Pro Java ME MMAPI

Mobile Media API for Java Micro Edition

Vikram Goyal

Apress®

Pro Java ME MMAPI: Mobile Media API for Java Micro Edition

ISBN-13: 978-1-59059-639-5

ISBN-10: 1-59059-639-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Robert Virkus
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editor: Julie McNamee
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor and Artist: Kinetic Publishing Services, LLC
Proofreader: Nancy Riddiough
Indexer: Carol Burbo
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 4

Media Player Lifecycle and Events

As a MIDlet transitions between different states during its lifecycle, so does a Player instance that has a lifecycle of its own. A Player instance has many more states that it transitions between. These states are well defined, and transitions between them raise events that interested parties can listen to and respond accordingly.

In this chapter, you'll learn about these different states, the lifecycle of a Player instance, and how a Player instance transitions between these states. Finally, you'll learn how events generated during these transitions can be captured by interested listeners and acted upon.

Overview

A Player instance goes through five different states during its lifetime. The capability for an instance to go through these many states gives developers greater control over the working of an instance. These states are UNREALIZED, REALIZED, PREFETCHED, STARTED, and CLOSED.

A Player instance is guaranteed to go through all these states if started; that is, the instance is not just created but playback (or recording as the case may be) is initiated. Moving between states is not necessarily linear and can happen either due to programmatic control or some external or internal events. Movement from the CLOSED state to any other state is not possible.

Movement between different states results in events being fired for any listeners to act on. These events are delivered asynchronously and in the order that they are generated. The whole event delivery mechanism is extensible, which allows you to define application-specific events. Several system-level events are already defined that will satisfy most cases.

Exploring the Different `Player` States

MMAPI allows you to programmatically move between different states (except moving away from the CLOSED state). This gives you greater control over the way you manage the lifecycle of a Player instance, increases responsiveness, and allows manageability of these instances. This section explores these different states and the methods that allow you to gain this control.

In a nutshell, a Player instance starts life in the UNREALIZED state. It moves from this state to the REALIZED state when the user calls the realize() method. The REALIZED state gives way to the PREFETCHED state when the `prefetch()` method is called. Calling `start()` moves the instance to the STARTED state, and calling `close()` leads to the CLOSED state. This simple transition path is displayed in Figure 4-1.

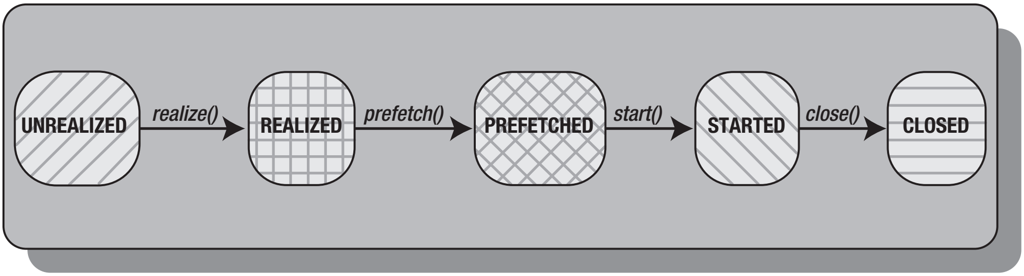

Figure 4-1. *A simple linear transition path for a* `Player` *instance*

Figure 4-1 shows the most *likely* transition path for a `Player` instance. Except for the CLOSED state, transitions can occur between the other states by calling special methods. These methods are covered shortly when the individual states are discussed.

Calling any of these methods to make a transition between different states is synchronous in nature. The methods don't return till the transition is complete. However, if any of these methods cannot make the transition, a `MediaException` is thrown to indicate so.

You can determine the current state of a `Player` instance by using the method `getState()`. This returns one of five constants defined in the `Player` interface corresponding to the five states shown in Figure 4-1. These constants are `UNREALIZED`, `REALIZED`, `PREFETCHED`, `STARTED`, and `CLOSED`.

Let's examine each of these states individually to see what they mean and how movement between them is not always so linear.

■ **Note** Most of these state transition methods are implemented in Java, as opposed to the code for parsing and decoding multimedia data, which is implemented in the native language of the device on which the MIDlet is running. Parsing and decoding are memory-intensive operations and implementing them in Java would sacrifice performance. Media transition methods, on the other hand, are not CPU-intensive and can be safely implemented in Java, as most of them are. Some parts of these methods may be implemented in native language to take advantage of device-specific performance features.

UNREALIZED

A `Player` instance starts life in an UNREALIZED state. When you use the `Manager` class to create a `Player` instance using any of the three `createPlayer()` methods, it creates a barely usable instance in the UNREALIZED state. An UNREALIZED `Player` is of no use because it doesn't have enough information to start functioning. It needs to acquire resources, such as audio and recording hardware on the device; set up in memory buffers for acquiring the media content; and communicate with the location of the media data. All these processes are performed in other states.

A Player instance moves away from the UNREALIZED state when the realize() method is called, which if successful, moves it to the REALIZED state. If unsuccessful, a MediaException is thrown, and the instance remains in the UNREALIZED state. If the realize() method blocks for a long time because it is a synchronous method, you can attempt to call the deallocate() method on the Player instance, which tries to keep the method in the UNREALIZED state. You'll learn more about this in the upcoming "REALIZED" section.

