Java Media Framework basics

Eric A. Olson

May 07, 2002

The Java Media Framework (JMF) is an exciting and versatile API that allows Java developers to process media in many different ways. This tutorial provides an overview of some of the major features of JMF, mainly through the use of working examples. Upon completion of this tutorial, you will understand the major players in the JMF architecture. You also will have worked directly with JMF, using live examples and source code that may be extended for more specific purposes.

About this tutorial

What is this tutorial about?

The Java Media Framework (JMF) is an exciting and versatile API that allows Java developers to process media in many different ways. This tutorial provides an overview of some of the major features of JMF, mainly through the use of working examples. Upon completion of this tutorial, you will understand the major players in the JMF architecture. You also will have worked directly with JMF, using live examples and source code that may be extended for more specific purposes.

Topics covered in this tutorial are as follows:

- Tips for downloading and installing JMF
- Major JMF classes and their uses in the JMF architecture
- Playing local media files
- Presenting a graphical user interface (GUI) for media access and manipulation
- Broadcasting media over a network
- Receiving broadcast media over a network

Almost any type of media manipulation or processing is possible through JMF. A comprehensive discussion of all the features JMF has to offer is well beyond the scope of this tutorial. Instead, we'll use three simplified media applications to learn about the framework's building blocks. In this way, the tutorial will prepare you for future study and the implementation of more specific applications.

Should I take this tutorial?

This tutorial will walk you through the basics of working with JMF. To accomplish this, we'll create three separate working example applications. Each of these examples will build upon the previous examples, displaying different aspects of JMF's functionality.

The examples in this tutorial assume that you have used and are already familiar with the Java programming language. In addition to the core Java and JMF classes, we will be working with some of the Java AWT and Swing classes (for presenting a GUI), as well as some Java networking classes (for transmitting media over a network). Some familiarity with both the GUI and networking classes will help you more quickly understand the discussion and examples here, but are not prerequisites of this tutorial.

The example applications we'll work with are as follows:

- **A simple audio media player** (the JMF HelloWorld application): This command-line-driven media player lets you play most audio types by simply specifying the media file name on the command line. The audio media player demo showcases mainly JMF-specific classes.
- **A GUI-driven media player**: We'll use built-in JMF interface components to build the GUI, so little GUI programming experience will be necessary for this exercise. The media viewer demo uses some of the Java AWT and Swing classes to present GUI components to the user.
- **A media broadcasting application**: This application allows a local media file to be transmitted over a network. The application is flexible enough to transmit media only to a specified network node, or to broadcast it to all the nodes within a subnet. This demo uses some of the Java networking APIs to allow media transmission over a network.

As part of the third exercise, we'll modify the GUI-enabled media player to let it receive and play broadcast media.

See Related topics for a listing of articles, tutorials, and other references that will help you learn more about the topics covered in this tutorial.

Installation requirements

To run the examples in this tutorial, you need the following tools and components:

- The **Java 2 platform, Standard Edition**, to compile and run the demo applications
- The **Java Media Framework, version 2.1.1a** or above
- A properly installed and configured **sound card**
- One or more **test machines**
- The demo **source files** in mediaplayer.jar (see Download).

The final demo application shows how JMF can be used over a network. If necessary, the demo can be run on a single machine using the machine as both the transmitter and receiver. To see the full power of using JMF over a network, however, you will need at least two machines that are reachable on the same network.

See Related topics to download the Java 2 platform, complete source files, and other tools essential for the completion of this tutorial.

Downloading the installer

The first step in installing the JMF on your machine is to download the installer through the JMF home page, which also contains links to the JMF source code and the API documentation. See Related topics for a link to download JMF.

Currently, JMF has versions for Windows, Solaris, Linux, and a pure-Java version that works on any machine with a JVM. To increase performance, you should download the version that is specific to your OS. Any code that is written and compiled with an OS-specific version of JMF is portable across other OSs. For example, if you download the Solaris version of JMF and compile a number of classes, those same classes can be used on a Linux machine without any problems.

Alternatively, you may choose to download the pure-Java, or "cross-platform," version of JMF. This version doesn't employ any OS-specific libraries. The cross-platform version is a good choice if your OS doesn't have its own version of JMF installed or if you don't know what OS the target machine will be running.

Installing JMF

After you've downloaded the JMF installer to your desktop, double-click the installer icon.

Most installers have an option to install native libraries in a system directory; for example, the Windows installer will present an option to "Move DLLs to Windows/System directory." For best results, choose this option, because it ensures the OS-specific libraries will be installed correctly.

You should also choose the option to update the system `CLASSPATH` and `PATH` variables during installation. If this option is turned off, remember to include the JMF jar files in the classpath when you compile or run any of the examples in this tutorial.

