Get Involved
- About Java.net
- Adopt a JSR
- Create a Project
- Link an Offsite Project

Get Informed
- Articles
- Blogs
- Events
- Java Magazine
- Oracle University

J2ME Tutorial, Part 4: Multimedia and MIDP 2.0

September 27, 2005
Vikram Goyal

MIDP 2.0 , along with the optional Mobile Media API 1.1 (MMAPI), offers a range of multimedia capabilities for mobile devices, including playback and recording of audio and video data from a variety of sources. Of course, not all mobile devices support all the options, and MMAPI is designed in such a way that it takes full advantage of the capabilities that are available, while ignoring those that it cannot support. MIDP 2.0 comes with a subset of the MMAPI which ensures that if a device does not support MMAPI, you can still use a scaled down version. This scaled down version only supports audio (including tones) and excludes anything to do with video or images.

Contents

In this part four of the tutorial series, you will learn how to incorporate multimedia capabilities in your MIDlets. You will learn how to query a device to determine the supported capabilities. You will also find out how to initiate playback from different locations. But first, a little theory is required to understand the basics of MMAPI and its subset in MIDP 2.0.

Mobile Media API (MMAPI) background

MMAPI defines the superset of the multimedia capabilities that are present in MIDP 2.0. It started life as JSR 135 and is currently at version 1.1. The current version includes some documentation changes and security updates, and is distributed as an optional jar file in the J2ME wireless toolkit 2.2. Although the release notes for the toolkit state that MMAPI 1.1 is bundled, the actual version is 1.0. I have blogged about this before and have submitted an official bug with Sun.

The MMAPI is built on a high-level abstraction of all the multimedia devices that are possible in a resource-limited device. This abstraction is manifest in three classes that form the bulk of operations that you do with this API. These classes are the `Player` and `Control` interfaces, and the `Manager` class. Another class, the `DataSource` abstract class, is used to locate resources, but unless you define a new way of reading data you will probably never need to use it directly.

In a nutshell, you use the `Manager` class to create `Player` instances for different media by specifying `DataSource` instances. The `Player` instances thus created are configurable by using `Control` instances. For example, almost all `Player` instances would theoretically support a `VolumeControl` to control the volume of the `Player`. Figure 1 shows this process.

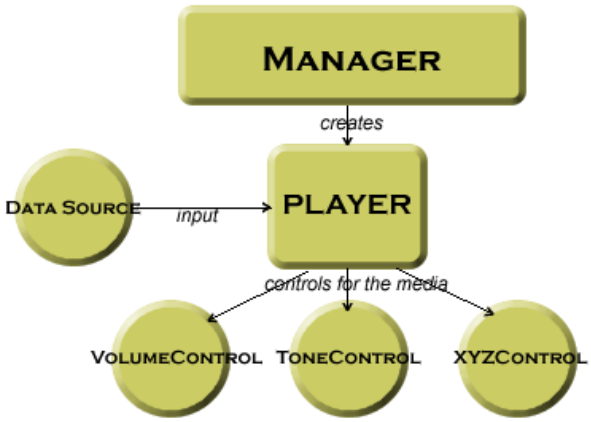

Figure 1. Player creation and management

`Manager` is the central class for creating players and it provides three methods to indicate the source of media. These methods, all static, are `createPlayer(DataSource source)`,

```
createPlayer(InputStream
  stream, String type)
```

and `createPlayer(String locator)`. The last method is interesting because it provides a URI style syntax for locating media. For example, if you wanted to create a `Player` instance on a web based audio file, you can use `createPlayer("http://www.yourwebsite.com/audio/song.wav")`. Similarly, to create a media `Player` to capture audio, you can use `createPlayer("capture://audio")`; and so on. Table 4.1 shows the supported syntax with examples.

Media Type	Example syntax
Capture audio	"`capture://audio`" to capture audio on the default audio capture device or "`capture://devmic0?encoding=pcm`" to capture audio on the devmic0 device in the PCM encoding
Capture video	"`capture://video`" to capture video from the default video capture device or "`capture://devcam0?encoding=rgb888&width=100&height=50`" to capture from a secondary camera, in rgb888 encoding mode and with a specified width and height
Start listening in on the built-in radio	"`capture://radio?f=105.1&st=stereo`" to tune into 105.1 FM frequency and stereo mode
Start streaming video/audio/text from an external source	"`rtp://host:port/type`" where type is one of audio, video or text
Play tones and MIDI	"`device://tone`" will give you a player that you can use to play tones or "`device://midi`" will give you a player that you can use to play MIDI

