

# Working with the Mobile Media API (JSR 135) - Part 1

**Author:** Martin de Jode  
**Status:** Version 1.2  
**Date:** 12/6/03

## 1. Introduction

This paper is the first of two papers covering the features offered by the Mobile Media API. In this paper we will provide an overview of the Mobile Media API and its architecture. We will then illustrate some of concepts introduced, as well as exploring in more detail the support offered for video playback, by means of a detailed study of a simple video player `MIDlet`. The second paper will discuss further features of the API including audio playback, tone generation and photo capture.

## 2. Mobile Media API

The Mobile Media API (JSR 135) is an optional API targeted at J2ME CLDC based devices that defines a framework to support playback and recording of audio and video, plus photo capture from an onboard camera, as well as tone generation. Although the MMAPI is targeted at CLDC based devices, other environments (e.g. CDC) are not excluded, but CLDC is taken as the lowest common denominator.

The aim of JSR 135 was to provide support for a wide range of features whilst acknowledging that highly constrained devices, may not implement all the functionality defined by the specification. Since MMAPI is an optional package, it does not mandate support for any particular media types or protocols, instead these are determined by the implementers of the API subject to the constraints imposed by the hardware.

Although the Mobile Media API doesn't mandate support for any specific media type it does require that an implementation supporting a given media type must implement certain features in the form of controls. Hence an implementation supporting video must provide a `VideoControl` and an implementation supporting tone generation must support a `ToneControl`. The specification also specifies what features an implementation should support (ie recommended practice). For instance an implementation supporting audio media types should provide an `AudioControl` and `StopTimeControl`; video media support should include `FramePositioningControl`, `StopTimeControl`, and a `VolumeControl` (if audio is also available). Other controls are entirely optional, such as `RecordControl` and `RateControl`. For a full breakdown of these requirements check out the [MMAPI specification](#).

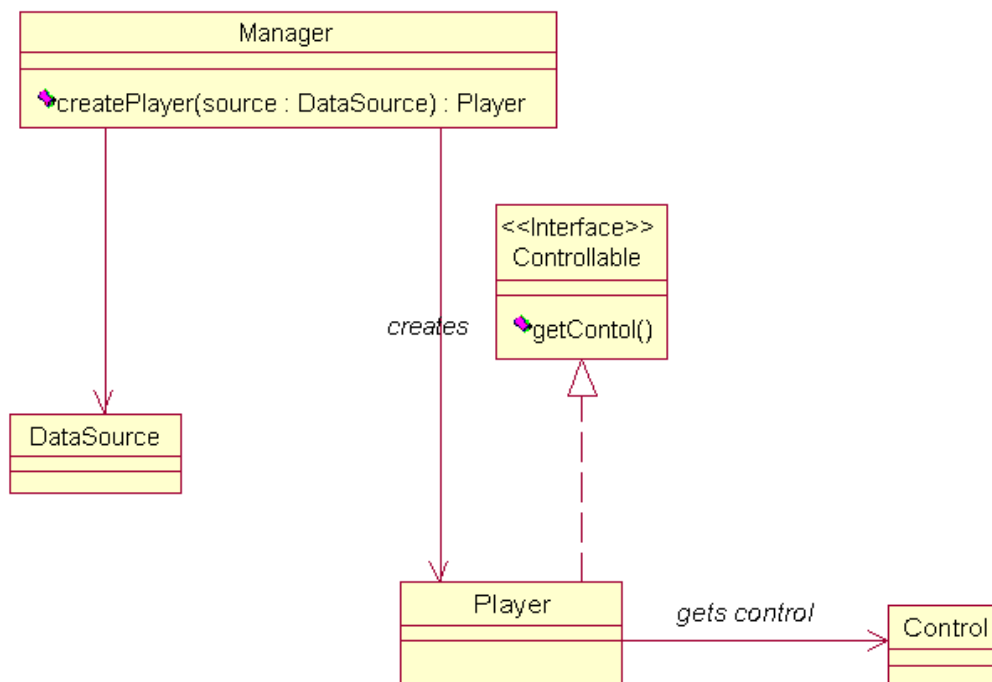
It is also worth mentioning that the MIDP 2.0 specification (JSR 118) supports an audio only subset of the Mobile Media API including support for

- Audio playback
- Volume control
- Tone generation.

The current version of Symbian OS, Version 7.0s includes MIDP 2.0 and therefore the audio subset of the Mobile Media API. Furthermore the Symbian OS based Nokia 3650 camera phone provides extensive support for the Mobile Media API with an implementation by Nokia, who were the specification lead on JSR 135.

