Java Media Players

Version .96, May 27, 1997

Java Media Framework is being developed by
Sun Microsystems, Inc., Silicon Graphics Inc., and Intel Corporation.

Contents

Preface

The Java Media Framework (JMF) is an application programming interface (API) for incorporating media data types into Java applications and applets. It is specifically designed to take advantage of of Java platform features. The JMF provides APIs for media players, media capture, and conferencing. This document describes the Java Media Player APIs and how they can be used to present time-based media such as audio and video.

Java Media Players

The 1.0 specification for Java Media Players addresses media display, and the concerns of the application builder in that domain, with an eye towards the other application domains and other levels of developer. There are two parts to this release: a user guide entitled Java Media Players, and the accompanying API documentation.

Status of Future Releases

Javasoft and its partners are developing new capabilities and features that will appear in a future release of the JMF specification. The features that we are considering for future releases include:

- *Incomplete Players* – A JMF Player is self-contained, and provides no access to media data. Additional interfaces that provide access to media data and allow selection of rendering components are in development and intended for a future release.
- *Rendering Interfaces* – Rendering interfaces for specific audio and video formats have to be finalized. Additional interfaces for a video or audio

renderer have not yet been fully developed or documented.

- *Capture Semantics* – The JMF Player architecture does not yet provide for media capture of the kind required for authoring or conferencing applications.
- *Data Definitions* – Audio and video formats have yet to be finalized. An overall structure for data manipulation and format negotiation among generic formats has been defined, but the specific interfaces for audio and video data have not yet been defined.
- *CODEC Architecture* – An architecture for CODECs needs to be defined in order to provide a common API for using CODECs and a mechanism to allow the installation of additional CODECs into the system.

Contact Information

JavaSoft

To obtain information about the Java Media Framework, see the web site at:

HTTP://www.javasoft.com/products/java-media/mediaplayer

Silicon Graphics

To obtain information about Java Media Framework implementations for Silicon Graphics hardware send mail to:

cosmo-motion-info@sgi.com

Intel Corporation

To obtain information about Java Media Framework Implementations for Intel hardware, see the web site at:

HTTP://developer.intel.com/ial/jmedia

Java Media Players

Sun Microsystems, Inc.
Silicon Graphics Inc.
Intel Corporation

The Java Media Framework (JMF) provides APIs for media players, media capture, and conferencing. This document describes the Java Media Player APIs and how they can be used to present time-based media such as audio and video.

The JMF API covers a wide range of applications, and addresses the requirements of developers working at different levels. Interest in JMF can be divided roughly across three application domains and three categories of developer. We have identified the following application domains:

- *Media Display* – Encompasses local and network playback of multimedia data within an application or applet. The focus of JMF in this area is to support the delivery of statically stored, synchronized media data, and to allow integration with the underlying platform's native environment and Java's core packages, such as `java.awt`. This area also encompasses streaming protocols such as RTP. The 1.0 Java Media Player APIs support media display.
- *Media Capture* – This domain imposes additional requirements above and beyond those of media display. Support for media capture implies the ability to record, save, and transfer data through local capture devices, such as microphones and cameras. A future release of JMF will define classes to represent renderers, capture devices, capture objects, and media data.

- *Media Conferencing* – This application domain encompasses conferencing, computer telephony integration, and simple authoring applications for media data. A future release of JMF will address media conferencing.

The Java Media Player APIs support three levels of use:

- *Client level*–a client programmer can create and control a Java Media Player for any standard media type by using a few simple method calls.
- *Enhancement level*–a programmer can modify an existing player to add new functionality by replacing selected player parts, such as renderers. By making it possible to replace individual player parts, JMF provides a way to add functionality to a player without building one from scratch.
- *Design level*–a programmer can add new players to support additional media formats. New players are created by extending the JMF, allowing new players to be used side-by-side with existing players.

By providing three distinct programming levels, JMF makes it easy to incorporate media in client applications and applets, while maintaining the flexibility needed for more sophisticated applications and platform customization. This document is intended primarily for client programmers.

1.0 Overview

JMF provides a platform-neutral framework for building media players. It is designed to support many media content types, including MPEG-1, MPEG-2, QuickTime, AVI, WAV, AU, and MIDI. Using Java Media Players, a programmer can synchronize and present time-based media from diverse sources.

Existing players on desktop computers are heavily dependent on native code for computationally intensive tasks like decompression and rendering. Some Java Media Players require native code to support the features of a specific hardware device or operating system, or to maintain compatibility with existing multimedia standards. Since Java accommodates both Java bytecode and native methods, developers and users can choose among different player implementations that use both Java and native objects.

1.1 Media Sources

A Java Media player encapsulates its media source; it is constructed to present a particular media source, identified by a universal resource locator (URL), and cannot be reused to present other media streams.

Java Media Players can present media data obtained from a variety of sources, such as local or network files and live broadcasts. JMF categorizes media sources according to whether or not the client is guaranteed to receive all of the data:

- *Pull Data Source*—the client is guaranteed to receive every packet from the data source, such as a local or network file. Established protocols for this type of data include Hypertext Transfer Protocol (HTTP) and FILE.
- *Push Data Source*—the data from the media source is not guaranteed to be delivered reliably and clients are expected to recover from gaps in the data. Push data sources include broadcast media, multicast media, and video-on-demand (VOD). For broadcast data, one protocol is the Real-time Transport Protocol (RTP), under development by the Internet Engineering Task Force (IETF). The MediaBase protocol developed by SGI is a protocol used for VOD.

The degree of control that a client program can extend to the user depends on the type of media source being presented. For example, a media source such as a file can be repositioned, allowing the user to replay the media stream or seek to a new location in the stream. A broadcast media source, however, is under server control and cannot be repositioned. Similarly, a VOD source might support limited user control, but probably not the degree of control available with a pull data source.

1.2 Players

A player is a software machine that processes a stream of data over time, reading data from a media source and rendering it at a precise time. A Java Media Player implements the methods defined by four interfaces:

- `Clock` defines the basic timing and synchronization operations that a player uses to control the presentation of media data.
- `Controller` extends `Clock` to provide methods for obtaining system resources and preloading data and a listening mechanism that allows you to receive notification of media events.
- `Duration` provides a way to determine the duration of the media being played.
- `Player` extends `Controller` and `Clock` to support standardized user controls. `Player` also relaxes some of the operational restrictions imposed by `Clock`.