Not many actions can be performed on an UNREALIZED Player instance. For example, you cannot retrieve any controls from this instance using the getControl() or the getControls() methods, because the instance doesn't have enough information to generate the controls. You cannot change the playback time of the media, provided that it allows you to change the media time in the first place, with the setMediaTime() method. You cannot even retrieve or change the instance's TimeBase for synchronization using the getTimeBase() or setTimeBase() methods. A call to any of these methods in the UNREALIZED state results in an IllegalStateException.

The only useful operation that you can do with an UNREALIZED instance, besides realizing or closing it, is to set the number of times the instance should loop using the setLoopCount() method. You can also retrieve the likely duration of the media, which almost always returns –1, indicating an unknown time (TIME_UKNOWN constant in the Player interface).

The Player interface has a static integer constant to represent this state, UNREALIZED.

Figure 4-2 summarizes the UNREALIZED state and its transitions.

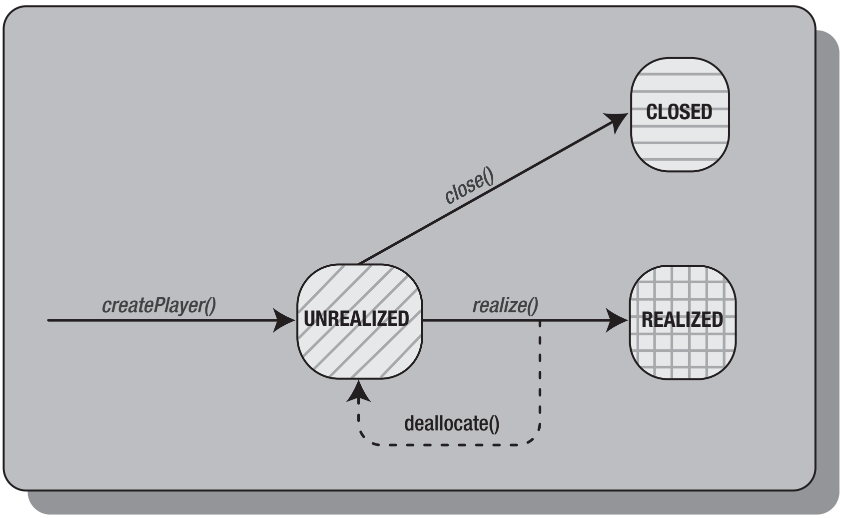

Figure 4-2. *UNREALIZED state transitions*

REALIZED

A Player instance moves from the UNREALIZED to the REALIZED state when the realize() method is called. The realize() method can be time consuming because it actually retrieves the media data. If the UNREALIZED state represents that the instance has connected to its media location, in a REALIZED state, it has in all probability retrieved this data (except for data that is streaming in nature).

After it has been REALIZED, a `Player` instance cannot go back to the UNREALIZED state. As you learned in the last section, a `realize()` method that is taking too long to return can be preempted by calling the `deallocate()` method, which keeps the instance in the UNREALIZED state. But from a REALIZED state itself, an instance can only go to the PREFETCHED or the CLOSED states. Figure 4-3 shows the REALIZED state transitions.

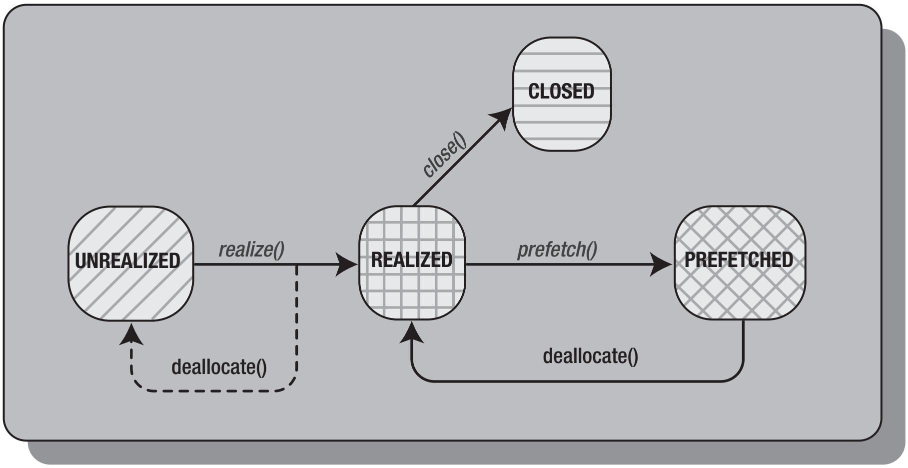

Figure 4-3. *REALIZED state transitions*

What actually happens within the `realize()` method is dependent on individual MMAPI implementations. However, after the method returns successfully, it is guaranteed that the underlying media data has been examined, and any available controls are up for use. Any resources required to play back the data that require exclusive use by your MIDlet on the device that it is running are also guaranteed *not* to have been acquired.

The `realize()` method throws, besides `IllegalStateException` and `MediaException`, a `SecurityException` as well. The `IllegalStateException` is thrown if the instance is in a CLOSED state; the `MediaException` is thrown if a media-specific error occurs; and the `SecurityException` is thrown if the MIDlet doesn't have permission to access the media file. This is closely related to Digital Rights Management (DRM) of digital data.