### 3. Architecture of the Mobile Media API

The basic architecture of the Mobile Media API is shown below



**Figure 1 The Architecture of the Mobile Media API**

Players are created using the `createPlayer(...)` factory method of the `javax.microedition.media.Manager` class. The `createPlayer(...)` method has the following signatures

```

Public static Player createPlayer(String locator)
Public static Player createPlayer(InputStream stream, String type)
Public static Player createPlayer(DataSource source)
  
```

The first `createPlayer(...)` method takes a `String` media locator as its argument. The syntax of the locator takes several forms, examples of which are shown below.

```

http://www.myserver.com/myvideo.mpeg
capture://audio
capture://video
capture://video?encoding=gray8&width=160&height=120
  
```

The syntax of the capture locator is specified in Augmented Backus-Naur Format (BNF) and described in more detail in the Mobile Media API [specification](#).

The second variant of the `createPlayer(...)` method takes an `InputStream` and MIME type as argument and would be used, for example, to read a byte stream from the RMS store

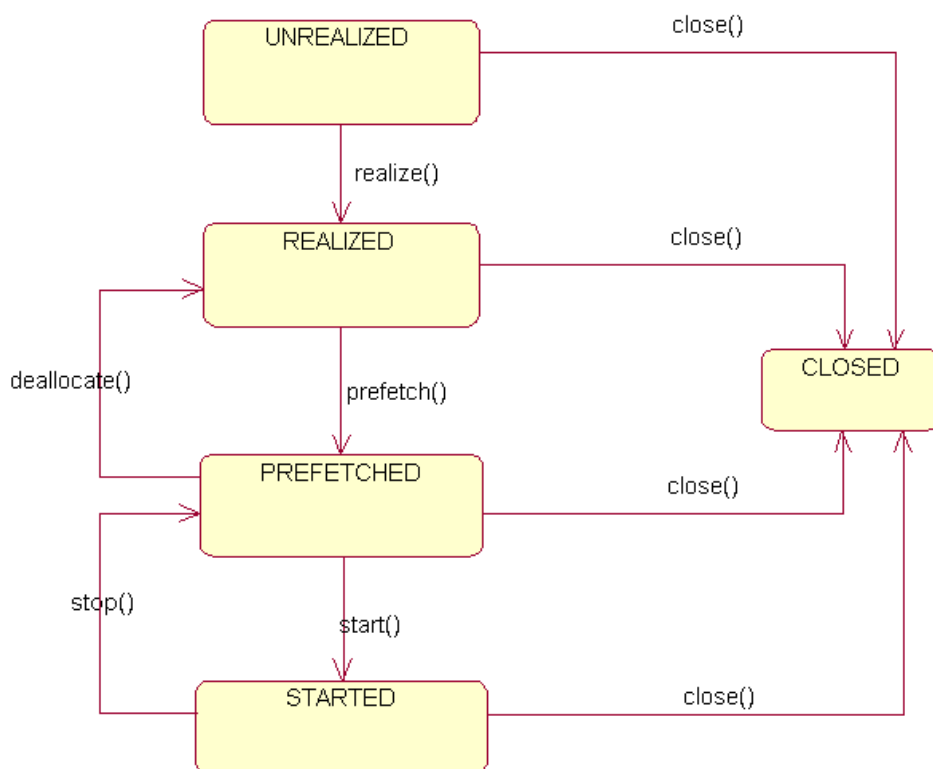
```
Player p = Manager.createPlayer(inputStream, "video/mpeg");
```

The final method takes a custom data source deriving from the abstract `DataSource` class and would be used for handling proprietary or application-defined protocols.

```
Player p = Manager.createPlayer(customDataSource);
```

Note that all variants of `createPlayer(...)` throw a `MediaException` if it is not possible to create a player of the correct type (perhaps due to that media type being unsupported). They also throw an `IOException` if there is a problem connecting to or reading from the source or stream.

A `Player` is a high level construct that controls the rendering of time based media data. The life cycle of a `Player` is shown in the diagram below.




---

**Figure 2 Lifecycle of a Player**

---

When a player is created it is in the `UNREALIZED` state. Calling the `realize()` method causes the `Player` to perform the communication necessary to locate all of the resources it needs to function (such as communicating with a server or a file system). When this method returns the player is in the `REALIZED` state. Calling the `prefetch()` method causes the `Player` to acquire the scarce and exclusive resources it needs to play the media, such as access to the phone's audio device. It may have to wait for another application to release these resources, before it can move to the `PREFETCHED` state. Once in this state the player is ready to start. In order to interrogate the state of the `Player` the `getState()` method of the `Player` class is provided.