Players share a common model for timekeeping and synchronization. A player's *media time* represents the current position in the media stream. Each player has a *time base* that defines the flow of time for the player. When a player is started, its media time is mapped to its time-base time. To be synchronized, players must use the same time base.

A player's user interface can include both a visual component and a control-panel component. You can implement a custom user-interface for a player or use the player's default control-panel component.

A player must perform a number of operations before it is capable of presenting media. Because some of these operations can be time consuming, JMF allows you to control when they occur by defining the operational states of a player and providing a control mechanism for moving the player between those states.

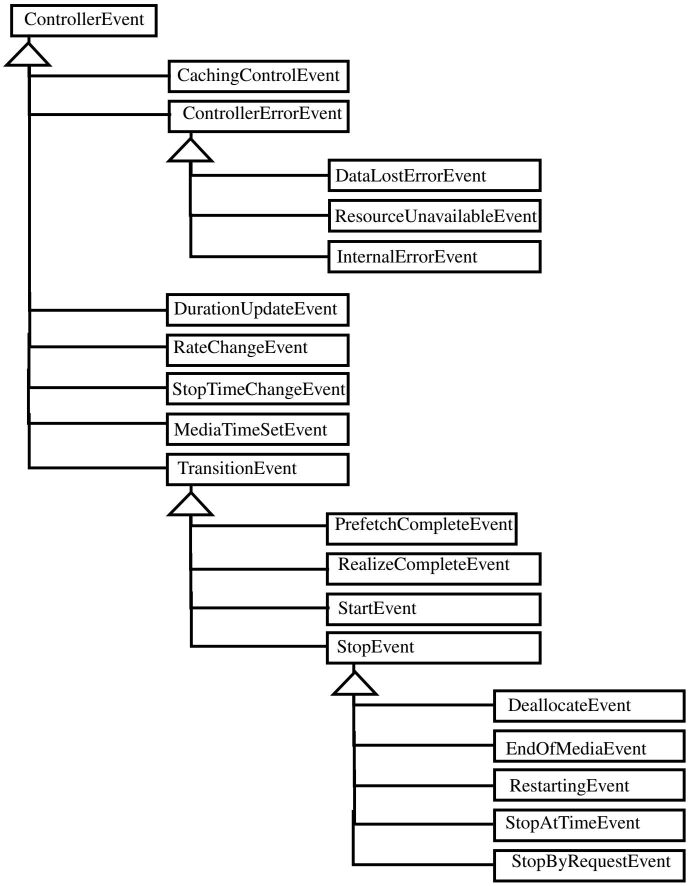
1.3 Media Events

The JMF event reporting mechanism allows your program to respond to media-driven error conditions, such as out-of-data or resource unavailable conditions. The event system also provides an essential notification protocol; when your program calls an asynchronous method on a player, it can only be sure that the operation is complete by listening for the appropriate event.

Two type of JMF objects post events: `GainControl` objects and `Controller` objects.

A `GainControl` object posts only one type of event, `GainChangeEvent`. To respond to gain changes, you implement the `GainChangeListener` interface.

A `Controller` can post a variety of events that are derived from `Controller-Event`. To receive events from a `Controller` such as a `Player`, you implement the `ControllerListener` interface. The following figure shows the events that can be posted by a `Controller`.



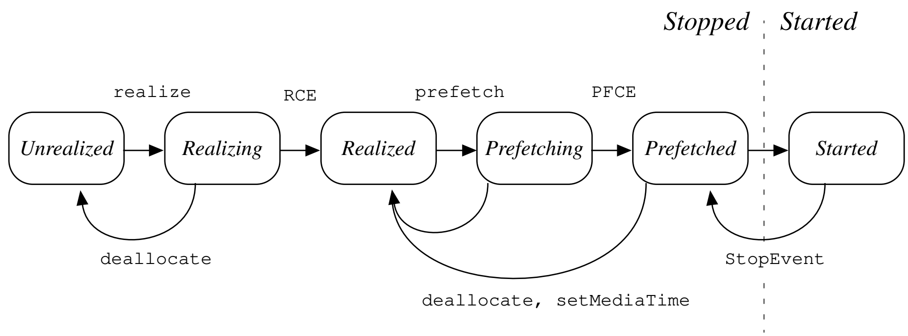
Controller events fall into three categories: change notifications, error events, and transition events:

- Change notification events such as `RateChangeEvent` and `DurationUpdateEvent` indicate that some attribute of the player has changed, often in response to a method call. For example, the player posts a `RateChangeEvent` when its rate is changed with a `setRate` call.
- `TransitionEvents` allow your program to respond to changes in a player's state. A player posts transition events whenever it moves from one state to another. (See Section 1.4 for more information about player states.)
- `ControllerErrorEvents` are posted by a player when it has encountered a problem and cannot recover. When a player posts a `ControllerErrorEvent`, it is no longer usable. You can listen for `ControllerErrorEvent` so that your

program can respond to player malfunctions, minimizing the impact on the user.

1.4 Player States

A Java Media Player can be in one of six states. The `Clock` interface defines the two primary states: *Stopped* and *Started*. `Controller` breaks the stopped state down into five standby states: *Unrealized*, *Realizing*, *Realized*, *Prefetching*, and *Prefetched*.



RCE = RealizeCompleteEvent; PFCE = PrefetchCompleteEvent

In normal operation, a player steps through each state until it reaches the *Started* state:

- A player in the *Unrealized* state has been instantiated, but does not yet know anything about its media other than its URL. When a media player is first created, it is *Unrealized*.
- When `realize` is called, a player moves from the *Unrealized* state into the *Realizing* state. A *Realizing* player is in the process of determining its resource requirements. During realization, a player acquires the resources that it only needs to acquire once. These might include rendering resources other than exclusive-use resources. (Exclusive-use resources are limited resources such as particular hardware devices that can only be used by one player at a time; such resources are acquired during *Prefetching*.)
- When a player finishes realizing, it moves into the *Realized* state. A *Realized* player knows what resources it needs and something about the media it is to present. Because a *Realized* player knows how to render itself, it can provide its visual components and controls. Its connections to other objects in the

system are in place, but it does not own any resources that would prevent another player from starting.

- When `prefetch` is called, a player moves from the *Realized* state into the *Prefetching* state. A *Prefetching* player is preparing to present its media. During this phase, the player can preload its media data, obtain exclusive-use resources, and anything else that it must do every time it prepares to play. Prefetching might have to recur if a player's media presentation is repositioned, or if a change in the player's rate requires that additional buffers be acquired or alternate processing take place.
- When a player finishes prefetching, it moves into the *Prefetched* state. A *Prefetched* player is ready to be started; it is as ready to play as it can be without actually being started.
- Calling `start` puts a player into the *Started* state. A *Started* player's time-base time and media time have been mapped and its clock is running, though the player might be waiting for a particular time to begin presenting its media data.