■ **Note** DRM is an industry term that refers to a service or technology to control access to digital data.

Because MMAPI is used to access some of the most popular digital data, such as music and video, it allows implementations to plug in a DRM technology to control this access. This allows very simple control over the media data. For example, `SecurityException` is thrown by the realize() method if the DRM indicates that the right of the user to use the media data has expired (because it could only be used within a specific timeframe, it could only be played once, or for any other DRM-based reason). The realize() method implementation calls the DRM technology built in to the device to make this call.

Ideally, a call like this happens when the Player instance is created. DRM kicks in when createPlayer() is called on a Manager class, and you'll receive a SecurityException for trying to create an instance on an expired or inaccessible content. (You'll also receive a SecurityException for protocol-specific restrictions built in to the user's device, such as accessing the network when no permission to access the network has been given.) However, you might want to replay content without needing to create another Player instance on the same media data (as you saw in Chapter 3 when you cached Player instances). In those cases, realize() checks to make sure that DRM rules haven't been breached; if they have been, realize() throws a SecurityException.

The Player interface has a static integer constant that represents this state, REALIZED.

PREFETCHED

In the PREFETCHED state, a Player instance is in the best possible state to get started with the playback (or recording in case of a player for capturing data) of media. The instance has decoded the data and acquired access to any exclusive resources required for playback (or recording). A Player instance that hasn't been prefetched cannot be started.

Player instances move into the PREFETCHED state when the user calls the prefetch() method on a REALIZED instance. Calling deallocate() does the reverse, that is, moves a PREFETCHED instance to the REALIZED state, thereby releasing any exclusive resources acquired to move into the PREFETCHED state.

Note that calling prefetch() doesn't necessarily mean that all the media data would have been decoded and ready for playback (or recording). It just means that most of the processing has been done and the media can be played back (or recorded) with the minimum possible latency. To accentuate this point, if an instance is already in the PREFETCHED state, and you use the prefetch() method, the instance will try and minimize the latency even further. However, the reduction in startup times is not guaranteed, and different implementations of MMAPI will differ in what they exactly do if prefetch() is called twice. If you do call prefetch() on an already PREFETCHED instance, it is guaranteed not to throw any errors.

An IllegalStateException is thrown if you call this method on a CLOSED instance. A MediaException is also thrown if an error occurs when processing or decoding the media data. However, this same exception is thrown if an exclusive resource cannot be acquired. For example, if a Player instance requires exclusive access to the audio hardware on a device, and this is not available because it is being used by some other instance, a MediaException will be thrown to indicate this. Although there's no way to differentiate between the two reasons for MediaException, in the case of the latter reason, you can call prefetch() again and if exclusive access is now possible, the instance will move to the PREFETCHED state.

Similar to the realize() method, calling the prefetch() method may throw a SecurityException if the Player instance has insufficient permissions to either decode media data or acquire exclusive resources.

Figure 4-4 shows the possible state transitions in the PREFETCHED state.

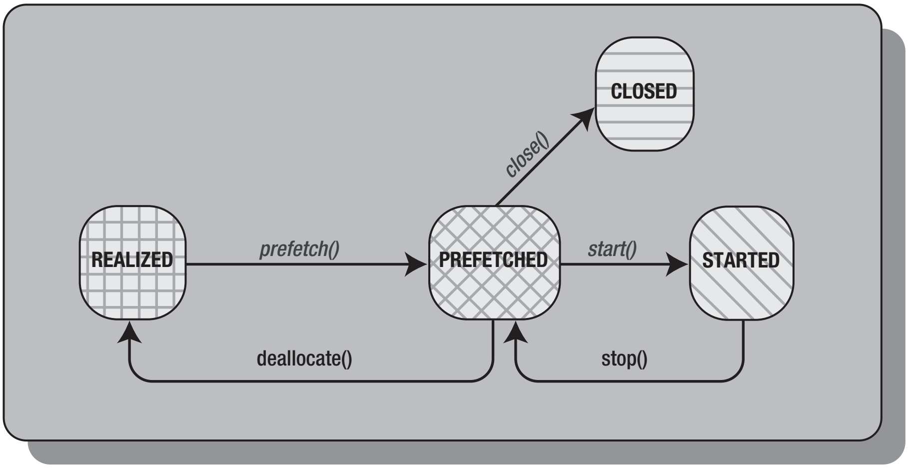

Figure 4-4. *PREFETCHED state transitions*

As you can see, there is no direct transition path for an `UNREALIZED` instance to go to the PREFETCHED state. This is why, if you call `prefetch()` on an `UNREALIZED` instance, it implies a call to the `realize()` method first. However, a transition from the PREFETCHED state to the REALIZED state can occur by using the `deallocate()` method. By doing so, you allow your instance to give up the exclusive resources acquired by your instance for other instances (or other MIDlets, applications, or AMS) to use.

An instance can arrive in the PREFETCHED state from the STARTED state as well. This transition can occur in several ways as explained in detail in the next section.

The `Player` interface has a static integer constant that represents this state, `PREFETCHED`.