Before media content can be played it is necessary to obtain one or more `Control` objects. `Control` objects are used to control media processing and are obtained from a `Controllable`, in this case the `Player`, using the `getControl(String controlType)` method. As described in the previous section a media player of a given type may support a variety of `Controls`. The `getControls()` method can be used to determine the available controls. Note the `getControl(...)` and `getControls()` methods must be invoked on a player in the `REALIZED` state otherwise an `IllegalStateException` will be thrown.

The final topic for this brief introduction to the Mobile Media API is the `PlayerListener` interface. The `PlayerListener` interface provides a `playerUpdate(...)` method for receiving asynchronous events from `Players`. A class may implement this interface and then register the `PlayerListener` using the `addPlayerListener(...)` method of the `Player` class. The `PlayerListener` listens for a range of standard pre-defined events including `STARTED`, `STOPPED`, `END_OF_MEDIA`. For a listing of all the standard events refer to the MMAPI [documentation](#).

## 4. Video Playback - a simple video player

We shall now illustrate some of the concepts introduced in the previous section with code highlights taken from a simple video player `MIDlet`. The `MIDlet` contains two classes: `VideoPlayer`, which extends `MIDlet` and has a `Form` attribute used to display a `TextField` and a `Gauge`, plus some controls ("Play", "Exit"); and a `VideoCanvas` class for rendering the video. Screen shots of the `MIDlet` running on a Nokia 3650 are shown below.



Figure 3 Screen shots of a simple video playing MIDlet

We shall concentrate on the `VideoCanvas` class as this contains the code calling into the Mobile Media API. The key signatures of the `VideoCanvas` class are shown below.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;
import java.io.*;
public class VideoCanvas extends Canvas implements CommandListener, PlayerListener, Runnable
    public VideoCanvas(VideoPlayer parent){}
    public void initializeVideo(String url){ }
    public void run(){ }
    public void playVideo(){ }
    public void paint(Graphics g) { }
    public void playerUpdate(Player p, String event, Object eventData){ }
    public void commandAction(Command c, Displayable s){ }
}
```

The `VideoCanvas` class naturally extends `Canvas`, but also implements the `PlayerListener` interface as well as the `Runnable` and `CommandListener` interface.

The key methods will now be described in more detail. The constructor is shown below and does some standard initialisation.

```
public VideoCanvas(VideoPlayer parent){
    super();
    this.parent = parent;
    display = Display.getDisplay(parent);
    close = new Command("close", Command.SCREEN, 1);
    addCommand(close);
    setCommandListener(this);
}
```

The `initializeVideo(...)` method takes the URL of the video payload as an argument and then launches a new `Thread` to perform the essential initialisation required to play video content.

```
public void initializeVideo(String url){
    this.url = url;
    initializer = new Thread(this);
    initializer.start();
}
```

The `run()` method, mandated by the `Runnable` interface, contains the initialisation of the `Player`. The rationale behind putting this initialisation in a separate thread is that in many cases (although not this particularly simple example) it may be desirable to perform the potentially time consuming initialisation in the background, so that when rendering of the video is required it starts with minimum latency. Another reason for performing the initialisation in a separate thread is so that it is possible to update a progress gauge, giving the user valuable feedback as to how the video acquisition is proceeding. Since `VideoCanvas` implements the `PlayerListener` interface we register this instance with the `Player` to receive calls backs. The `prefetch()` and `realize()` methods are called on the player and a gauge updated at the return of each method.

```
public void run(){
    try {
        player = Manager.createPlayer(url);
        parent.updateGauge();
        player.addPlayerListener(this);
    }
```

```

        player.realize();
        parent.updateGauge();
        player.prefetch();
        parent.updateGauge();
    } catch (IOException ioe) {
        //handle
    } catch (MediaException me) {
        //handle
    }
    }
    playVideo();
}

```

Once the player is in the `PREFETCHED` state we are ready to render the video content. In this example our `playVideo()` method is called immediately.