Table 4.1. List of supported protocols and example syntax

A list of supported protocols for a given content type can be retrieved by calling the method `getSupportedProtocols(String contentType)` which returns a `String` array. For example, if you call this method with the argument "`audio/x-wav`" it will return an array with three values in it: `http`, `file` and `capture` for the wireless toolkit. This lets you know that you can retrieve the content type "`audio/x-wav`", by using `http` and `file` protocols, and capture it using the `capture` protocol. Similarly, a list of supported content types for a given protocol can be accessed by calling the method `getSupportedContentTypes(String protocol)`. Thus, calling `getSupportedContentTypes("capture")` will return `audio/x-wav` and `video/vnd.sun.rgb565` for the wireless toolkit, indicating that you can capture standard audio and rgb565 encoded video. Note that passing null

in any of these methods will return all supported protocols and content types respectively.

Once a `Player` instance is created using the `Manager` class methods, it needs to go through various stages before it can be used. Upon creation, the player is in an `UNREALIZED` state and must be `REALIZED` and `PREFETCHED` before it can be `STARTED`. Realization is the process in which the player examines the source or destination media resources and has enough information to start acquiring them. Prefetching happens after realization and the player actually acquires these media resources. Both realization and prefetching processes may be time- and resource-consuming, but doing them before the player is started ensures that there is no latency when the actual start happens. Once a player is started, using the `start()` method, and is processing media data, it may enter the `PREFETCHED` state again when the media processing stops on its own (because the end of the media was reached, for example), you explicitly call the `stop()` method on the `Player` instance, or when a predefined time (called `TimeBase`) is reached. Going from `STARTED` to `PREFETCHED` state is like pausing the player, and calling `start()` on the `Player` instance restarts from the previous paused point (if the player had reached the end of the media, this means that it will restart from the beginning).

Good programming practice requires that you call the `realize()` and `prefetch()` methods before you call the `start()` method to avoid any latency when you want the player to start. The `start()` method implicitly calls the `prefetch()` method (if the player is not in a `PREFETCHED` state), which in turn calls the `realize()` method (if the player is not in a `REALIZED` state), but if you explicitly call these methods first, you will have a `Player` instance that will start playing as soon as you call `start()`. A player can go into the `CLOSED` state if you call the `close()` method on it, after which the Player instance cannot be reused. Instead of closing, you can deallocate a player by calling `deallocate()`, which returns the player to the `REALIZED` state, thereby releasing all the resources that it would have acquired.

Figure 2 shows the various states and the transitions between them.

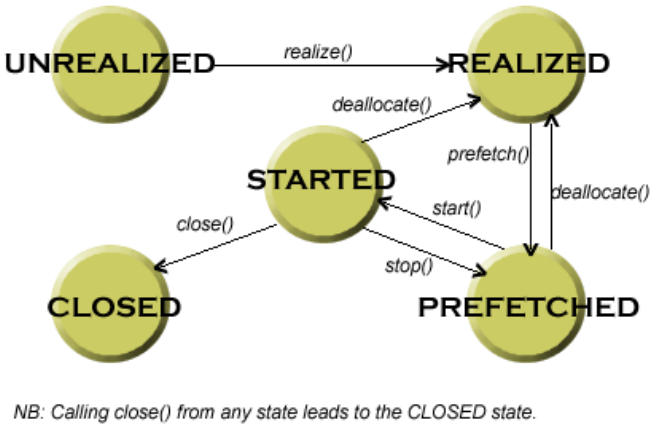

Figure 2. Media player states and their transitions

Notification of the transitions between different states can be delivered to attached listeners on a player. To this end, a `Player` instance allows you to attach a `PlayerListener` by using the method

```
addPlayerListener(PlayerListener
  listener)
```

. Almost all transitions states are notified to the listener via the method `playerUpdate(Player player, String event, Object eventData)`.

A player also enables control over the properties of the media that it is playing by using *controls*. A control is a media processing function that may be typical for a particular media type. For example, a `VideoControl` controls the display of video, while a `MIDIControl` provides access to MIDI devices' properties. There are, of course, several controls that may be common across different media, `VolumeControl` being an example. Because the `Player` interface extends the `Controllable` interface, it provides means to query the list of the available controls. You do this by calling the method `getControls()`, which returns an array of `Control`

instances, or `getControl(String controlType)`, which returns an individual `Control` (`null` if the `controlType` is not supported).