A simple audio player

Overview

In this section, we'll walk through the first exercise of creating a simple audio player. This example introduces you to the `Manager` class and the `Player` interface, which are two of the major pieces in building most any JMF-based application.

The functional goal of this example is to play a local audio file through a command-line interface. We'll walk through the source code and review what is happening in each line. After completing this section, you will have a demo application on which you can play any audio file type supported by JMF, including MP3, WAV, and AU, among many others.

Refer to the SimpleAudioPlayer.java file in the source code distribution to follow this exercise.

Importing the necessary classes

The first few lines of the `SimpleAudioPlayer` class include the following calls, which import all necessary classes:

```
    import javax.media.*;

    import java.io.File;
    import java.io.IOException;
    import java.net.URL;
    import java.net.MalformedURLException;
```

The `javax.media` package is one of the many packages defined by JMF. `javax.media` is the core package, containing the definitions of the `Manager` class and the `Player` interface, among others. We'll focus on the `Manager` class and the `Player` interface in this section, and deal with some of the other `javax.media` classes in later sections.

In addition to the `import javax.media` statement, the above code fragment includes several import statements that create the input to our media player.

The Player interface

In the next code fragment, the public class `SimpleAudioPlayer` and the `Player` instance variable are defined:

```
    public class SimpleAudioPlayer {

       private Player audioPlayer = null;
```

The term `Player` may sound quite familiar, because it is based on our common use of audio- and video-based media players. In fact, instances of this interface act much like their real-life counterparts. `Player`s expose methods that relate to the functions of a physical media player such as a stereo system or VCR. For example, a JMF media `Player` has the ability to start and stop a stream of media. We will use the start and stop functionality of the `Player` throughout this section.

Creating a Player over a file

JMF makes it quite simple to obtain a `Player` instance for a given media file. The `Manager` class acts as a factory for creating many of the specific interface types exposed in JMF, including the `Player` interface. Therefore, the `Manager` class is responsible for creating our `Player` instance, as shown below:

```
    public SimpleAudioPlayer(URL url) throws IOException,
NoPlayerException,
       CannotRealizeException {
       audioPlayer = Manager.createRealizedPlayer(url);
    }

    public SimpleAudioPlayer(File file) throws IOException,
NoPlayerException,
       CannotRealizeException {
       this(file.toURL());
    }
```

If you're following along with the source for this section, you may have noticed that the `Manager` class contains other methods for creating `Player` instances. We'll explore some of these methods -- such as passing in instances of a `DataSource` or `MediaLocator` -- in a later section.

Player states

JMF defines a number of different states that a `Player` instance may be in. These states are as follows:

- Prefetched
- Prefetching
- Realized
- Realizing
- Started
- Unrealized

Working with states

Because working with media is often quite resource intensive, many of the methods exposed by JMF objects are non-blocking and allow for asynchronous notification of state changes through a series of event listeners. For example, a `Player` must go through both the *Prefetched* and *Realized* states before it may be started. Because these state changes can take some time to complete, a JMF media application can assign one thread to the initial creation of a `Player` instance, then move on to other operations. When the `Player` is ready, it will notify the application of its state changes.

In a simple application such as ours, this type of versatility is not so important. For this reason, the `Manager` class also exposes utility methods for creating *Realized* players. Calling a `createRealizedPlayer()` method causes the calling thread to block until the player reaches the *Realized* state. To call a non-blocking player-creation method, we use one of the `createPlayer()` methods on the `Manager` class. The following line of code creates a *Realized* player, which we need in our example application:

```
        audioPlayer = Manager.createRealizedPlayer(url);
```

Starting and stopping the Player

Setting up a `Player` instance to be started or stopped is as simple as calling the easily recognized methods on the `Player`, as shown here:

```
        public void play() {
           audioPlayer.start();
        }

        public void stop() {
           audioPlayer.stop();
           audioPlayer.close();
        }
```

Calling the `play()` method on the `SimpleAudioPlayer` class simply delegates the call to the `start()` method on the `Player` instance. After calling this method, you should hear the audio file played through the local speakers. Likewise, the `stop()` method delegates to the player to both stop and close the `Player` instance.

Closing the `Player` instance frees any resources that were used for reading or playing the media file. Because this is a simple example, closing the `Player` is an acceptable way of ending a session. In a real application, however, you should carefully consider whether you want to get rid of the `Player` before you close it. Once you've closed the player, you will have to create a new `Player` instance (and wait for it to go through all of its state changes) before you can play your media again.