```

public void playVideo(){
    try {
        videoControl = (VideoControl)player.getControl("videoControl");
        if (videoControl != null) {
            videoControl.initDisplayMode(videoControl.USE_DIRECT_VIDEO, this);
        }

        parent.updateGauge();

        int cHeight = this.getHeight();
        int cWidth = this.getWidth();
        videoControl.setDisplaySize(cWidth, cHeight);
        display.setCurrent(this);
        videoControl.setVisible(true);
        player.start();
    } catch (MediaException me) {
        //handle
    }
}

```

The `playVideo()` method handles rendering the video onto the `Canvas`. To do this we must obtain a `VideoControl`, by calling `getControl(...)` on a realized `Player`, and cast down appropriately.

The `initDisplayMode(...)` method is used to initialise the video mode that determines how the video is displayed. This method take an integer mode value as its first argument with two predefined values `USE_GUI_PRIMITIVE` or `USE_DIRECT_VIDEO`. In the case of the MIDP implementations (supporting the LCDUI) `USE_GUI_PRIMITIVE` will result in an instance of a `javax.microedition.lcdui.Item` being returned. For example

```
Item display = control.initDisplayMode(control.USE_GUI_PRIMITIVE, null);
```

For CDC implementations supporting AWT then `USE_GUI_PRIMITIVE` will return an instance of `java.awt.Component`. For implementations that support both LCDUI and AWT, the required type must be specified by a `String` as the second argument. For example

```
Item display =
control.initDisplayMode(control.USE_GUI_PRIMITIVE, "javax.microedition.lcdui.Item");
```

The `USE_DIRECT_VIDEO` mode can only be used with implementations that support the LCDUI and a second argument of type `javax.microedition.lcdui.Canvas` or a subclass, must be supplied. This is the approach adopted in the example code above. Methods of `VideoControl` can be used to manipulate the size and the location of the video with respect to the `Canvas` where it will be displayed. Since we are using direct video as the display mode it is necessary to call `setVisible(true)` in order for the video to be displayed (in the case of `USE_GUI_PRIMITIVE` the video is shown by default when the GUI primitive is displayed). Finally we start the rendering of the video with the `start()` method.

Since our class is an instance of `Canvas` we must implement a `paint()` method as shown below

```

public void paint(Graphics g){
    g.setColor(128, 128, 128);
    g.fillRect(0, 0, getWidth(), getHeight());
}

```

```
}
```

Here our implementation simply fills the canvas with a suitable background color. The video content is then rendered directly onto the `Canvas` by the `VideoControl`.

Since our `VideoCanvas` class implements the `PlayerListener` interface we must provide a `playerUpdate(...)` method

```
public void playerUpdate(Player p, String event, Object eventData) {
    if (event == PlayerListener.END_OF_MEDIA) {
        if (rePlay == null) {
            rePlay = new Command("re-play", Command.SCREEN, 1);
            addCommand(rePlay);
        }
    }
}
```

In the code shown above we simply listen for the `END_OF_MEDIA` event and add a replay option to our commands when the trailer has finished playing.

Finally we shall have a look at the `commandAction(...)` method mandated by `CommandListener`

```
public void commandAction(Command c, Displayable s) {
    if(c == rePlay){
        try{
            player.start();
        } catch (MediaException me) {
            //handle
        }
    }
    else if(c == close){
        player.close();
        parent.form.delete(1);
        display.setCurrent(parent.form);
        url=null;
        parent=null;
    }
}
```

Selecting the “Replay” option simply calls `player.start()`; the `Player` is already initialised and the display set up, so this is all that is required. Selecting the “Close” option closes down the video player, releasing all acquired resources, and returns the user to the previous screen.

## 5. Summary

In this first paper of a series of two, we have introduced the Mobile Media API and presented an overview of the architecture of the API. We then proceeded to elucidate these concepts by providing a detailed study of the support provided by the API for video playback by means of code highlights from a simple video player MIDlet. The code for `VideoPlayer` will be made available from the Symbian Developer Network [portal](#).

The second paper of this series will discuss further features available in the MobileMedia API, including support for audio playback, tone generation and capturing photographs. It will also discuss in more detail the implementation of the Mobile Media API available on the Symbian OS based Nokia 3650 camera phone.

## 6. Resources

Source code for `VideoPlayer` example MIDlet will be available here <http://www.symbian.com/developer/downloads/java.html>

Mobile Media API Specification

<http://jcp.org/aboutJava/communityprocess/final/jsr135/>

Mobile Media API Emulator for the J2ME Wireless Toolkit

<http://java.sun.com/products/mmapi/emulator/>

Series 60 Concept SDK incorporating support for the Mobile Media API

<http://forum.nokia.com>

Forum Nokia Papers

“A Brief introduction to the Mobile Media API”

“Technical Note: The Nokia 3650 Mobile Media API”