A player posts `TransitionEvents` as it moves from one state to another. The `controllerListener` interface provides a way for your program to determine what state a player is in and to respond appropriately.

This mechanism allows you to manage player latency by controlling when a player begins realizing and prefetching. It also provides a way that you can ensure that the player is in an appropriate state before calling methods on the player.

1.4.1 Methods Available in Each Player State

To prevent race conditions and deadlocks, not all methods can be called on a player in every state. The following table identifies the restrictions imposed by the JMF. If you call a method that is illegal in a player's current state, the player throws an error or exception.

Table 1: Restrictions on Player Methods

Method	Unrealized Player	Realized Player	Prefetched Player	Started Player
getStartLatency	NotRealizedError	legal	legal	legal
getTimeBase	NotRealizedError	legal	legal	legal
setMediaTime	NotRealizedError	legal	legal	legal

Method	Unrealized Player	Realized Player	Prefetched Player	Started Player
setRate	NotRealizedError	legal	legal	legal
getVisualComponent	NotRealizedError	legal	legal	legal
getControlPanelComponent	NotRealizedError	legal	legal	legal
getGainControl	NotRealizedError	legal	legal	legal
setStopTime	NotRealizedError	legal	legal	StopTimeSetError if previously set
syncStart	NotPrefetchedError	NotPrefetchedError	legal	ClockStartedError
setTimeBase	NotRealizedError	legal	legal	ClockStartedError
deallocate	legal	legal	legal	ClockStartedError
addController	NotRealizedError	legal	legal	ClockStartedError
removeController	NotRealizedError	legal	legal	ClockStartedError
mapToTimeBase	ClockStoppedException	ClockStoppedException	ClockStoppedException	legal

1.5 Calling JMF Methods

JMF uses the following convention for errors and exceptions:

- Java Media Errors are thrown when a program calls a method that is illegal in the current context. Errors are thrown in situations where you have control over the context and the requested operation could result in a race condition or deadlock. For example, it is an error to call certain methods on a *Started* player. It is your responsibility to ensure that a player is stopped before using these methods.
- Java Media Exceptions are thrown when a program calls a method that cannot be completed or is not applicable in the current context. Exceptions are thrown in situations where you do not necessarily have control over the current context. For example, an exception is thrown if you attempt to synchronize two players with incompatible time bases. This is not an error because you could not determine ahead of time that the time bases were incompatible. Similarly, if you call a method that is only applicable for a *Started* player and the player is stopped, an exception is thrown. Even if you just started the player, it might have already stopped in response to other conditions, such as end of media.

Some JMF methods return values that indicate the results of the method call. In some instances, these results might not be what you anticipated when you called the method; by checking the return value, you can determine what actually happened. For example, the return value might indicate:

- What value was actually set. For example, not all players can present media data at five times the normal rate. If you call `setRate(5.0)`, the player will set its rate as close as it can to 5.0 and return the rate it actually set. That rate might be 5.0, or it might be 1.0; you need to check the return value to find out.
- That the operation could not be completed. For example, when you call `createPlayer`, the method returns `null` if the requested player could not be created.
- That the information you requested is not currently available. For example, a player might not know its duration until it has played its media stream once. If you call `getDuration` on such a player before it has played, `getDuration` returns `DURATION_UNKNOWN`. If you call `getDuration` again after the player has played, it might be able to return the actual duration of the media stream.

2.0 Example: Creating an Applet to Play a Media File

The sample program `PlayerApplet` demonstrates how to create a Java Media Player and present an MPEG movie from within a Java applet. This is a general example that could easily be adapted to present other types of media streams.

The player's visual presentation and its controls are displayed within the applet's presentation space in the browser window. If you create a player in a Java application, you are responsible for creating the window to display the player's components.

Note: While `PlayerApplet` illustrates the basic usage of a Java Media Player, it does not perform the error handling necessary in a real applet or application. For a more complete sample suitable for use as a template, see "Appendix A: Java Media Applet" on page 31.

2.1 Overview of PlayerApplet

The APPLET tag is used to invoke `PlayerApplet` in an HTML file. The WIDTH and HEIGHT fields of the HTML APPLET tag determine the dimensions of the applet's presentation space in the browser window. The PARAM tag identifies the media file to be played. For example, `PlayerApplet` could be invoked with:

```
<APPLET CODE=ExampleMedia.PlayerApplet
WIDTH=320 HEIGHT=300>
<PARAM NAME=FILE VALUE="Astrnmy.mpg">
</APPLET>
```

When an user opens a web page containing `PlayerApplet`, the applet loads automatically and runs in the specified presentation space, which contains the player's visual component and default controls. The player starts and plays the MPEG movie once. The user can use the default player controls to stop, restart, or replay the movie. If the page containing the applet is closed while the player is playing the movie, the player automatically stops and frees the resources it was using.

To accomplish this, `PlayerApplet` extends `Applet` and implements the `ControllerListener` interface, defining four methods:

- `init`—creates a player for the file that was passed in through the PARAM tag and registers `PlayerApplet` as a controller listener so that it can observe media events posted by the player. (`PlayerApplet's controllerUpdate` method is called whenever the player posts an event.)
- `start`—starts the player when `PlayerApplet` is started.
- `stop`—stops and deallocates the player when the `PlayerApplet` is stopped.
- `controllerUpdate`—responds to player events to display the player's components.