Creating a SimpleAudioPlayer

Finally, this media player application contains a `main()` method, which lets it be invoked from the command line by passing in the file name. In the `main()` method, we make the following call, which creates the `SimpleAudioPlayer`:

```
        File audioFile = new File(args[0]);
        SimpleAudioPlayer player = new SimpleAudioPlayer(audioFile);
```

The only other thing we have to do before we can play our audio file is to call the `play()` method on the created audio player, as shown here:

```
        player.play();
```

To stop and clean up the audio player, we make the following call, also found in the `main()` method:

```
        player.stop();
```

Compiling and running the SimpleAudioPlayer

Compile the example application by typing `javac SimpleAudioPlayer.java` at a command prompt. This creates the `SimpleAudioPlayer.class` file in the working directory.

Then run the example application by typing the following at a command prompt:

```
        java SimpleAudioPlayer audioFile
```

Replace *audioFile* with the file name of an audio file on your local system. Any relative file names will be resolved relative to the current working directory. You should see some messages indicating the file that is being played. To stop playing, press the Enter key.

If compilation failed, check to make sure that the JMF jar files are included in the current `CLASSPATH` environment variable.

JMF user interface components

Playing video

In the previous section, we walked through the steps of setting up an application that lets you play audio files through a command-line interface. One of the great features of JMF is that you don't need to know anything about the media file types in order to configure a media player; everything is handled internally. For instance, in our previous example, we did not need to tell the application to create a `Player` specifically for an MP3 file, since the MP3 setup was handled for us.

As you will see in this section, the same holds true for handling video files. JMF handles all of the details of interfacing with the media file types.

The main difference in dealing with video media rather than audio is that we must create a visual representation of a screen to be able to display the video. Luckily, JMF handles many of these details for us. We will create a `Player` instance much like we did in the previous example, and obtain many of the visual components to create our visual media viewer directly from JMF objects.

In this section, we'll walk through the second example application. Refer to the MediaPlayerFrame.java in the source code distribution to follow this exercise.

About the example

In this section we'll create an application that can display and run both audio and video media from your local machine. As part of this exercise, we will explore some of JMF's built-in GUI components. Familiarity with AWT and Swing will help you follow the examples, but is not required. We won't go into much detail about the source code for the GUI unless it relates directly to JMF GUI components. You will find that many details that are not covered here are explained in the source code comments.

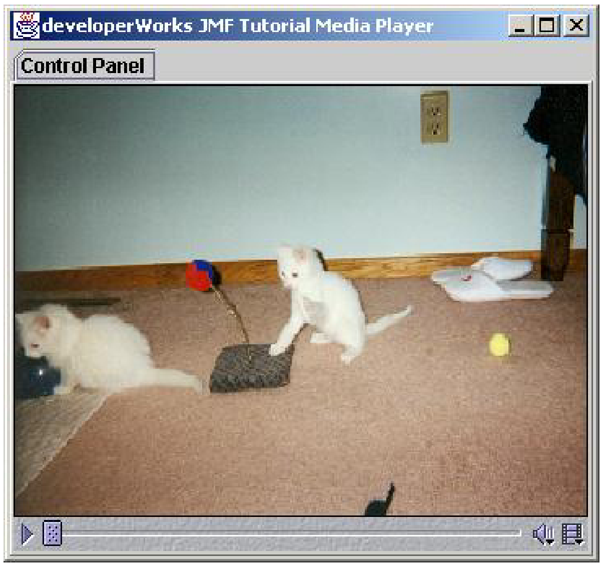
Most of the concepts, classes, and methods we'll use in this example will be familiar from the first example. The basics of setting up the `Player` are almost identical. The biggest difference is that we're going to dig a little deeper into the `Player` instance, particularly when it comes to getting information about the media from the `Player`.

Getting started

The video player example is designed to run from the command line, much like the audio player example did, but this example is GUI based. We start by invoking the application and passing in the file name of the media, just as we did in the previous section. Next, the application displays a window with components that let us manipulate the media.

In the opening lines of the `MediaPlayerFrame` we define the class and extend the `javax.swing.JFrame` class. This is how we define our media player as a separate window on the desktop. Any client that creates an instance of our media player class may then display it by using the `show()` method defined on the `JFrame` class.

Below is a sample screenshot showing the `MediaPlayerFrame` playing an MPEG movie:



Getting the GUI components

The `Player` interface exposes methods to obtain references to selected visual components. In the `MediaPlayerFrame` we use the following components:

- `player.getVisualComponent()` is the visual component responsible for displaying any video media.
- `player.getControlPanelComponent()` is a visual component for handling time-based operations (that is, start, stop, rewind) as well as containing some useful information on the media stream.
- `player.getGainControl().getControlComponent()` is a visual component for handling volume (gain) operations. The `getGainControl()` method returns a `GainControl` instance, which may be used to change gain levels programmatically.