STARTED

A `Player` instance in the STARTED state is playing back (or recording, streaming, and so on) actual media. This is the most useful state that an instance can be in. This state is achieved by calling the `start()` method on a `PREFETCHED` instance. However, note that calling the `start()` method doesn't guarantee that the instance will actually move immediately into the STARTED state. By calling the `start()` method, you are telling the instance to move into the STARTED state as soon as possible. Only when the `start()` method returns successfully is the instance considered to be in this state.

Of course, the instance may not be able to successfully move into the STARTED state when this method is called. Besides a `SecurityException`, thrown if there is not enough permission to start the media, a `MediaException` may be thrown if an error occurs when processing the media for playback (or recording). When any of these exceptions is thrown, the instance remains in the PREFETCHED state. As expected, an `IllegalStateException` is thrown if you try calling `start()` on an instance that is in the CLOSED state.

STARTED is the only state that has an automatic transition based on the state of the media playback (or recording). If you call `start()` on a `Player` instance, it automatically moves to the PREFETCHED state if the end of media playback is reached; that is, there is nothing left to play. STARTED also automatically moves to the PREFETCHED state if a preset stop time is reached. Preset stop times are set using the `StopTimeControl`.

Media that has a very short playback time moves to the PREFETCHED state almost immediately. For example, consider that you are trying to play an audio file and want to initiate some action when its `Player` instance is in the STARTED state. When you call the `start()` method, the instance temporarily moves to this state, but before you can react to the STARTED event, the playback would be over for a very short audio file, and the instance would have moved back to the PREFETCHED state. Thus, there are no guarantees for successfully acting on a STARTED instance because of this automatic transition.

Besides these automatic transitions, a `Player` instance also moves back to the PREFETCHED state when the `stop()` method is called in the STARTED state and the method returns successfully. The effect of the `stop()` method is to pause the instance at the current media time. (Note that there is *no* corresponding STOPPED state for the STARTED state. When stopped, an instance is in the PREFETCHED state.) Similar to the `start()` method, the `stop()` method also throws the `IllegalStateException` and `MediaException`. The first exception is thrown if this method is called on a `CLOSED` instance, whereas the second is thrown if the instance cannot be stopped.

If you call the `start()` method again, after you have stopped a previously started instance, it resumes at the media time that it was stopped at, effectively restarting paused media. This can, of course, be overridden by using the method `setMediaTime()`, which allows you to restart the playback from whenever you want it to. As expected, this will not work for media that is being recorded, and may or may not be supported for streaming data. In cases where this is not supported, calling `setMediaTime()` will throw a `MediaException`.

After a `Player` instance is in the STARTED state, calling either `setTimeBase()` or `setLoopCount()`throws an `IllegalStateException`. This makes sense. A `Player`'s `TimeBase` allows it to synchronize itself with other instances via the internal clock. After the instance has already started, this clock will be out of sync if changed midstream. Similarly, changing the number of times the instance must play back in the STARTED state will cause confusion over this count.

If you call the `start()` method on an `UNREALIZED` instance, it implies a call to the `realize()` and `prefetch()` methods, in that order. If you call it on a `REALIZED` instance, it implies a call to the `prefetch()` method first. In short, no direct transition occurs between the UNREALIZED state and the STARTED state on the one hand, and the REALIZED and STARTED state on the other, with the `start()` method taking care of these transitions for you. This is why listings in Chapter 3 were able to get away with calling the `start()` method only. However, calling the `start()` method on a `PREFETCHED` instance is always better than calling on `UNREALIZED` or `REALIZED`, because it reduces the overall startup time. You should only call the `start()` method after you've brought the instance to the PREFETCHED state, rather than letting the `start()` method bring it to that state.

Figure 4-5 shows the state transitions for the STARTED state.

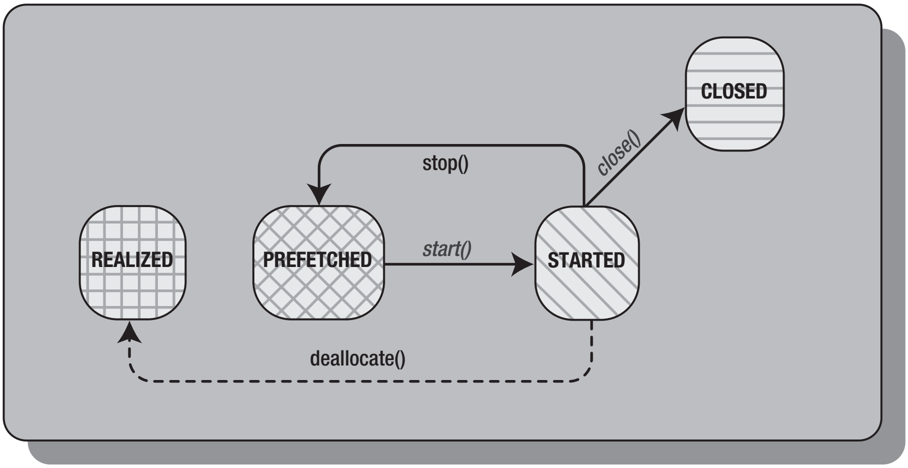

Figure 4-5. *STARTED state transitions*

You can call the `deallocate()` method on a `STARTED` instance, which internally implies a call to the `stop()` method first, and thus, the state of the instance transitions from STARTED to PREFETCHED to REALIZED, if both methods return successfully.