2.2 PlayerApplet Code Listing

```
PlayerApplet.java:
package ExampleMedia

import java.applet.*;
import java.awt.*;
import java.net.*;
import java.media.*;

public class PlayerApplet extends Applet implements
ControllerListener {
    Player player = null;
    public void init() {
          setLayout(new BorderLayout());
          String mediaFile = getParameter("FILE");
          try {
               URL mediaURL = new URL(getDocumentBase(),
                                       mediaFile);
               player = Manager.createPlayer(mediaURL);
               player.addControllerListener(this);
          } catch (Exception e) {
               System.err.println("Got exception "+e);
        }
    }
    public void start() {
          player.start();
    }
    public void stop() {
          player.stop();
          player.deallocate();
    }
    public synchronized void controllerUpdate(ControllerEvent
                                                  event) {
          if (event instanceof RealizeCompleteEvent) {
              Component comp;
          if ((comp = player.getVisualComponent()) != null)
              add ("Center", comp);
        if ((comp = player.getControlPanelComponent()) != null)
                add ("South", comp);
          validate();
       }
    }
}
```

2.3 Initializing the Applet

When a Java applet starts, its `init` method is invoked automatically. You override `init` to prepare your applet to be started. `PlayerApplet` performs four tasks in `init`:

1. Retrieves the applet's FILE parameter.
2. Uses the FILE parameter to locate the media file and build a `URL` object that describes that media file.
3. Creates a player for the media file by calling `Manager.createPlayer`.
4. Registers the applet as a controller listener with the new player by calling `addControllerListener`. Registering as a listener causes `PlayerApplet's` `controllerUpdate` method to be called automatically whenever the player posts a media event. The player posts media events whenever its state changes. This mechanism allows you to control the player's transitions between states and ensure that the player is in a state in which it can process your requests. (For more information, see "Player States" on page 6.)

```
public void init() {
     setLayout(new BorderLayout());
     // 1. Get the FILE parameter.
     String mediaFile = getParameter("FILE");
     try {
          // 2. Create a URL from the FILE parameter. The URL
class is defined in java.net.
         URL mediaURL = new URL(getDocumentBase(), mediaFile);
          // 3. Create a player with the URL object.
          player = Manager.createPlayer(mediaURL);
        // 4. Add PlayerApplet as a listener on the new player.
          player.addControllerListener(this);
     } catch (Exception e) {
          System.err.println("Got exception "+e);
          }
    }
```

2.4 Controlling the Player

The `Applet` class defines `start` and `stop` methods that are called automatically when the page containing the applet is opened and closed. You override these methods to define what happens each time your applet starts and stops.

`PlayerApplet` implements `start` to start the player whenever the applet is started:

```
public void start() {
     player.start();
}
```

Similarly, `PlayerApplet` overrides `stop` to stop and deallocate the player:

```
public void stop() {
     player.stop();
     player.deallocate();
}
```

Deallocating the player releases any resources that would prevent another player from being started. For example, if the player uses a hardware device to present its media, `deallocate` frees that device so that other players can use it.

2.5 Responding to Media Events

`PlayerApplet` registers itself as a `ControllerListener` in its `init` method so that it receives media events from the player. To respond to these events, `PlayerApplet` implements the `controllerUpdate` method, which is called automatically when the player posts an event.

`PlayerApplet` responds to one type of event, `RealizeCompleteEvent`. When the player posts a `RealizeCompleteEvent`, `PlayerApplet` displays the player's components:

```
public synchronized void controllerUpdate(ControllerEvent
event) {
     if (event instanceof RealizeCompleteEvent) {
          Component comp;
          if ((comp = player.getVisualComponent()) != null)
```

```
                add ("Center", comp);
        if ((comp = player.getControlPanelComponent()) != null)
                add ("South", comp);
          validate();
    }
```

A player's user-interface components cannot be displayed until the player is realized; an unrealized player doesn't know enough about its media stream to provide access to its user-interface components. `PlayerApplet` waits for the player to post a `RealizeCompleteEvent` and then displays the player's visual component and default control panel by adding them to the applet container. Calling `validate` triggers the layout manager to update the display to include the new components.

3.0 Creating and Displaying a Player

You create a player indirectly through the media `Manager`. To display the player, you get the player's components and add them to the applet's presentation space or application window.

3.1 Creating a Player

When you need a new player, you request it from the `Manager` by calling `createPlayer`. The `Manager` uses the media URL that you specify to create an appropriate player.

This mechanism allows new players to be integrated seamlessly. From the client perspective, a new player is always created the same way, even though the player might actually be constructed from interchangeable parts or dynamically loaded at runtime.

3.2 Displaying a Player and Player Controls

JMF specifies the timing and rendering model for displaying a media stream, but a player's interface components are actually displayed using `java.awt`, Java's core package for screen display. A player can have two types of components, its visual component and its control components.

3.2.1 Displaying a Player's Visual Component

The component in which a player displays its media data is called its visual component. Even an audio player might be associated with a visual component, such as a speaker icon or an animated character.

To display a player's visual component, you:

1. Get the component by calling `getVisualComponent`.
2. Add it to the applet's presentation space or application window.

You can access the player's display properties, such as its *x* and *y* coordinates, through its visual component. The layout of the player components is controlled through the layout manager.

3.2.2 Displaying a Player's Controls

A player is often associated with a control panel that allows the user to control the media presentation. For example, a player might be associated with a set of buttons to start, stop, and pause the media stream, and with a slider control to adjust the volume.

Every Java Media Player provides a default control panel. To display a player's default control panel, you get it by calling `getControlPanelComponent` and add it to the applet's presentation space or application window. If you prefer to define a custom user-interface, you have access to the interfaces through which the standard control panel is implemented.

A player's control-panel component is often a client of two different classes of objects. For example, to start and stop the player or set its media time, the control panel calls the player directly. But many players have other properties that can be managed by the user. For example, a video player might allow the user to adjust brightness and contrast, which are not managed through the `Player` interface.To handle these types of controls, JMF defines the `Control` interface.

A media player can have any number of `Control` objects that define control behaviors and have corresponding user interface components. You can get these controls by calling `getControls` on the player. For example, to determine if a player supports the `CachingControl` interface and get the `CachingControl` if it does, you can call `getControls`:

```
    Control[] controls = player.getControls ();
```

```
for (int i = 0; i < controls.length; i++) {
     if (controls[i] instanceof CachingControl) {
         cachingControl = (CachingControl) controls[i];
     }
}
```

The controls that are supported by a particular player depends on the player implementation.

3.2.3 Displaying a Gain Control Component

`GainControl` extends the `Control` interface to provide a standard API for adjusting audio gain. To get this control, you must call `getPlayerGainControl`; `getControls` does not return a player's `GainControl`. `GainControl` provides methods for adjusting the audio volume, such as `setLevel` and `setMute`. Like other controls, the `GainControl` is associated with a GUI component that can be added to an applet's presentation space or an application window

3.2.4 Displaying a Player's Download Progress

Downloading media data can be a time consuming process. In cases where the user must wait while data is downloaded, a progress bar is often displayed to reassure the user that the download is proceeding and to give some indication of how long the process will take. The `CachingControl` interface is a special type of control supported by players that can report their download progress. You can use this interface to display a download progress bar to the user.