As I said earlier, MIDP 2.0 contains a subset of the broad MMAPI 1.1. This is to ensure that devices that only support MIDP 2.0 can still use a consistent method of discovery and usage that can scale if the broader API is present. The subset only supports tones and audio with only two controls for each, `ToneControl` and `VolumeControl`. Additionally, datasources are not supported, and hence, the `Manager` class in MIDP 2.0 is simplified and does not provide the `createPlayer(DataSource source)` method.

In the next few sections, you will learn how to play audio and video from a variety of sources in your multimedia MIDlets.

Using Mobile Media API (MMAPI)

Perhaps the easiest way to learn about MMAPI is to start by acquiring and playing a simple audio file. All multimedia operations, whether simple audio playback or complex video capture, will follow similar patterns. The `Manager` class will be used to create a `Player` instance using a `String` locator. The `Player` will then be realized, prefetched and played till it is time to close it. There are small differences, and I will point these out as we go along.

Figure 3 shows part of the operation of this simple audio file playback.

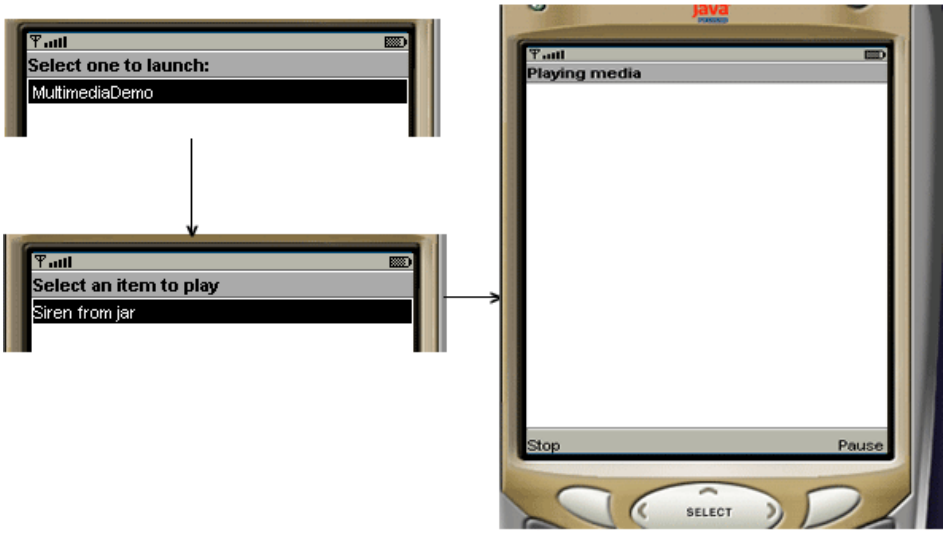

Figure 3. Simple audio file playback

When the user launches the MIDlet, he is given the option of playing the only item in the list, which is a "Siren from jar" item. On selecting this item, the screen changes to show the text "Playing media" and two commands become available to the user: pause and stop. The media starts playing in the background and the user can pause the audio or stop and return to the one item list.