The `Player` interface has a static integer constant that represents this state, `STARTED`.

CLOSED

A `Player` instance in the CLOSED state is no longer usable. No methods can be called on it in this state with the exception of the `getState()` method. All other states transition to this state when the `close()` method is called, but there are no automatic transitions. If you call `close()` in the CLOSED state, no exception is thrown and the call is ignored. In fact, this method does not throw any exceptions; if any errors occur, the method returns silently and moves the instance to the CLOSED state anyway. All resources held by the instance are released, including any connections, exclusive device resources, and internal buffers.

■ **Note** Although the MMAPI specification does not say so, calling `close()` during different states causes different actions to be performed. Because the specification is silent on this issue, different implementations implement the actions in their own way. However, most implementations try to call the `deallocate()` method before performing any closeup actions. Recall that the `deallocate()` method can be called in all states (except, of course, the CLOSED state, whereas in the REALIZED state, the `deallocate()` call is ignored). Calling the `deallocate()` method in the STARTED state causes the `stop()` method to be called first. Thus, in all probability, if you call `close()` on a `STARTED Player` instance, it will go through the following cleanup methods: `close()` ➤ `stop()` ➤ `deallocate()`.

The `Player` interface has a static integer constant that represents this state, `CLOSED`.

Responding to `Player` Events

Each state transition and many other events generate a regular stream of notifications for any listener objects interested in a `Player` instance. This event delivery mechanism is implemented using an asynchronous model that is similar to most Java event delivery mechanisms.

The key class in this mechanism is the `PlayerListener` interface. Any class may implement this interface, register this implementation with the target `Player` instance, and start receiving notifications as the instance goes through its lifecycle. Several events are defined within this interface that cover a comprehensive list of `Player` events. You can create your own events as well and listen and react to them.

The `PlayerListener` interface defines only one method that implementations must implement. This method—`playerUpdate(Player player, String event, Object eventData)`—is invoked when an event takes place. To register an implementation with a `Player` instance, you use the method `addPlayerListener(PlayerListener listener)`; to remove the instance, you use `removePlayerListener(PlayerListener listener)`. Multiple listeners can be attached to a single instance, and multiple instances can send their events to the same listener. Because the `playerUpdate()` method receives the instance as an argument, it knows how to differentiate between the different instances.

As a simple example, let's modify Listing 3-1 from Chapter 3 to receive notifications from the simple `Player` instance that was created in that listing. The new code will display the events on the device screen as they are received. Listing 4-1 shows this modified code in the MIDlet called `EchoEventsMIDlet`.

Listing 4-1. `EchoEventsMIDlet` *Echoes Player Events Onscreen*

```
package com.apress.chapter4;

import javax.microedition.media.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class EchoEventsMIDlet extends MIDlet implements PlayerListener {

  private StringItem stringItem;

  public void startApp() {

    try {

      Form form = new Form("Player State");
      stringItem = new StringItem("", null)
      form.append(stringItem);
      Display.getDisplay(this).setCurrent(form);

      Player player = Manager.createPlayer(
        getClass().getResourceAsStream(
        "/media/audio/chapter4/baby.wav"), "audio/x-wav");
      player.addPlayerListener(this);
```

```
      player.start();

    } catch(Exception e) {
      e.printStackTrace();
    }
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }

  public void playerUpdate(Player player, String event, Object eventData) {
    stringItem.setText(event);
    System.err.println(event);
  }
}
```

The `EchoEventsMIDlet` acts as the listener for the `Player` instance that it creates by implementing `PlayerListener` and adding the `playerUpdate()` method. When created, the `Player` instance registers this MIDlet by using the method `addPlayerListener(this)`. Any event that is now generated by the corresponding event is delivered to the `playerUpdate()` method.

The other change in this MIDlet from Listing 3-1 is to create a `Form` object with a single `StringItem` on it to promptly display onscreen and print the current event received by the `playerUpdate()` method to the error output stream.

Running this MIDlet returns mostly consistent results because the Sun Java Wireless Toolkit's DefaultColorPhone emulator fires an extra `volumeChanged` event.

■ **Note** You may not notice the `volumeChanged` event on the DefaultColorPhone emulator because it may happen too fast depending on the time it takes for the media file to enter the STARTED state. Check the error output stream. The `volumeChanged` event is fired when the `Player` instance enters the PREFETCHED state (on the Sun MMAPI implementation).

The Motorola emulator and actual C975 device do not fire this `volumeChanged` event. The events that are fired and received by all three environments are the `started` and the `endofmedia` events. As you may guess, the `started` event represents when an instance has entered the STARTED state. The `endofmedia` event is delivered when no more media is left to play (or record/stream). This event is delivered each time an instance that is set to loop reaches the end of the media for each loop.

Of course, many more events than the two (or three if you consider the DefaultColorPhone) are fired by this simple example. Table 4-1 lists all the events defined in the MMAPI specification in the `PlayerListener` interface defined as constants.