You can call `getControls` to determine whether or not a player supports the `CachingControl` interface. If it does, the player will post a `CachingControlEvent` whenever the progress bar needs to be updated. If you implement your own progress bar component, you can listen for this event and update the download progress whenever `CachingControlEvent` is posted.

A `CachingControl` also provides a default progress bar component that is automatically updated as the download progresses. To use the default progress bar in an applet:

1. Implement the `ControllerListener` interface and listen for `CachingControlEvents` in `controllerUpdate`.
2. The first time you receive a `CachingControlEvent`:

   a. Call `getCachingControl` on the event to get the caching control.

b. Call `getProgressBar` on the `CachingControl` to get the default progress bar component.

c. Add the progress bar component to the applet's presentation space.

3. Each time you receive a `CachingControlEvent`, check to see if the download is complete. When `getContentProgress` returns the same value as `getContentLength`, remove the progress bar.

4.0 Controlling Media Players

The `Clock` and `Player` interfaces define the methods for starting and stopping a player.

4.1 Starting a Player

You typically start a player by calling `Player.start`.The `start` method tells the player to begin presenting media data as soon as possible. If necessary, `start` prepares the player to start by performing the realize and prefetch operations. If `start` is called on a *Started* player, the only effect is that a `StartEvent` is posted in acknowledgment of the method call.

Clock defines a `syncStart` method that can be used for synchronization. See "Synchronizing Players" on page 25 for more information.

To start a player at a specific point in a media stream:

1. Specify the point in the media stream at which you want to start by calling `setMediaTime`.
2. Call `start` on the player.

4.2 Stopping a Player

There are three situations in which a player will stop:

- When the `stop` method is called on the player.
- When the player has reached the specified stop time.
- When the player has run out of media data.

When a non-broadcast player is stopped, its media time is frozen. If the stopped player is subsequently restarted, media time resumes from the stop time. When you stop a broadcast player, however, only the receipt of the media data is stopped, the data continues to be broadcast. When you restart the broadcast player, the playback will resume wherever the broadcast is at that point in time.

You use the `stop` method to stop a player immediately. If you call `stop` on a *Stopped* player, the only effect is that a `StopByRequestEvent` is posted in acknowledgment of the method call.

4.2.1 Stopping a Player at a Specified Time

You can call `setStopTime` to indicate when a player should stop. The player stops when its media time passes the specified stop time. If the player's rate is positive, the player stops when the media time becomes greater or equal to the stop time. If the player's rate is negative, the player stops when the media time becomes less than or equal to the stop time. The player stops immediately if its current media time is already beyond the specified stop time.

For example, assume that media time is 5.0 and the `setStopTime` is called to set the stop time to 6.0. If the player's rate is positive, media time is increasing and the player will stop when the media time becomes greater than 6.0. However, if the player's rate is negative, it is playing in reverse and the player will stop immediately because the media time is already beyond the stop time. (For more information about player rates, see "Setting a Player's Rate" on page 24.)

You can always call `setStopTime` on a stopped player. However, you can only set the stop time on a *Started* player if the stop time is not currently set. If the player already has a stop time, `setStopTime` throws an error.

You can call `getStopTime` to get the currently scheduled stop time. If the clock has no scheduled stop time, `getStopTime` returns `Long.MAX_VALUE`. To remove the stop time so that the player continues until it reaches end-of-media, call `setStopTime(Long.MAX_VALUE)`.

5.0 Managing Player States

The transitions between states are controlled with five methods:

- `realize`
- `prefetch`
- `start`

- `deallocate`
- `stop`

By controlling when these methods are called, you can manage the state of a player. For example, you might want to minimize start-latency by preparing the player to start before you actually start it.

You can implement the `ControllerListener` interface to manage these control methods in response to changes in the player's state. Listening for a player's state transitions is also important in other cases. For example, you cannot get a player's components until the player has been *Realized*. By listening for a `RealizeCompleteEvent` you can get the components as soon as the player is *Realized*.

5.1 Preparing a Player to Start

Most media players cannot be started instantly. Before the player can start, certain hardware and software conditions must be met. For example, if the player has never been started, it might be necessary to allocate buffers in memory to store the media data. Or if the media data resides on a network device, the player might have to establish a network connection before it can download the data. Even if the player has been started before, the buffers might contain data that is not valid for the current media position.

5.1.1 Realizing and Prefetching the Player

JMF breaks the process of preparing a player to start into two phases, *Realizing* and *Prefetching*. Realizing and prefetching a player before you start it minimizes the time it takes the player to begin presenting media when `start` is called and helps create a highly-responsive interactive experience for the user. Implementing the `ControllerListener` interface allows you to control when these operations occur.

You call `realize` to move the player into the *Realizing* state and begin the realization process. You call `prefetch` to move the player into the *Prefetching* state and initiate the prefetching process. "Player States" on page 6 describes the operations that a player performs in each of these states. You cannot synchronously move the player directly into the *Realized* or *Prefetched* state. When it completes the operation, the player posts a `RealizeCompleteEvent` or `PrefetchCompleteEvent`.

A player in the *Prefetched* state is prepared to start and its start-up latency cannot be further reduced. However, setting the media time through `setMediaTime` might return the player to the *Realized* state, increasing its start-up latency.

Keep in mind that a *Prefetched* player ties up system resources. Because some resources, such as sound cards, might only be usable by only one program at a time, this might prevent other players from starting.

5.1.2 Determining a Player's Start-up Latency

To determine how much time is required to start a player, you can call `getStart-Latency`. For players that have a variable start latency, the return value of `get-StartLatency` represents the maximum possible start latency.

The start-up latency reported by `getStartLatency` might differ depending on the player's current state. For example, after a `prefetch` operation, the value returned by `getStartLatency` is typically smaller.

A player is not guaranteed to start at a specified time unless you have established that the start time is feasible by calling `getStartLatency`. For some media types, `getStartLatency` might be unable to return a useful value.

5.2 Starting and Stopping a Player

Calling `start` moves a player into the *Started* state. As soon as `start` is called, methods that are only legal for stopped players cannot be called until the player has been stopped.

If `start` is called and the player has not been prefetched, `start` performs the realize and prefetch operations as needed to move the player into the *Prefetched* state. The player posts transition events as it moves through each state.

When `stop` is called on a player, the player is considered to be stopped immediately; `stop` is synchronous. However, a player can also stop asynchronously when it reaches either the end of its media stream or the stop time previously set with `setStopTime`.