The corresponding code is shown in Listing 1.

```
package com.j2me.part4;

import java.util.Hashtable;
import java.util.Enumeration;

import javax.microedition.lcdui.Item;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.Form;
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Command;
```

```java
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.CommandListener;

import javax.microedition.media.Player;
import javax.microedition.media.Control;
import javax.microedition.media.Manager;
import javax.microedition.media.PlayerListener;

public class MediaMIDlet extends MIDlet
  implements CommandListener, PlayerListener {

  private Display display;
  private List itemList;
  private Form form;

  private Command stopCommand;
  private Command pauseCommand;
  private Command startCommand;

  private Hashtable items;
  private Hashtable itemsInfo;

  private Player player;

  public MediaMIDlet() {
    display = Display.getDisplay(this);
    // creates an item list to let you select multimedia files to play
    itemList = new List("Select an item to play", List.IMPLICIT);

    // stop, pause and restart commands
    stopCommand = new Command("Stop", Command.STOP, 1);
    pauseCommand = new Command("Pause", Command.ITEM, 1);
    startCommand = new Command("Start", Command.ITEM, 1);

    // a form to display when items are being played
    form = new Form("Playing media");

    // the form acts as the interface to stop and pause the media
    form.addCommand(stopCommand);
    form.addCommand(pauseCommand);
    form.setCommandListener(this);

    // create a hashtable of items
    items = new Hashtable();

    // and a hashtable to hold information about them
    itemsInfo = new Hashtable();

    // and populate both of them
    items.put("Siren from jar", "file://siren.wav");
    itemsInfo.put("Siren from jar", "audio/x-wav");
  }

  public void startApp() {

    // when MIDlet is started, use the item list to display elements
    for(Enumeration en = items.keys(); en.hasMoreElements();) {
    itemList.append((String)en.nextElement(), null);
  }

    itemList.setCommandListener(this);

    // show the list when MIDlet is started
    display.setCurrent(itemList);
  }

  public void pauseApp() {
    // pause the player
    try {
      if(player != null) player.stop();
    } catch(Exception e) {}
  }

  public void destroyApp(boolean unconditional) {
    if(player != null) player.close(); // close the player
  }

  public void commandAction(Command command, Displayable disp) {

    // generic command handler

    // if list is displayed, the user wants to play the item
    if(disp instanceof List) {
      List list = ((List)disp);
```

```
      String key = list.getString(list.getSelectedIndex());

      // try and play the selected file
      try {
        playMedia((String)items.get(key), key);
      } catch (Exception e) {
        System.err.println("Unable to play: " + e);
        e.printStackTrace();
      }
    } else if(disp instanceof Form) {

      // if showing form, means the media is being played
      // and the user is trying to stop or pause the player
      try {

        if(command == stopCommand) { // if stopping the media play

          player.close(); // close the player
          display.setCurrent(itemList); // redisplay the list of media
          form.removeCommand(startCommand); // remove the start command
          form.addCommand(pauseCommand); // add the pause command

        } else if(command == pauseCommand) { // if pausing

          player.stop(); // pauses the media, note that it is called stop
          form.removeCommand(pauseCommand); // remove the pause command
          form.addCommand(startCommand); // add the start (restart) command
        } else if(command == startCommand) { // if restarting

          player.start(); // starts from where the last pause was called
          form.removeCommand(startCommand);
          form.addCommand(pauseCommand);
        }
      } catch(Exception e) {
        System.err.println(e);
      }
    }

  }

  /* Creates Player and plays media for the first time */
  private void playMedia(String locator, String key) throws Exception {

    // locate the actual file, we are only dealing
    // with file based media here
    String file = locator.substring(
      locator.indexOf("file://") + 6,
      locator.length());

    // create the player
    // loading it as a resource and using information about it
    // from the itemsInfo hashtable
    player = Manager.createPlayer(
        getClass().getResourceAsStream(file), (String)itemsInfo.get(key));

    // a listener to handle player events like starting, closing etc
    player.addPlayerListener(this);

    player.setLoopCount(-1); // play indefinitely
    player.prefetch(); // prefetch
    player.realize(); // realize

    player.start(); // and start
  }

  /* Handle player events */
  public void playerUpdate(Player player, String event, Object eventData) {

    // if the event is that the player has started, show the form
    // but only if the event data indicates that the event relates to newly
    // stated player, as the STARTED event is fired even if a player is
    // restarted. Note that eventData indicates the time at which the start
    // event is fired.
    if(event.equals(PlayerListener.STARTED) &&
      new Long(0L).equals((Long)eventData)) {

      display.setCurrent(form);
    } else if(event.equals(PlayerListener.CLOSED)) {

      form.deleteAll(); // clears the form of any previous controls
    }
  }
}
```

Listing 1. Simple Audio playback

You now have an audio player with code that leaves room to add playback for other media. To start, the MIDlet displays a list of items that can be played. At the moment, it only contains a single item called "Siren from jar". Notice that in the code, "Siren from jar" corresponds to a file-based access. This implies that the actual location of this media will be in the MIDlet jar file. When the user selects this item, a **Player** object is created specifically for it in the playMedia() method. This method loads this player, attaches a listener to it, prefetches it, realizes it and finally, starts it. Also notice that it plays the media continually.

Because the listener for the Player is the MIDlet class itself, the playerUpdate() method catches the player events. Thus, when the user starts hearing the siren, the Form is displayed, allowing the user to stop or pause it. Stop takes the user back to the list, while pause pauses the siren and replays from the paused marker when restarted.