Table 4-1. *A Complete List of* `Player` *Events Defined in the* `PlayerListener` *Interface*

Player Event Constant	Constant Value	When Fired
`BUFFERING_STARTED`	`bufferingStarted`	When an instance has started buffering media data for processing or playback.
`BUFFERING_STOPPED`	`bufferingStopped`	When an instance has exited the buffering stage.
`CLOSED`	`closed`	When an instance is closed.
`DEVICE_AVAILABLE`	`deviceAvailable`	When a system resource required by a `Player` instance becomes available for use.
`DEVICE_UNAVAILABLE`	`deviceUnavailable`	When a system resource required by a `Player` instance becomes unavailable. This event must precede the previous event.
`DURATION_UPDATED`	`durationUpdated`	When the duration of previously unknown media data becomes available.
`END_OF_MEDIA`	`endOfMedia`	When an instance has reached the end of the media during the current loop.
`ERROR`	`error`	When an error, which is usually fatal, occurs.
`RECORD_ERROR`	`recordError`	When an error occurs during recording (audio or video).
`RECORD_STARTED`	`recordStarted`	When recording of media data has started.
`RECORD_STOPPED`	`recordStopped`	When recording of media data has stopped.
`SIZE_CHANGED`	`sizeChanged`	When the size of a video display has changed for whatever reason.
`STARTED`	`started`	When the instance has entered the STARTED state.
`STOPPED`	`stopped`	When the instance has paused due to the `stop()` method being called.
`STOPPED_AT_TIME`	`stoppedAtTime`	When the instance has paused due to the `StopTimeControl`'s `setStopTime()` method.
`VOLUME_CHANGED`	`volumeChanged`	When the volume of an audio device is changed.

If you look at the signature of the `playerUpdate()` method, you'll see that it takes three parameters. The first is the `Player` instance that has thrown the event, the second is the actual event, and the third is the `eventData` as an `Object`. The `eventData` is interesting because it contains specific information about each event that can help you do something when the particular event is fired. For example, when the `stopped` event is received by this method, the `eventData` is a `Long` object identifying the media time when the corresponding `Player` instance is stopped. Similarly, the `started` event's `eventData` contains the media time when the media is started. Almost each event carries some useful information in the corresponding `eventData`; Table 4-2 shows the complete list.

Table 4-2. *Events and Corresponding Event Data*

Event	Event Data
BUFFERING_STARTED	A Long object designating the time when buffering has started.
BUFFERING_STOPPED	A Long object designating the time when buffering has stopped.
CLOSED	Event data is null when this event is fired.
DEVICE_AVAILABLE	A String object that is the name of the device that is now available.
DEVICE_UNAVAILABLE	A String object that is the name of the device that is not available.
DURATION_UPDATED	A Long object designating the new duration of the media.
END_OF_MEDIA	A Long object that contains the media time when the Player instance reached the end of media and stopped.
ERROR	A String that contains the error message.
RECORD_ERROR	A String that contains the error message.
RECORD_STARTED	A Long object that designates the media time when recording has started.
RECORD_STOPPED	A Long object that designates the media time when recording has stopped.
SIZE_CHANGED	A VideoControl control object that contains information about the new size.
STARTED	A Long object designating the media time when the Player instance is started.
STOPPED	A Long object designating the media time when the Player instance is stopped (paused).
STOPPED_AT_TIME	Similar to STOPPED, the eventData contains the media time when the Player instance is stopped in the form of a Long object.
VOLUME_CHANGED	A VolumeControl control object that contains information about the new volume.

Note that in MMAPI all times are measured in microseconds, not milliseconds.

Understanding the Event Delivery Mechanism

The event delivery mechanism in MMAPI is based on an asynchronous model that allows you to create multimedia applications that do not block the main application thread. This means that events are fired using an event delivery thread separate from the main application thread. This thread may or may not be in existence till an actual event is to be delivered. For example, in the Sun's MMAPI reference implementation, this thread is only created when the first event is fired. Even then, this thread remains active only for another five seconds, after which, if no more events are delivered, the thread exits. A new thread is created the next time an event needs to be delivered. Most actual commercial implementations follow a similar model that only differs in the time that they stay alive for; however, they are guaranteed to all follow this asynchronous nature of event delivery.

The MMAPI also guarantees that events will be delivered to their respective listeners in the order they are generated. This way, events that occur very fast after one another are guaranteed to be received by the registered listeners in order, without getting overwhelmed by newer events. For example, suppose you start playing a media file, which would fire a `STARTED` event. However, if the media file is short, it will end very quickly and generate an `END_OF_MEDIA` event almost immediately after it sends the `STARTED` event. The listener is guaranteed to receive the `STARTED` event before the `END_OF_MEDIA` event even if it occurs nearly simultaneously.

Of course, if an error occurs at any stage during a `Player` instance creation or usage so that the instance cannot continue working, the event delivery mechanism sends an `ERROR` event. The receipt of this event implies that the instance is unusable and is in a CLOSED state.

Creating an Event Handling Class

In Chapter 3, you created a MIDlet that allowed you to select a media audio file from a list to play and control its volume. In this section, you'll create an event handling class and attach it to the functional `Player` instances created in that MIDlet. This event handling class is basic, but it gives you an idea of how to listen for events, handle them accordingly, and use the `event` and `eventData` parameters. Listing 4-2 shows this event handling class, called `EventHandler`.