When a player stops, it posts a `StopEvent`. To determine why the player stopped, you must listen for the specific stop events: `DeallocateEvent`, `EndOfMediaE-vent`, `RestartingEvent`, `StopAtTimeEvent`, or `StopByRequestEvent`.

5.3 Releasing Player Resources

The `deallocate` method tells a player to release any exclusive resources and minimize its use of non-exclusive resources. Although buffering and memory management requirements for players are not specified, most Java Media Players allocate buffers that are large by the standards of Java objects. A well-implemented player releases as much internal memory as possible when `deallocate` is called.

The `deallocate` method can only be called on a *Stopped* player. To avoid `ClockStartedErrors`, you should call `stop` before you call `deallocate`. Calling `deallocate` on a player in the *Prefetching* or *Prefetched* state returns it to the *Realized* state. If `deallocate` is called while the player is realizing, the player posts a `deallocateEvent` and returns to the *Unrealized* state. (Once a player has been realized, it can never return to the *Unrealized* state.)

You generally call `deallocate` when the player is not being used. For example, an applet should call `deallocate` as part of its `stop` method. By calling `deallocate`, the program can maintain references to the player, while freeing other resources for use by the system as a whole. (JMF does not prevent a *Realized* player that has formerly been *Prefetched* or *Started* from maintaining information that would allow it to be started up more quickly in the future.)

5.4 Implementing the ControllerListener Interface

`ControllerListener` is an asynchronous interface for handling events generated by `Controller` objects. By implementing the `ControllerListener` interface and using the player control methods, you can manage the timing of potentially time-consuming player operations such as prefetching.

To implement the `ControllerListener` interface, you need to:

1. Register your class as a listener by calling `addControllerListener`.
2. Implement the `controllerUpdate` method.

When a controller posts an event, it calls `controllerUpdate` on each registered listener. Typically, `controllerUpdate` is implemented as a series of `if-else` statements of the form:

```
if(instanceof EventType){
...
```

```
} else if (instanceof OtherEventType){
...
}
```

This filters out the events that you are not interested in. If you have registered as a listener with multiple players, you also need to determine which player generated the event. Controller events come “stamped” with a reference to their source that you can access by calling `getSource`.

You should also check the target state by calling `getTargetState` before calling any of the methods that are restricted to *Stopped* players. If `start` has been called, the player is considered to be in the *Started* state, though it might be posting transition events as it prepares the player to present media.

Some classes of controller event are stamped with additional state information. For example, the `StartEvent` and `StopEvent` classes each define a method that allows you to retrieve the media time at which the event occurred.

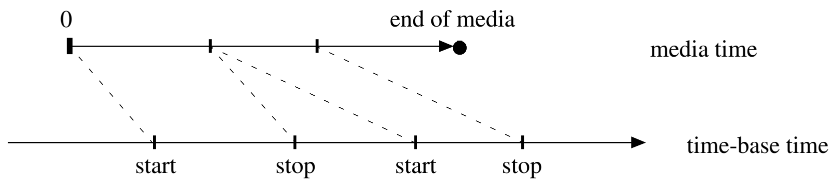
6.0 Managing Timing and Synchronization

In many cases, instead of playing a single media stream from beginning to end, you want to play a portion of the stream or synchronize the playback of a stream. The JMF `TimeBase` and `Clock` interfaces define the mechanism for managing the timing and synchronization of media playback.

A *time base* represents the flow of time. A *time-base time* cannot be transformed or reset. A Java Media Player uses its time base to keep time in the same way that a quartz watch uses a crystal that vibrates at a known frequency to keep time. The system maintains a master time-base that measures time in nanoseconds from a specified base time, such as January 1, 1970. The system time-base is driven by the system clock and is accessible through the `Manager.getSystemTimeBase` method.

A player’s *media time* represents a point in time within the stream that the player is presenting. The media time can be started, stopped, and reset, much like a stopwatch.

A clock defines the mapping between a time base and the media time.



A Java Media Player can answer several useful timing queries about the media source it is presenting. Of course, timing information is subject to the physical characteristics and limitations of both the media source and of the network device on which it is stored.

6.1 Setting the Media Time

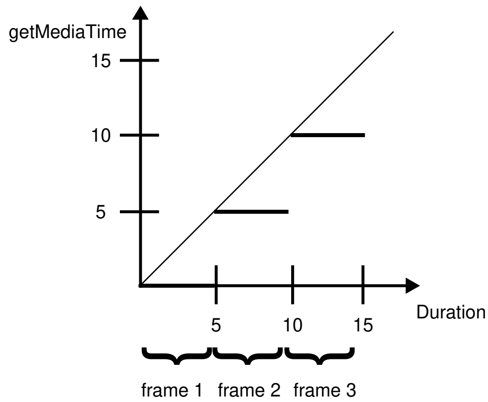
Setting a player's media time is equivalent to setting a read position within a media stream. For a media data source such as a file, the media time is bounded; the maximum media time is defined by the end of the media stream.

To set the media time you call `setMediaTime`, specifying a time in nanoseconds.

6.2 Getting the Current Time

Calling `getMediaTime` returns the player's current media time in nanoseconds. If the player is not presenting media data, this is the point from which media presentation will commence. There is not a one-to-one correspondence between a media time and a particular frame. Each frame is presented for a certain period of time, and the media time continues to advance during this period.

For example, imagine you have a slide show player that displays each slide for 5 seconds—the player essentially has a frame rate of 0.2 frames per second.



If you start the player at time 0.0, while the first “frame” is displayed, the media time advances from 0.0 to 5.0. If you start at time 2.0, the first frame is displayed for 3 seconds, until time 5.0 is reached.

Media time is measured in nanoseconds because different types of media with varying frame rates can be presented together.

You can get a player’s current time-base time by getting the player’s time base and calling `getRefTime`:

```
myCurrentTBTime = player1.getTimeBase().getRefTime();
```

When a player is running, you can get the time-base time that corresponds to a particular media time by calling `mapToTimeBase`.

6.3 Setting a Player’s Rate

The player’s rate determines how media time changes with respect to time-base time; it defines how many units a player’s media time advances for every unit of time-base time. The player’s rate can be thought of as a temporal scale factor. For example, a rate of 2.0 indicates that media time passes twice as fast as the time-base time when the player is started.

In theory, a player’s rate could be set to any real number, with negative rates interpreted as playing the media in reverse. However, some media formats have dependencies between frames that make it impossible or impractical to play them in reverse, or at non-standard rates.