Working with visual components

All of the above interface methods return an instance of the `java.awt.Component` class. Each instance is a visual component that may be added to our frame. These components are tied directly to the `Player`, so any manipulation of visual elements on these components will cause a corresponding change to the media displayed by the `Player`.

It is important that we ensure each of these components is not `null` before we add it to our frame. Because not every type of media player contains every type of visual component, we should only add components that are relevant for the type of player we have. For instance, an audio player generally does not have a visual component, so `getVisualComponent()` returns `null`. You would not want to add a visual component to the audio player frame.

Obtaining media-specific controls

A `Player` instance may also expose other controls through its `getControl()` and `getControls()` methods -- `getControls()` returns a collection of `Control` objects, whereas the `getControl()` method looks for a specific `Control`. Different types of players may choose to expose controls for operations specific to a given media type or to the transport mechanism used to obtain that media. If you were to write an application that handled only certain media types, you could count on certain `Control` objects being available through the `Player` instance.

Because our player is very abstract and designed to work with many different media types, we simply expose all the `Control` objects to the user. If we find any extra controls, we can use the `getControlComponent()` method to add their corresponding visual component to a tabbed pane. This way, the user will be able to view any of these components through the player. The following code fragment exposes all the control objects to the user:

```
    Control[] controls = player.getControls();
    for (int i = 0; i < controls.length; i++) {
       if (controls[i].getControlComponent() != null) {
          tabPane.add(controls[i].getControlComponent());
       }
    }
```

For a real application to do something useful with a `Control` instance (besides being able to display its visual component), the application would need to know the specific type of the `Control` and cast it to that type. After that point, the application could use the control to manipulate the media programmatically. For example, if you knew the media you were working with always exposed a `Control` of type `javax.media.control.QualityControl`, you could cast to the `QualityControl` interface and then change any quality settings by calling any of the methods on the `QualityControl` interface.

Using a MediaLocator

The last big difference between our new GUI-based media player and our first simple player is that we'll use a `MediaLocator` object rather than a `URL` to create the `Player` instance, as shown below:

```
    public void setMediaLocator(MediaLocator locator) throws IOException,
       NoPlayerException, CannotRealizeException {

       setPlayer(Manager.createRealizedPlayer(locator));
    }
```

We'll discuss the reason for this change in a later section. For now, think of a `MediaLocator` object as being very similar to a URL, in that both describe a resource location on a network. In fact, you may create a `MediaLocator` from a URL, and you may get a URL from a `MediaLocator`. Our new media player creates a `MediaLocator` instance from a URL and uses that to create a `Player` over the file.

Compiling and running the MediaPlayerFrame

Compile the example application by typing `javac MediaPlayerFrame.java` at a command prompt. This creates a file named `MediaPlayerFrame.class` in the working directory.

To run the example application, type the following at a command prompt:

```
       java MediaPlayerFrame mediaFile
```

You should replace *mediaFile* with the file name of a media file on your local system (either audio or video will do). Any relative file names will be resolved relative to the current working directory. You should see a window that displays GUI controls for manipulating your media file. For a list of audio and video file formats acceptable for use in JMF, see Related topics.

If the initial compilation failed, check to make sure that the JMF jar files are included in the current `CLASSPATH` environment variable.

MediaPlayerFrame in action

Earlier in this section you saw a screenshot of a video player playing an MPEG video file. The following screenshot shows an audio player playing an MP3 file:

Take a look at the complete MediaPlayerFrame source code to further study the examples in this exercise.

Conceptual JMF

The JMF architecture

Now that you've seen how easy it is to play local media files using JMF, we'll take a step back and look at the bigger picture of how you can use JMF to create more sophisticated media-based applications. This will by no means be a comprehensive survey of the JMF architecture. Rather, this section will give you a general idea of how high-level JMF components can be combined to create desired effects.

The JMF component architecture is very flexible, and its components can generally be classified in three groups:

- **Input** describes some sort of media that is used as an input to the rest of the process.
- A **process** performs some sort of action on the input. A process has a distinct input and output. A large number of processes are available, and can be applied to an input or a group of inputs. These processes can be chained together so that the output from one process is used as an input to another process. In this manner multiple processes may be applied to an input. (This stage is optional -- our first two examples contained no real data processing, only an input from a file and an output through the `Player`.)
- **Output** describes some sort of destination for media.

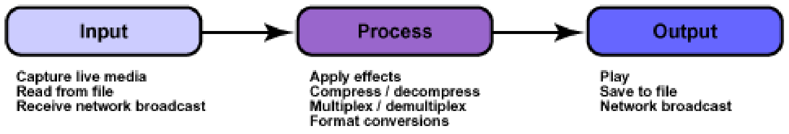
From this description, you might be thinking that the JMF component architecture sounds quite like that behind a typical stereo or VCR. It is easy to imagine using JMF in a manner similar to that of turning on a television or adjusting the sound on a stereo. For example, the simple act of recording a favorite TV show can be thought of in these component-based terms:

- **Input** is the TV broadcast stream, carrying both audio and video information on the same channel.