Listing 4-2. `EventHandler` *Is the Listener for Functional* `Player` *Instances Created in the Previous Chapter*

```
package com.apress.chapter4;

import javax.microedition.media.*;
import javax.microedition.media.control.*;
import javax.microedition.lcdui.StringItem;

public class EventHandler implements PlayerListener {

  private StringItem item;

  public EventHandler(StringItem item) {
    this.item = item;
  }

  public void playerUpdate(Player player, String event, Object eventData) {

    if(event.equals(PlayerListener.VOLUME_CHANGED)) {

      // a player's volume has been changed
      VolumeControl vc = (VolumeControl)eventData;
      updateDisplay("Volume Changed to: " + vc.getLevel());

      if(vc.getLevel() > 60)  {
        updateDisplay("Volume higher than 60 is too loud");
        vc.setLevel(60);
      }
    } else if(event.equals(PlayerListener.STOPPED)) {
```

```
        // player instance paused
        updateDisplay("Player paused at: " + (Long)eventData);
      } else if(event.equals(PlayerListener.STARTED)) {

        // player instance started (or restarted)
        updateDisplay("Player started at: " + (Long)eventData);
      } else if(event.equals(PlayerListener.END_OF_MEDIA)) {

        // player instace reached end of loop
        updateDisplay("Player reached end of loop.");
      } else if(event.equals(PlayerListener.CLOSED)) {

        // player instance closed
        updateDisplay("Player closed.");
      } else if(event.equals(PlayerListener.ERROR)) {

        // if an error occurs, eventData contains the error message
        updateDisplay("Error Message: " + (String)eventData);
      }
    }

    public void updateDisplay(String text) {

      // update the item on the screen
      item.setText(text);

      // and write to error stream as well
      System.err.println(text);
    }

}
```

The `EventHandler` constructor accepts a `StringItem` screen item to which it can write updates as it receives events. The `playerUpdate()` method is where the updates are written to the screen based on the event that has occurred.

If you change the volume of a `Player` instance, the associated `VolumeControl` is retrieved from the `eventData` after casting it appropriately. You can then query the new volume level from this control.

■**Note** You can, of course, retrieve the same `VolumeControl` by querying the associated `Player` instance with the `getControl(" VolumeControl ")` method call that returns a reference to the same instance as referenced by the `eventData` parameter. The direct referencing `eventData` omits a method call, whereas `getControl()` method omits the use of a cast.

Here, the handler informs the user that volume over 60 is too loud and resets the volume back to 60. Note that resetting the volume in turn generates another `VOLUME_CHANGED` event!

The rest of the events are handled accordingly, and you can use the associated event data with appropriate casts. The `updateDisplay()` method updates the screen as well as writes message to the error output stream because some of the messages on the screen will happen too quickly.

The event generating `Player` instances now need to be told to send the instances to this handling class. This is done in the `CachingAudioPlayerCanvas` class where these instances are first created. This class is now modified to add a `StringItem` to display the messages from the event handler, create the `EventHandler` class, and set each `Player` instance up with this class as the listener. These changes are shown in bold in Listing 4-3.

Listing 4-3. *Enabling Event Handling in the* `CachingAudioPlayerCanvas` *Class*

```
package com.apress.chapter4;

import java.util.*;
import javax.microedition.lcdui.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;

public class CachedAudioPlayerCanvas implements ItemStateListener {

  // the parent MIDlet
  private CachingAudioPlayer parent;

  // form that contains canvas elements
  private Form form;

  // gauge to allow user to manipulate volume
  private Gauge gauge;

  // the volume control
  private VolumeControl volume;

  // the player used to play media
  private Player player;

  // is the player paused?
  private boolean paused = false;

  // to display event info
  private StringItem eventInfo;

  // the event handler
  private EventHandler handler;

  private Hashtable players;
```

```
public CachedAudioPlayerCanvas(CachingAudioPlayer parent) {

  this.parent = parent;

  // create form and add elements and listeners
  form = new Form("");
  gauge = new Gauge("Volume: 50", true, 100, 50);
  eventInfo = new StringItem("", null);
  form.append(gauge);

  // add the event info string item
  form.append(eventInfo);

  // create the EventHandler
  handler = new EventHandler(eventInfo);

  form.addCommand(parent.exitCommand);
  form.addCommand(parent.backCommand);
  form.setCommandListener(parent);

  // a change in volume gauge will be handled by this class
  form.setItemStateListener(this);

  players = new Hashtable();
}

public void playMedia(String locator) {

  try {

    // first look for an existing instance
    player = (Player)players.get(locator);

    if(player == null) {

      // create the player for the specified string locator
      player = Manager.createPlayer(
        getClass().getResourceAsStream(locator), "audio/x-wav");

      // add the EventHandler as a listener
      player.addPlayerListener(handler);

      // fetch it
      player.prefetch();

      // put this instance in the Hashtable
      players.put(locator, player);
```

```
      }

      // get the volume control
      volume = (VolumeControl)player.getControl("VolumeControl");

      // initialize it to 50
      volume.setLevel(50);

      // initialize the gauge
      gauge.setValue(volume.getLevel());
      gauge.setLabel("Volume: " + volume.getLevel());

      // play it twice
      player.setLoopCount(2);

      // start the player
      player.start();

      // set the title of the form
      form.setTitle("Playing " + locator);

    } catch(Exception e) {
      e.printStackTrace();
    }
  }

... rest of the code omitted as it doesn't change from Listing 3-4 ...
```

A single `EventHandler` instance is used for all three `Player` instances that are created. Because `playerUpdate()` receives the instance that generated the event, it's easy to distinguish between these instances, if necessary.