When `setRate` is called on a player, the method returns the rate that is actually set, even if it has not changed. Player's are only guaranteed to support a rate of 1.0.

6.4 Getting a Player's Duration

Since your program might need to determine how long a given media stream will run, all players implement the `Duration` interface. This interface comprises a single method, `getDuration`. Duration represents the length of time that a media object would run for, if played at the default rate of 1.0. A media stream's duration is accessible only through the player itself. The value returned by `getDuration` is an absolute value that represents time in nanoseconds.

If the duration cannot be determined, `getDuration` returns `DURATION_UNKNOWN`. This can happen if the player has not yet reached a state where the duration of the media source is available, or if the media source does not have a defined duration, as in the case of a live broadcast.

6.5 Synchronizing Players

To synchronize the playback of multiple media streams, you can synchronize the players by associating them with the same time base. To do this, you use the `getTimeBase` and `setTimeBase` methods defined by the `Clock` interface. For example, you could synchronize `player1` with `player2` by setting `player1` to use `player2's` time base:

```
player1.setTimeBase(player2.getTimeBase());
```

When you synchronize players by associating them with the same time base, you must still manage the control of each player individually. Because managing synchronized players in this way can be complicated, JMF provides a mechanism that allows a `Player` to assume control over any `Controller`. The player manages the states of the controllers automatically, allowing you to interact with the entire group through a single point of control. For more information, see "Using a Player to Manage and Synchronize other Controllers" on page 27.

In a few situations, you might want to manage the synchronization of multiple players yourself so that you can control the rates or media times independently. If you do this, you must:

- Register as a listener for each synchronized player.
- Determine which player's time base is going to be used to drive the other players and set the time base for the synchronized players. Not all players can

assume a new time base. For example, if one of the players you want to synchronize has a push data source, that player's time base must be used to drive the other players.

- Set the rate for all of the players. If a player cannot support the rate you specify, it returns the rate that was used. (There is no mechanism for querying the rates that a player supports.)
- Synchronize the players' states. (For example, stop all of the players.)
- Synchronize the operation of the players:
  - Set the media time for each player.
  - Prefetch all of the players.
  - Determine the maximum start latency among the synchronized players.
  - Start the players by calling `syncStart` with a time that takes into account the maximum latency.

You must listen for transition events for all of the players and keep track of which ones have posted events. For example, when you prefetch the players, you need to keep track of which ones have posted `PrefetchComplete` events so that you can be sure all of the players are prefetched before calling `syncStart`. Similarly, when you request that the synchronized players stop at a particular time, you need to listen for the stop event posted by each player to determine when all of the players have actually stopped.

In some situations, you need to be careful about responding to events posted by the synchronized players. To be sure of the players' states, you might need to wait at certain stages for all of the synchronized players to reach the same state before continuing.

For example, assume that you are using one player to drive a group of synchronized players. A user interacting with that player sets the media time to 10, starts the player, and then changes the media time to 20. You then:

- Pass along the first `setMediaTime` call to all of the synchronized players
- Call `prefetch` on the players to prepare them to start
- Call `stop` on the players when the second set media time request is received.
- Call `setMediaTime` on the players with the new time.
- Restart the prefetching operation.
- When all of the players have been prefetched, start them by calling `syncStart`, taking into account their start latencies.

In this case, simply listening for `PrefetchComplete` events from all of the players before calling `syncStart` isn't sufficient. You can't tell whether those events were posted in response to the first or second prefetch operation. To avoid this problem, you can block when you call `stop` and wait for all of the players to post stop events before continuing. This guarantees that the next `PrefetchComplete` events you receive are the ones you are really interested in.

7.0 Using a Player to Manage and Synchronize other Controllers

Synchronizing players manually using `syncStart` requires that you carefully manage the states of all of the synchronized players. You must control each one individually, listening for events and calling control methods on them as appropriate. Even with only a few players, this quickly becomes a difficult task. Through the `Player` interface, JMF provides a simpler solution: a `Player` can be used to manage the operation of any `Controller`.

When you interact with a managing `Player`, your instructions are automatically passed along to the managed controllers as appropriate. The managing player takes care of the state management and synchronization for all of the other Controllers.

This mechanism is implemented through the `Player.addController` and `Player.removeController` methods. When you call `addController` on a `Player`, the `Controller` you specify is added to the list of controllers managed by the player. Conversely, when you call `removeController`, the specified `Controller` is removed from the list of managed controllers.

Typically when you need to synchronize players or other controllers, you should use this `addController` mechanism. It is simpler, faster, and less error-prone than attempting to manage synchronized players individually.

When a `Player` assumes control of a `Controller`:

- The `Controller` assumes the `Player's` time-base.
- The `Player's` duration becomes the longer of the controller's duration and its own. If multiple controllers are placed under a player's control, the player's duration is the longest of all of their durations.
- The Player's start latency becomes the longer of the controller's duration and its own. If multiple controllers are placed under a player's control, the player's start latency is the longest of all of their latencies.

A managing `Player` only posts completion events for asynchronous methods after every added `Controller` has posted the event. The managing `Player` reposts other events generated by the managed `Controllers` as appropriate.

7.1 Adding a Controller

You use the `Controller.addController` method to add a `Controller` to the list of controllers managed by a particular `Player`. To be added, a `Controller` must be in the *Realized* state; otherwise, a `NotRealizedError` is thrown. Two players cannot be placed under control of each other.

Once a `Controller` has been added to a `Player`, do not call methods directly on the added `Controller`. To control an added `Controller`, you interact with the managing `Player`.

To have `player2` assume control of `player1`, call:

```
player2.addController(player1)
```

7.2 Managing the Operation of Added Controllers

To control the operation of a group of controllers managed by a particular `Player`, you interact directly with the managing `Player`. Do not call control methods on the managed controllers directly.

For example, to prepare all of the managed `Controllers` to start, call `prefetch` on the managing `Player`. Similarly, when you want to start them, call `start` on the managing `Player`. The managing `Player` makes sure that all of the controllers are *Prefetched*, determines the maximum start latency among the controllers, and calls `syncStart` to start them, specifying a time that takes the maximum start latency into account.

When you call a `Controller` method on the managing `Player`, the `Player` propagates the method call to the managed `Controllers` as appropriate. Before calling a `Controller` method on a managed `Controller`, the `Player` ensures that the `Controller` is in the proper state. The following table shows the `Controller` methods that affect managed `Controllers`.