Having created this generic class, it is now fairly easy to add other types of media to it. Besides audio, video is the primary media that would be played. To allow the MediaMIDlet to play video, the only change that needs to be made is in the playerUpdate() method, to create a video screen. This is shown in the following code snippet, with the changes highlighted in bold.

```
/* Handle player events */
  public void playerUpdate(Player player, String event, Object eventData) {

    // if the event is that the player has started, show the form
        // but only if the event data indicates that the event relates to newly
        // stated player, as the STARTED event is fired even if a player is
        // restarted. Note that eventData indicates the time at which the start
        // event is fired.
        If(event.equals(PlayerListener.STARTED) &&
          new Long(0L)Equals((Long)eventData)) {

        // see if we can show a video control, depending on whether the media
                 // is a video or not
                 VideoControl vc = null;
                 if((vc = (VideoControl)player.getControl("VideoControl")) != null) {
                   Item videoDisp =
                         (Item)vc.initDisplayMode(vc.USE_GUI_PRIMITIVE, null);
                   form.append(videoDisp);
                 }

                display.setCurrent(form);
        } else if(event.equals(PlayerListener.CLOSED)) {

          form.deleteAll(); // clears the form of any previous controls
        }
  }
```

The change allows you to play video files with the help of this MediaMIDlet as well. If the method determines that the player has a VideoControl, it exposes it by creating a GUI for it. This GUI is then attached to the current form. Of course, now you need to attach a video file to the list so that you can test it.

Recall that not all mobile phones will play all video files (or audio files for that matter). To see the list of video files supported by a device, use the **Manager**.getSupportedContentTypes(**null**) method. In the case of the Wireless Toolkit, video/mpeg is supported and therefore, this video will play. Add this to the list as shown here

```
items.put("Promo Video from jar", "file://promo.mpg");
itemsInfo.put("Promo Video from jar", "video/mpeg");
```

put the video in the res folder, and you should now be able to select and play it as well when the MIDlet is run. The result is shown in Figure 4.

Figure 4. Video playback with MediaMIDlet

Streaming media over the network

It is highly likely that the media files, especially video files, will not be distributed with your MIDlet, unless they are really small in size. For a successful MIDlet application, the ability to stream media over the network is essential. MediaMIDlet can play media over the network easily by specifying an HTTP based file. However, there are two issues to be considered in such a case.

First, media access over the network requires explicit permission from the end user. After all, this network usage will likely incur a cost to the user, which he must agree to. There are ways to ask this permission once and store the result within the MIDlet environment, but I will not go into too much detail over here. For the moment, it is sufficient to be aware of this issue.

Second, media access over the network can be a time consuming operation. This operation should not be done in the main application thread, in case it ties it up in an intermittent network and blocks forever. All network access should be done in a separate thread.