Handling a Custom Event

As you may realize by now, there is no special event class in the MMAPI. That is, events are distinguished as `String` constants in the `PlayerListener` interface. To create, rather to handle, a custom event, you do not need to extend any other class.

Custom event creation is primarily designed for MMAPI implementations. This means that the MMAPI specification, having designed its own mandatory events, makes it open for MMAPI implementations to create and broadcast their own events. The description of these events would be made clear in the documentation for each implementation, and the events are likely to be named in the reverse domain name convention. For example, the MMAPI reference implementation from Sun defines a custom event called `com.sun.midi.lyrics`, which is a Sun-specific event for karaoke lyrics.

Although the `PlayerListener` interface provides for events that are most common, in some special cases, you may want to define your own. For example, let's say you wanted to do something special that requires an event to be raised whenever an audio file has been played halfway through. None of the predefined events will satisfy this requirement, so you'll need to raise and handle your own custom event. But how do you actually create and raise an event?

The short answer is that you can't. Unless you are ready to implement your own version of a `Player` instance that handles the type of media that you are after. This is not an easy task and requires you to handle all the steps required in realizing, prefetching, and decoding, not to mention interfacing with the controls that it exposes. Further, you have to use your own version over the version supplied with the MMAPI implementation that you are working with. After you have accomplished these difficult tasks, you may be able to plug in and raise your own event.

Handling custom events is, as you may expect, much easier. You only need to know the name of the event and the type of `eventData` that it exposes to be able to use it in the `playerUpdate()` method. Thus, the following code fragment will catch the `com.sun.midi.lyrics` event, and the event data exposed will be a `byte` array:

```
if(event.equals("com.sun.midi.lyrics")) {
  byte[] data = (byte[])eventData;
}
```

Note that the MMAPI specification states that to catch standard events in the `playerUpdate()` method, you should use the reference equality check, and for custom events, you should use the object equality check. Thus, `(event == PlayerListener.CLOSED)` should be preferred over `event.equals(PlayerListener.CLOSED)`, and `event.equals("com.sun.midi.lyrics")` must be used for custom events. Standard events are automatically interned because they are constants; therefore, using the reference check will be faster than the object equality check. However, the same cannot be guaranteed for custom events, so you must always use the object equality test. The `EventHandler` in Listing 4-2 used the object equality test and is now converted to use the reference check in Listing 4-4 to make it more responsive.

Listing 4-4. *Converting* `EventHandler` *to Use Reference Checking Instead of Object Equality*

```
package com.apress.chapter4;

import javax.microedition.media.*;
import javax.microedition.media.control.*;
import javax.microedition.lcdui.StringItem;

public class EventHandler implements PlayerListener {

  private StringItem item;

  public EventHandler(StringItem item) {
    this.item = item;
  }

  public void playerUpdate(Player player, String event, Object eventData) {
```

```
  if(event == (PlayerListener.VOLUME_CHANGED)) {

    // a player's volume has been changed
    VolumeControl vc = (VolumeControl)eventData;
    updateDisplay("Volume Changed to: " + vc.getLevel());

    if(vc.getLevel() > 60)  {
      updateDisplay("Volume higher than 60 is too loud");
      vc.setLevel(60);
    }
  } else if(event == (PlayerListener.STOPPED)) {

    // player instance paused
    updateDisplay("Player paused at: " + (Long)eventData);
  } else if(event == (PlayerListener.STARTED)) {

    // player instance started (or restarted)
    updateDisplay("Player started at: " + (Long)eventData);
  } else if(event == (PlayerListener.END_OF_MEDIA)) {

    // player instance reached end of loop
    updateDisplay("Player reached end of loop.");
  } else if(event == (PlayerListener.CLOSED)) {

    // player instance closed
    updateDisplay("Player closed.");
  } else if(event == (PlayerListener.ERROR)) {

    // if an error occurs, eventData contains the error message
    updateDisplay("Error Message: " + (String)eventData);
  }
}

public void updateDisplay(String text) {

  // update the item on the screen
  item.setText(text);

  // and write to error stream as well
  System.err.println(text);
}
```

```
}
```

Due to device fragmentation, not all MMAPI implementations support reference check for events. Instead, you have to use `equals()` for comparison from Listing 4-2, instead of the improved code from Listing 4-4. The trick is to test your target device(s) for what is supported and optimize accordingly.

Summary

The several different states that a `Player` instance goes through in processing and playing media data allows developers to gain control over these states, provide feedback, and process events at these stages. These states are UNREALIZED, REALIZED, PREFETCHED, STARTED, and CLOSED, and the transitions between them are well defined and accessible.

In this chapter, you learned the background behind these states, the how and why of the transitions that take place between them, and how to respond to the various events generated during these transitions. You learned to create an event handling class and also how to listen to custom events.

The next chapter will introduce you to accessing media data over the network using MMAPI, a task that must be handled efficiently and cleanly for responsive multimedia MIDlets.