- **Process** is a recording device (that is, a VCR or any number of digital devices) converting the analog or digital audio and video broadcast stream into a format suitable for copying to a tape or some other media.
- **Output** is the device writing the formatted track (audio and video) to some type of media.

The JMF data processing model

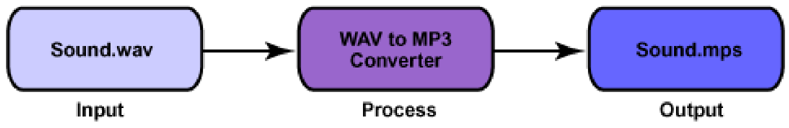
The image below illustrates the JMF data processing model and gives some examples of each type:



Using this model, it is easy to follow our first two examples from file input through the audio and video output on a local machine. In the sections that follow, we'll also explore some of the networking functionality of JMF by broadcasting and receiving audio media over a network.

Process model example

By chaining together JMF's input, process, and output models, we can begin to conceive of the many media-based operations that are possible through JMF. One example would be converting media of one type into another and storing the output to a new file. For instance, say we wanted to convert an audio file in the WAV format to an MP3 format without destroying the original file. The steps we would take to perform the conversion are illustrated in the following process model:



The input for this example is a WAV file. It is processed by a media format converter, and the output is placed in a new file. Now, let's look at each step in this model through the JMF API. We'll use the input, process, and output models as conceptual guideposts.

JMF input

In JMF, an input is generally represented by a `MediaLocator` object. As stated previously, the `MediaLocator` looks and acts much like a URL, in that it uniquely identifies a resource in a network. In fact, it is possible to create a `MediaLocator` using a URL; we did this in our two previous example applications.

For the purpose of our media conversion example, we could build a `MediaLocator` to describe the original WAV file. As we will see in the next few sections, a `MediaLocator` may also be used to represent a media stream being broadcast across a network. In this case, instead of building the

`MediaLocator` to identify a file on the local file system, the `MediaLocator` would describe the URL of the broadcast -- much like a resource on the Web is identified by its URL.

The difference between a MediaLocator and a URL

A successfully created `URL` object requires that the appropriate `java.net.URLStreamHandler` class be installed on the system. The purpose of the stream handler is to be able to handle the stream type described by the URL. A `MediaLocator` object does not have this requirement. For example, our next application will transmit audio over a network using the Real-Time Transport Protocol (RTP). Because most systems do not have a `URLStreamHandler` installed for the RTP protocol, an attempt to create a `URL` object for this purpose would fail. For this application, only the `MediaLocator` object can succeed.

For more information on `URL` objects and on creating and registering a `URLStreamHandler` , refer to the javadoc in the JDK API documentation (see Related topics).

JMF processors

When we are working with JMF, the processor component of the application is represented by an instance of the `Processor` interface. You should already be somewhat familiar with the `Processor`, as it is an extension of the `Player` interface. Because the `Processor` inherits from the `Player` interface, it also inherits all of the valid states from the `Player`. In addition, the `Processor` adds two more states: *Configuring* and *Configured*. These extra states (and associated events) are used to communicate when the `Processor` is gathering information from the input stream.

For our final example application, we will create a `Processor` to convert audio encoded in the MP3 format into a format suitable for broadcast over a network. We will discuss the steps to create a simple `Processor` in a later panel.

JMF output

There are a few ways to represent the output phase of the JMF process model. The simplest (and the one we will use in the final example) is the `javax.media.DataSink` interface. A `DataSink` reads media content and renders it to some destination. In the audio-format conversion scenario at the beginning of this section, the MP3 (output) file would be represented by the `DataSink`. In our final example, we will use a `DataSink` to actually do the work of broadcasting audio media over a network. A `DataSink` is created through the `Manager` class by specifying a `DataSource` (the input to the `DataSink`) and a `MediaLocator` (the output of the `DataSink`).

A `DataSource` instance represents input data, which is used in `Player`s, `Processor`s, and `DataSink`s. The output of a `Processor` is also represented as a `DataSource` object. This is how `Processor`s can be chained together to perform multiple operations on the same media data. This is also how the output from a `Processor` can be used as input to a `Player` or to a `DataSink` (which would render the media to an output destination).

The final destination of a `DataSink` is specified by a `MediaLocator` object. Just as before, the `MediaLocator` represents a network resource; that is, it's where the media stream will be rendered.