Keeping these two issues in mind, Listing 2 shows a modified version of Listing 1 (and the code that was added to it to play video files). The first issue is taken care of by the underlying Application Management Software (AMS). It explicitly asks for user permission once network access is required. The second issue is taken care of by separating the network media access code in its own thread.

```
package com.j2me.part4;

import java.util.Hashtable;
import java.util.Enumeration;

import javax.microedition.lcdui.Item;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Alert;
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.CommandListener;

import javax.microedition.media.Player;
import javax.microedition.media.Control;
import javax.microedition.media.Manager;
import javax.microedition.media.PlayerListener;
import javax.microedition.media.control.VideoControl;
```

```java
public class MediaMIDletV2 extends MIDlet
  implements CommandListener {

  private Display display;
  private List itemList;
  private Form form;

  private Hashtable items;

  public MediaMIDletV2() {

    display = Display.getDisplay(this);

    // creates an item list to let you select multimedia files to play
    itemList = new List("Select an item to play", List.IMPLICIT);


    // a form to display when items are being played
    form = new Form("Playing media");

    // create a hashtable of items
    items = new Hashtable();

    // and populate both of them
    items.put("Siren from web", "http://www.craftbits.com/j2me/siren.wav");
    items.put(
      "Promo Video from web",
      "http://www.craftbits.com/j2me/promo.mpg");
  }

  public void startApp() {

    // when MIDlet is started, use the item list to display elements
    for(Enumeration en = items.keys(); en.hasMoreElements();) {
      itemList.append((String)en.nextElement(), null);
    }

    itemList.setCommandListener(this);

    // show the list when MIDlet is started
    display.setCurrent(itemList);
  }

  public void pauseApp() {
  }

  public void destroyApp(Boolean unconditional) {
  }


  public void commandAction(Command command, Displayable disp) {

    // generic command handler

    // if list is displayed, the user wants to play the item
    if(disp instanceof List) {
      List list = ((List)disp);

      String key = list.getString(list.getSelectedIndex());

      // try and play the selected file
      try {
        playMedia((String)items.get(key));
      } catch (Exception e) {
        System.err.println("Unable to play: " + e);
        e.printStackTrace();
      }
    }

  }


  /* Creates Player and plays media for the first time */
  private void playMedia(String locator) throws Exception {

    PlayerManager manager =
      new PlayerManager(form, itemList, locator, display);
    form.setCommandListener(manager);
    Thread runner = new Thread(manager);
    runner.start();
  }
}

class PlayerManager implements Runnable, CommandListener, PlayerListener {
```

```java
Form form;
List list;
Player player;
String locator;
Display display;

private Command stopCommand;
private Command pauseCommand;
private Command startCommand;

public PlayerManager(Form form, List list, String locator, Display display) {
this.form = form;
this.list = list;
this.locator = locator;
this.display = display;

  // stop, pause and restart commands
  stopCommand = new Command("Stop", Command.STOP, 1);
  pauseCommand = new Command("Pause", Command.ITEM, 1);
  startCommand = new Command("Start", Command.ITEM, 1);

  // the form acts as the interface to stop and pause the media
  form.addCommand(stopCommand);
  form.addCommand(pauseCommand);
}

public void run() {

  try {
    // since we are loading data over the network, a delay can be
    // expected
    Alert alert = new Alert("Loading. Please wait ....");
    alert.setTimeout(Alert.FOREVER);
    display.setCurrent(alert);

    player = Manager.createPlayer(locator);

    // a listener to handle player events like starting, closing etc
    player.addPlayerListener(this);

    player.setLoopCount(-1); // play indefinitely
    player.prefetch(); // prefetch
    player.realize(); // realize

    player.start(); // and start
  } catch(Exception e) {
    System.err.println(e);
    e.printStackTrace();
  }
}

public void commandAction(Command command, Displayable disp) {
  if(disp instanceof Form) {
    // if showing form, means the media is being played
    // and the user is trying to stop or pause the player
    try {
      if(command == stopCommand) { // if stopping the media play
        player.close(); // close the player
        display.setCurrent(list); // redisplay the list of media
        form.removeCommand(startCommand); // remove the start command
        form.removeCommand(pauseCommand); // remove the pause command
        form.removeCommand(stopCommand);  // and the stop command
      } else if(command == pauseCommand) { // if pausing
        player.stop(); // pauses the media, note that it is called stop
        form.removeCommand(pauseCommand); // remove the pause command
        form.addCommand(startCommand); // add the start (restart) command
      } else if(command == startCommand) { // if restarting
        player.start(); // starts from where the last pause was called
        form.removeCommand(startCommand);
        form.addCommand(pauseCommand);
      }
    } catch(Exception e) {
      System.err.println(e);
    }
  }
}

/* Handle player events */
public void playerUpdate(Player player, String event, Object eventData) {

  // if the event is that the player has started, show the form
  // but only if the event data indicates that the event relates to newly
  // stated player, as the STARTED event is fired even if a player is
  // restarted. Note that eventData indicates the time at which the start
```

```
    // event is fired.
    if(event.equals(PlayerListener.STARTED) &&
      new Long(0L)Equals((Long)eventData)) {

      // see if we can show a video control, depending on whether the media
      // is a video or not
      VideoControl vc = null;
      if((vc = (VideoControl)player.getControl("VideoControl")) != null) {
        Item videoDisp =
          (Item)vc.initDisplayMode(vc.USE_GUI_PRIMITIVE, null);
        form.append(videoDisp);
      }

      display.setCurrent(form);
    } else if(event.equals(PlayerListener.CLOSED)) {

      form.deleteAll(); // clears the form of any previous controls
    }
  }

}
```

Listing 2. Media access over the network in its own thread

As you can see, all of the code that interacts with the media has been moved to the `PlayerManager` class, which is run in its own thread. Figure 5 shows how the interaction with the MIDlet will work now.

Figure 5. The process of media access over the network

Notice how the MIDlet asks for user permission to access the network before the player can be created. Permission granted once is assumed granted for the whole time the MIDlet is running; therefore, repeated network access to access other files does not bring up this screen.

This brings us to the end of this part in this tutorial series. I have only given a brief overview of the Mobile Media API and its subset in MIDP 2.0, with a few basic examples, which can be downloaded here. There are several other things that you can do with this API, like creating and playing tones, capturing audio and video and live radio streaming. Please explore the API documentation and use the examples given in this tutorial to experiment with the capabilities that this API provides.

Vikram Goyal is the author of Pro Java ME MMAPI.

Comments

Hello, I am not able download

by smanyam - 2010-10-24 13:07

Hello,

I am not able download the source zip. Please provide. Thanks.

Login or register to post comments