, etc.)
- Lifecycle of a `Player` can be understood using the following diagram.



- The `Player.setMediaTime()` can be called to randomly place the pointer to a specific time. This is similar to, say, fast forwarding/reversing the data to the x^{th} minute of play.
- Loop count can be between 1 and 127. -1 will cause indefinite looping. 0 is not allowed.

UNREALIZED state

- The `Player` does not have information about the data.
- The following methods CANNOT be invoked (will cause `IllegalStateException`)
 - `getContentType()`
 - `set/getTimeBase()`
 - `setMediaTime()`
 - `getControls()`
 - `getControl()`
- The `Player.realize()` method is used to realize the `Player`.
 - This is a time-consuming method since it takes time to get information about the data source.
 - While the `Player` is being realized, the `deallocate()` method can be called to stop the realization and move the `Player` back to the UNREALIZED state if needed.

REALIZED state

- The `Player` has enough information about the data that it can play.
- Once the `Player` is realized, it NEVER returns to the UNREALIZED state.
- The `Player.prefetch()` method can be used to move the player to the PREFETCHED state.
 - This is a time-consuming method since the `Player` might try to get as much information or data as possible to minimize the start time of the `Player`.
 - If this method before realizing the `Player`, the `Player.realize()` method is implicitly called.

PREFETCHED state

- The `Player` has all the information needed and is ready to be started.
- When a `Player` is stopped, it returns to this state.
- The `Player` can be started using the `Player.start()` method.
 - If this method is called before the `Player` is in the `PREFETCHED` state, then the `Player.prefetch()` method is implicitly called.
 - If the `Player` was stopped previously by calling the `Player.stop()` method, then the `Player` will start playing from where it left off. In essence, this is equivalent of pressing the PAUSE button in a stereo.

STARTED state

- The `Player` has started playing. This method is a non-blocking method and returns as soon as the `Player` has started playing. The playing itself happens in the background.
- The following methods will throw an `IllegalStateException` if invoked.
 - `setTimeBase()`
 - `setLoopCount()`
- If a loop count is set, then once the end of the data has reached, the `Player` will loop back to the beginning and start all over once again till there are no more loops.
- If the `Player.stop()` method is called, the `Player` returns to the `PREFETCHED` state.

CLOSED state

- This state can be reached by calling the `Player.close()` method when the `Player` is in the `REALIZED`, `PREFETCHED`, or `STARTED` state.

Audio and Video Capture

- Audio and Video capture can be done using a `Player` that implements the `RecordControl` interface.
- The capture can be done by following these steps.
 - Realize the `Player` using the `Player.realize()` method.
 - Get the `RecordControl` interface of the `Player` using the `Player.getControl("RecordControl")` method.
 - Realize the `Player`.
 - Set the `RecordControl`'s stream using the `RecordControl.setRecordStream()` or `RecordControl.setRecordLocation()` method.
 - Start the `RecordControl` using the `RecordControl.start()` method.
 - Start the `Player` using the `Player.startRecord()` method.
 - Stop the `Player` using the `Player.stop()` method.
 - Stop the `RecordControl` using the `RecordControl.stopRecord()` method.
 - Commit the data captured using the `RecordControl.commit()` method.
- If the `Player` is not running and the `RecordControl.startRecord()` is called, it will be on stand-by till the player has actually started running. So there will be no "empty-recording".
- The `RecordControl.reset()` method can be used to erase the captured content.

System Properties

Property	Description
<code>supports.mixing</code>	true if mixing is supported. false otherwise. Mixing means that at least two tones can be played simultaneously using any combination of the <code>Manager.playTone()</code> and the audio playback methods.
<code>supports.audio.capture</code>	true if audio capture is supported. false otherwise.
<code>supports.video.capture</code>	true if video capture is supported. false otherwise.
<code>supports.recording</code>	true if recording is supported. false otherwise.
<code>audio.encodings</code>	Space separated supported audio encodings.
<code>video.encodings</code>	Space separated supported video encodings.
<code>video.snapshot.encodings</code>	Space separated supported image encodings for video capture.

Identify correct and incorrect statements or examples about the media class hierarchies in both MIDP 2.0 and MMAPI 1.1.

Class Hierarchy of media support in MIDP 2.0 and MMAPI 1.1

- `Controllable` Interface is the base interface.
 - `Player` implements the interface and plays media data.
 - `SourceStream` implements the interface and provides custom protocol support. Not in MIDP 2.0.
 - `DataSource` is a class used to provide custom media protocols. Not in MIDP 2.0.
- `Manager` class is a factory to get `Player` implementations.
 - Default constants provided for Tone Sequence (`TONE_DEVICE_LOCATOR`) and MIDI playback (`MIDI_DEVICE_LOCATOR`). MIDI is not supported in MIDP 2.0.
- `Player` class is used to play media data. An implementation is obtained from the `Manager` factory.
 - `TimeBase` is used by the `Player` to track time.
- `Control` interface is the super-interface for all controls that can be implemented by a `Player`. Only the identified controls are supported in MIDP 2.0.
 - `Audio Controls`
 - § `ToneControl` controls the Tone Sequence format. Supported in MIDP 2.0.
 - § `VolumeControl` controls the `Player`'s volume. Supported in MIDP 2.0.
 - § `MIDIControl` controls MIDI playback.
 - § `PitchControl` controls the pitch of a MIDI note.
 - § `TempoControl` controls the tempo of a MIDI note.
 - `Video Controls`
 - § `VideoControl` controls video media playback.
 - § `FramePositioningControl` controls the framing of video.
 - `General Controls`
 - § `StopTimeControl` allows you to set a preset stop time. This is like a sleep-time.
 - § `RateControl` is used in conjunction with `TimeBase` to calibrate time used by `Player`.
 - § `GUIControl` is used to display a user interface for the associated `Player`.
 - § `RecordControl` is used to record the media played by the `Player`.
 - § `MetaDataControl` can be used to get some general meta-data information about the media content.
- `PlayerListener` is a listener interface that can be implemented and registered to a `Player` to listen to the `Player`'s events.