Broadcasting and receiving media

JMF and Real-Time Transport Protocol (RTP)

Many network-friendly features are built directly into JMF, which makes broadcasting and receiving media over a network very easy for client programmers. When a user on a network wants to receive any type of streaming media, he shouldn't have to wait for the entire broadcast to download to the machine before viewing the media; the user should be able to view the broadcast in real time. This concept is referred to as *streaming media*. Through streaming media, a network client can receive audio being broadcast by another machine or even intercept a live video broadcast as it is happening.

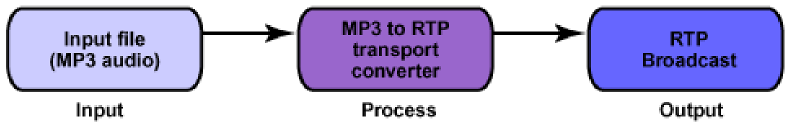
The Real-Time Transport Protocol (RTP) is defined in IETF RFC 1889. Developed to carry extremely time-sensitive data over a network in a quick and reliable manner, RTP is used in JMF to provide users with a way to transport media streams to other network nodes.

In this section, we'll walk through our final example application. Here, you'll learn how to broadcast an MP3 file stored on one machine to other machines in the same network. The actual MP3 source file never leaves the host machine, nor is it copied to any other machine; rather, it will be translated into a format that can be broadcast using RTP and sent over the network. Upon being received by a client, the source file (now in the form of RTP packets) may again be translated, this time to a format suitable for play on the receiving machine.

Refer to the MediaTransmitter.java file in the source code distribution to follow the exercises in this section.

Setting up the process model

We can talk about our final example in terms of the process model we defined in the previous section. On the transmitting machine, the process model would look something like this:



In fact, the source code for the `MediaTransmitter` object contains the following three lines:

```
    private MediaLocator mediaLocator = null;

    private DataSink dataSink = null;

    private Processor mediaProcessor = null;
```

These three instance variables may be mapped directly into the above process model diagram, as follows:

- The `mediaProcessor` variable is our processor; it will be responsible for converting our audio media from the MP3 file format to a format suitable for transmission over the RTP protocol.

- The `dataSink` variable is our output block.
- When we create the `DataSink` we will specify a `MediaLocator`, which is the actual destination of the `DataSink`.

When we run our processed media through the `DataSink`, it will be transmitted to whatever location(s) we have specified in the `MediaLocator`.

The RTP MediaLocator

In the previous two exercises, we created `MediaLocator` instances by using a URL, which we obtained from a file. For this exercise we must create a `MediaLocator` that describes the network output for media transmission; in other words, we must create a `MediaLocator` that can be the destination for our audio broadcast. An RTP `MediaLocator` conforms to the following form, which looks like a typical URL:

```
    rtp://address:port/content-type
```

Let's look at each piece of the above URL specification:

- **address** is the address to which the media will be transmitted. To transmit in unicast mode (to one specific IP address), the address should be the IP address of the intended receiving machine. To transmit in broadcast mode (to all machines within a subnet), the address should be the subnet address with *255* as the last section. For example, if I were on the subnet denoted as `192.168.1` and I wanted to broadcast to all nodes, I could specify `192.168.1.255` as the address; this would enable each node in the subnet to listen to the broadcast media.
- **port** must be a port that has been agreed upon by both transmitters and receivers.
- **content-type** is the type of streamed media. In our case this will always be `audio`.

The following simple example of an RTP broadcast `MediaLocator` will let any machine on the specified network receive streamed media:

```
    rtp://192.168.1.255:49150/audio
```

Creating a processor

The first thing we do in the `setDataSource()` method is to create a `Processor` instance. The following `Processor` will be responsible for converting our MP3 audio media into an RTP representation of the audio:

```
     public void setDataSource(DataSource ds) throws IOException,
       NoProcessorException, CannotRealizeException, NoDataSinkException {

         mediaProcessor = Manager.createRealizedProcessor(
             new ProcessorModel(ds, FORMATS, CONTENT_DESCRIPTOR));
```

In the `Manager` class, we can create a `Processor` object through one of two method types: `createProcessor()` or `createRealizedProcessor()`. You will likely note that these two methods act in a manner similar to the methods used in previous exercises to create a `Player`. For the current example, we will create a realized `Processor`. We can do this because the application we're working on is simple, and we don't care to do any real work until after the `Processor` is in the *Realized* state.

Creating a ProcessorModel

To create a realized `Processor`, we need to create a `ProcessorModel` instance that describes the media types for the inputs and outputs to the `Processor`. In order to create the `ProcessorModel`, we need the following things:

- A `DataSource`, which is the media that will be processed (the input file).
- A `javax.media.Format` array, which describes the format of the input media.
- A `javax.media.protocol.ContentDescriptor` instance, which describes the output format of our processor. The `DataSource` for the transmitter is passed in as a parameter to the method.