Function	Stopped Player	Started Player
`setMediaTime`	Invokes `setMediaTime` on all managed Controllers.	Stops all managed Controllers, invokes `setMediaTime`, and restarts Controllers.

`setRate`	Invokes `setRate` on all managed Controllers. Returns the actual rate that was supported by all Controllers and set.	Stops all managed Controllers, invokes `setRate`, and restarts Controllers. Returns the actual rate that was supported by all Controllers and set.
`start`	Ensures all managed Controllers are Prefetched and invokes `syncStart` on each of them, taking into account their start latencies.	Illegal.
`realize`	Invokes `realize` on all managed Controllers.	Illegal.
`prefetch`	Invokes `prefetch` on all managed Controllers.	Illegal.
`stop`	No effect.	Invokes `stop` on all managed Controllers.
`deallocate`	Invokes `deallocate` on all managed Controllers.	Invokes `deallocate` on all managed Controllers.
`setStopTime`	Invokes `setStopTime` on all managed Controllers. (`Player` must be *Realized*.)	Invokes `setStopTime` on all managed Controllers. (Can only be set once on a *Started* `Player`.)
`syncStart`	Invokes `syncStart` on all managed Controllers. (`Player` must be *Prefetched*.)	Illegal.

7.3 Removing a Controller

You use the `Controller.removeController` method to remove a `Controller` from the list of controllers managed by a particular `Player`.

To have `player2` release control of `player1`, call:

```
player2.removeController(player1)
```

8.0 Extending JMF

The JMF architecture allows advanced developers to create and integrate new types of controllers and data sources. For example, you might implement a new `Player` that supports a special media format.

This section introduces the JMF Player Factory architecture and describes how JMF can be extended.

8.1 Understanding the Player Factory Architecture

Appendix A: Java Media Applet

This Java Applet demonstrates proper error checking in a Java Media program. Like `PlayerApplet`, it creates a simple media player with a media event listener.

When this applet is started, it immediately begins to play the media clip. When the end of media is reached, the clip replays from the beginning.

```
import java.applet.Applet;
import java.awt.*;
import java.lang.String;
import java.net.URL;
import java.net.MalformedURLException;
import java.io.IOException;
import java.media.*;

/**
 * This is a Java Applet that demonstrates how to create a simple
 * media player with a media event listener.  It will play the
 * media clip right away and continuously loop.
 *
 * <!-- Sample HTML
 * <applet code=TypicalPlayerApplet width=320 height=300>
 * <param name=file value="Astrnmy.avi">
 * </applet>
 * -->
 */
public class TypicalPlayerApplet extends Applet implements
ControllerListener {

// media player
```

```
Player player = null;
// component in which video is playing
Component visualComponent  = null;
// controls gain, position, start, stop
Component controlComponent = null;
// displays progress during download
Component progressBar      = null;
/**
 * Read the applet file parameter and create the media
 * player.
 */
public void init() {
    setLayout(new BorderLayout());
    // input file name from html param
    String mediaFile = null;
    // URL for our media file
    URL url = null;
   // URL for doc containing applet
    URL codeBase = getDocumentBase();

    // Get the media filename info.
    // The applet tag should contain the path to the
    // source media file, relative to the html page.

    if ((mediaFile = getParameter("FILE")) == null)
        Fatal("Invalid media file parameter");

    try {
      // Create an url from the file name and the url to the
        // document containing this applet.

        if ((url = new URL(codeBase, mediaFile)) == null)
            Fatal("Can't build URL for " + mediaFile);

        // Create an instance of a player for this media
        if ((player = Manager.createPlayer(url)) == null)
            Fatal("Could not create player for "+url);

        // Add ourselves as a listener for player's events
        player.addControllerListener(this);
    } catch (MalformedURLException e) {
        Fatal("Invalid media file URL!");
    } catch(IOException e) {
        Fatal("IO exception creating player for "+url);
    }
```

```
    // This applet assumes that its start() calls
    // player.start().This causes the player to become
    // Realized. Once Realized, the Applet will get
    //  the visual and control panel components and add
    // them to the Applet.  These components are not added
    // during init() because they are long operations that
    // would make us appear unresposive to the user.
}

/**
 * Start media file playback.  This function is called the
 * first time that the Applet runs and every
 * time the user re-enters the page.
 */
public void start() {
    // Call start() to prefetch and start the player.
    if (player != null)
      player.start();
}

/**
 * Stop media file playback and release resources before
 * leaving the page.
 */
public void stop() {
    if (player != null){
        player.stop();
        player.deallocate();
    }
}

/**
 * This controllerUpdate function must be defined in order
 * to implement a ControllerListener interface.  This
 * function will be called whenever there is a media event.
 */
public synchronized void
                 controllerUpdate(ControllerEvent event) {

    // If we're getting messages from a dead player,
    // just leave
    if (player == null)
        return;

    // When the player is Realized, get the visual
    // and control components and add them to the Applet
```

```
if (event instanceof RealizeCompleteEvent) {

    if ((visualComponent =
                 player.getVisualComponent()) != null)
        add("Center", visualComponent);

    if ((controlComponent =
            player.getControlPanelComponent()) != null)
        add("South",controlComponent);

    // force the applet to draw the components
    validate();
}

else if (event instanceof CachingControlEvent) {

    // Put a progress bar up when downloading starts,
    // take it down when downloading ends.

  CachingControlEvent  e = (CachingControlEvent) event;
    CachingControl      cc = e.getCachingControl();
    long cc_progress       = e.getContentProgress();
    long cc_length         = cc.getContentLength();

    // Add the bar if not already there ...
    if (progressBar == null)
        if ((progressBar =
              cc.getProgressBarComponent()) != null) {
            add("North", progressBar);
            validate();
        }

    // Remove bar when finished ownloading
    if (progressBar != null)
        if (cc_progress == cc_length) {
            remove (progressBar);
            progressBar = null;
            validate();
        }
}

else if (event instanceof EndOfMediaEvent) {
    // We've reached the end of the media; rewind and
    // start over
    player.setMediaTime(0);
    player.start();
```

```
        }

        else if (event instanceof ControllerErrorEvent) {
           // Tell TypicalPlayerApplet.start() to call it a day
            player = null;
           Fatal (((ControllerErrorEvent)event).getMessage());
        }
    }

    void Fatal (String s) {
        // Applications will make various choices about what
        // to do here.  We print a message and then exit
        System.err.println("FATAL ERROR: " + s);
        throw new Error(s);  // Invoke the uncaught exception
                           // handler System.exit() is another
                              // choice
    }
}
```