Defining the input and output formats

Because our `MediaTransmitter` class will always be used to take one type of input media (MP3) and create one type of output (audio RTP) these objects are declared as static. We create a new `javax.media.format.AudioFormat` instance to describe the media input type (see the javadoc to learn about the available formats). This is why our processor may only take MP3 audio files.

We also create a `javax.media.protocol.ContentDescriptor` instance to describe what we want the output of our processor to be. In our case this is an RTP media stream. This is why our processor may only produce RTP streams.

The following code snippet shows how we set up the format and content descriptor variables, which are used to create the `ProcessorModel` object.

```
      private static final Format[] FORMATS = new Format[] {
        new AudioFormat(AudioFormat.MPEG_RTP)};
      private static final ContentDescriptor CONTENT_DESCRIPTOR =
        new ContentDescriptor(ContentDescriptor.RAW_RTP);
```

Linking inputs, processors, and outputs

Now that we have a `Processor` in the *Realized* state, we need to set up the `DataSink` to be able to actually broadcast the RTP media. Creating the `DataSink` is simply a matter of making another call to the `Manager` object, as shown below:

```
        dataSink = Manager.createDataSink(mediaProcessor.getDataOutput(),
                                          mediaLocator);
```

The `createDataSink()` method takes the output of our new `Processor` (as a `DataSource` parameter) and the `MediaLocator` object, which we created simultaneously with the `MediaTransmitter` object. From this, you can begin to see how our different components are linked together in the process model: we take the outputs from a `Processor` and use them as the inputs to other components. For this particular application, the `Processor` output is used as an input to the `DataSink`, which is then used to transmit media.

Creating the DataSource instance

At this point we are all but done with setting up our media player for broadcast transmission. We just have to create the `DataSource` instance, which we'll use to create our processor (that is, the parameter passed to the `setDataSource()` method on our `MediaTransmitter`). Here's the code to create the `DataSource` instance:

```
        File mediaFile = new File(args[1]);
        DataSource source = Manager.createDataSource(new MediaLocator(
                                mediaFile.toURL()));
```

This code is from the `main()` method on the `MediaTransmitter` object. Here we create a `File` object from the second argument passed in through the command line. We create a `MediaLocator` from the file, and subsequently create a `DataSource` from the locator. This newly created `DataSource` is a reference to the input file for the transmitter. We can then use this `DataSource` to initialize the transmitter.

Starting and stopping the MediaTransmitter

We start the `MediaTransmitter` by calling the `startTransmitting()` method on it, as shown here:

```
      public void startTransmitting() throws IOException {
          mediaProcessor.start();

          dataSink.open();
          dataSink.start();
      }
```

This method starts the processor first, then opens and starts the `DataSink`. After this call, the receiving machines should be able to listen in on the media transmitter.

Stopping the transmitter is just as simple. The following method call stops and closes both the `DataSink` and the `Processor`:

```
      public void stopTransmitting() throws IOException {
          dataSink.stop();
          dataSink.close();

          mediaProcessor.stop();
          mediaProcessor.close();
      }
```

Compiling and running the MediaTransmitter

Compile the example application by typing `javac MediaTransmitter.java` at a command prompt to create a .class file of the same name in your working directory.

To run the example application, type the following at a command prompt:

```
        java MediaTransmitter rtpMediaLocator audioFile
```

This example should create a media broadcast of the myAudio.mp3 file. Don't forget to replace *rtpMediaLocator* with an RTP URL for the media transmission, as discussed earlier. You should also replace *audioFile* with the file name of an audio file on your local system. Any relative file names will be resolved relative to the current working directory. You should see some messages indicating what file is being played. To stop playing, press the Enter key.

An example command-line interaction for the transmitter is:

```
        java MediaTransmitter rtp://192.168.1.255:49150/audio myAudio.mp3
```

If initial compilation failed, check to make sure that the JMF jar files are included in the current `CLASSPATH` environment variable. Refer to the MediaTransmitter source code to further explore this application and exercise.

Receiving transmitted media

Now, you may ask, "What good is it to broadcast media if nobody can see or listen to it?" Fortunately, setting up a client to receive broadcast media requires only a very small change to the MediaPlayerFrame source code of our second example application.

The `MediaPlayerFrame` class needs one minor tweak to be able to receive and play the transmitted audio media. In the `main()` method, you should comment out the following line:

```
        mpf.setMediaLocator(new MediaLocator(new File(args[0]).toURL()));
```

And uncomment the following line:

```
        mpf.setMediaLocator(new MediaLocator(args[0]));
```

This simple change allows us to create a `MediaLocator` object using the passed-in `String`, rather than creating a `File` reference and using that to create the `MediaLocator`. All other code remains the same.

Specifying the RTP URL

Refer to Compiling and running the MediaPlayerFrame for instructions on how to compile and run the `MediaPlayerFrame` example application. The only difference is that now you need to specify the RTP URL for the transmitter. An example command-line interaction for the receiver is:

```
        java MediaPlayerFrame rtp://192.168.1.255:49150/audio
```

Notes on running the network media transmitter

If you only have access to one machine on a network, you can still run the transmitter application. When you start the transmitter, you may either use a broadcast address for the RTP URL, or specify the machine address for the machine you're working on. In order to be able to tune into the transmission, the receiver must use the exact same RTP URL upon being started.

If you're running a truly networked version of these examples, each machine you use needs to have JMF installed in order to either transmit or receive streamed media. This is necessary because both the transmitter and receiver applications make heavy use of JMF APIs.

In either case, be sure to use the same `address` and `port` parameters in the RTP URL specification; otherwise the media transmission will not work.

Wrapup

Summary

I hope this tutorial has offered you an exciting glimpse of what you can do with the JMF API. We have created three small applications to play local audio and video, as well as broadcast and receive media over a network. The source code for these applications contains many javadoc-style comments. These should help to answer any remaining questions you have.

Many JMF topics were not covered in this tutorial. Rather, we've focused on the basic concepts and applications of JMF; with this foundation, you can more easily branch out into other areas of study. To get started on deeper applications of JMF, you may want to follow up on some of the advanced topics mentioned in the next section. For further reading on any of the topics covered in this tutorial, refer to Related topics.

Advanced topics

A number of worthwhile exercises were beyond the scope of this tutorial. With the brief introductions below and further study on your own, you could extend the source for our three applications, and also expand your knowledge of how JMF works. Try the following exercises to get started:

- **Media capture**: JMF contains rich APIs for capturing media data. if you're interested in doing media capture with JMF, you should begin your exploration with the `javax.media.CaptureDeviceManager` class and the `javax.media.protocol.CaptureDevice` interface APIs. For an advanced exercise, consider working with `CaptureDeviceManager` and the `CaptureDevice` interface to add media capture functionality to the GUI version of the media player application.
- **Session managers**: Because this tutorial is an introduction to JMF, we kept our output representation very simple, implementing only the `javax.media.DataSink` output. Another output representation is with the `javax.media.rtp.SessionManager`. This manager class allows clients to create and monitor their own RTP streams and connections. Through the `SessionManager` and the subsequently created streams, it is possible to more closely monitor RTP sessions. For an advanced exercise, convert our third demo application to use the `SessionManager`, and then monitor the outgoing RTP streams and which clients are tuning in.
- **Using Multicast with JMF** : Our broadcast demo application explained how to send media over a network to either one or all machines on a network. It is also possible to use the Multicast Transport Protocol in JMF to provide for more sophisticated, multi-user networks. The JMF User's Guide provides a more in-depth discussion of using JMF with the Multicast protocol. See Related topics to pursue this topic further.
- **Transmitting video**: Our final demo application looked at how to transmit an MP3 audio file, but JMF is very capable of sending video over a network as well. Take a look at the API documentation for the `Format` and `ContentDescriptor` classes to get a better idea of how this is done.

- **Import/export RTP media streams**: JMF also allows RTP streams to be saved to a file for use at a later time. For instance, a teleconference may be saved for viewing at a later time. Because the stream is already saved in the RTP format, there is no need to convert it again, which can result in a performance improvement for the transmitting application. Try setting the input/output `MediaLocator` of the `DataSink` object to a file rather than a URL. You will find this topic discussed in more depth in the JMF User's Guide.

Downloadable resources

Description	Name	Size
	j-jmf.zip	12KB

Related topics

- The JMF home page is the best resource for discovering more information about JMF.
- You'll find the JMF specifications, including API documentation and the JMF user guides, on the Java Developer Connection. You will need access to all of these resources if you want to do any kind of in-depth JMF programming.
- The official JMF-supported file formats page lists all file formats that are recognizable and playable by the JMF. The file formats page also contains references for learning more about capture devices and RTP formats.
- The IETF RTP RFC describes the RTP protocol in great detail.
- Refer to the JMF API Guide for a much more involved description of the RTP protocol and how it relates directly to JMF.
- Columbia University maintains a helpful RTP FAQ.
- Another Sun tutorial walks through the basics of network programming in the Java programming language.
- Todd Montgomery maintains an MTP page, where you can find a comprehensive list of links related to Multicast Transport Protocol.
- See the developerWorks tutorials page for a complete listing of free tutorials.
- Download mediaplayer.jar (see Download), the complete source for the examples used in this tutorial.
- The Java 2 Platform, Standard Edition is available from Sun Microsystems.