Java Media Framework

Multimedia Systems: Module 3 Lesson 1

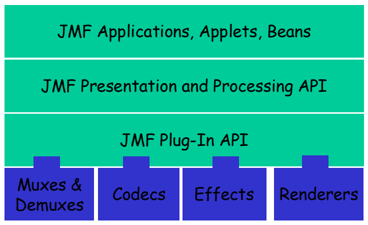
Summary:

- JMF Core Model
  - Architecture
  - Models: time, event, data
- JMF Core Functionality
  - Presentation
  - Processing
  - Capture
  - Storage and Transmission
- JMF Extensibility

Sources:

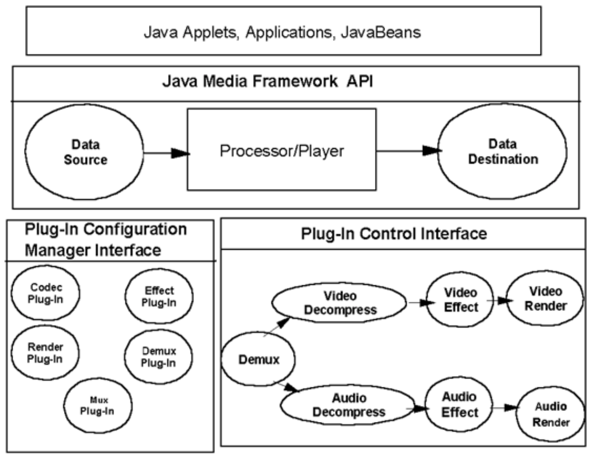
- JMF 2.0 API Programmers Guide from Sun: http://java.sun.com/products/java-media/jmf/2.1/guide/
- JMF 2.0 API
- JMF White Paper from IBM http://www-4.ibm.com/software/developer/library/jmf/jmfwhite.html

High Level Architecture


JMF Applications, Applets, Beans
JMF Presentation and Processing API
JMF Plug-In API
Muxes & Demuxes
Codecs
Effects
Renderers

- A ***renderer*** is an abstraction of a presentation device. For audio, the presentation device is typically the computer's hardware audio card that outputs sound to the speakers. For video, the presentation device is typically the computer monitor.
- A ***demultiplexer*** extracts individual tracks of media data from a multiplexed media stream. A ***mutliplexer*** performs the opposite function, it takes individual tracks of media data and merges them into a single multiplexed media stream.
- A ***codec*** performs media-data compression and decompression. Each codec has certain input formats that it can handle and certain output formats that it can generate
- An ***effect*** filter modifies the track data in some way, often to create special effects such as blur or echo

Framework

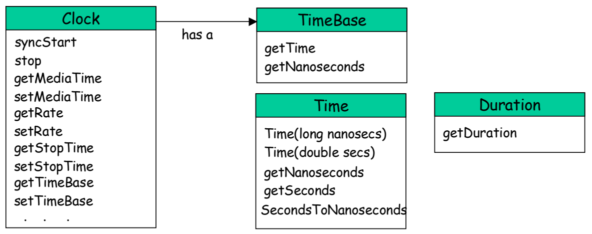


The Java Media Framework Structure

JMF

- **Media Streams**
  - A *media stream* is the media data obtained from a local file, acquired over the network, or captured from a camera or microphone. Media streams often contain multiple channels of data called *tracks*. For example, a Quicktime file might contain both an audio track and a video track.
  - A track's *type* identifies the kind of data it contains, such as audio or video. The *format* of a track defines how the data for the track is structured.
    - Formats: (V) Cinepak, H.261, MPEG-1, MPEG-2, Indeo; (A): μ-Law, PCM, ADPCM, MPEG-1, MPEG-3, G.723.1, GSM
  - Media streams can be categorized according to how the data is delivered:
    - **Pull**--data transfer is initiated and controlled from the client side. E.g., HTTP and FILE are pull protocols.
    - **Push**--the server initiates data transfer and controls the flow of data. E.g., Real-time Transport Protocol (RTP) is a push protocol used for streaming media. Similarly, the SGI MediaBase protocol is a push protocol used for video-on-demand (VOD).

JMF Time Model



- The `Clock` interface is implemented by objects that support the Java Media time model. E.g., this interface might be implemented by an object that decodes and renders MPEG movies.
- A Clock uses a `TimeBase` to keep track of the passage of time while a media stream is being presented. A `TimeBase` provides a constantly ticking time source.

Time Model

- A Clock object's *media time* represents the current position within a media stream--the beginning of the stream is media time zero, the end of the stream is the maximum media time for the stream. The *duration* of the media stream is the length of time that it takes to present the media stream.
- To keep track of the current media time, a Clock uses:
  - The *time-base start-time* -- the time that its `TimeBase` reports when the presentation begins.
  - The *media start-time* -- the position in the media stream where presentation begins.
  - The *playback rate* --how fast the Clock is running in relation to its TimeBase. The *rate* is a scale factor that is applied to the TimeBase.
- When presentation begins, the media time is mapped to the time-base time and the advancement of the time-base time is used to measure the passage of time. During presentation, the current media time is calculated using the following formula:

  MediaTime = MediaStartTime + Rate(TimeBaseTime - TimeBaseStartTime)

Managers

- JMF uses four managers:
  - **Manager**--handles the construction of Players, Processors, DataSources, and DataSinks. This level of indirection allows new implementations to be integrated seamlessly with JMF. From the client perspective, these objects are always created the same way whether the requested object is constructed from a default implementation or a custom one.
  - **PackageManager**--maintains a registry of packages that contain JMF classes, such as custom Players, Processors, DataSources, and DataSinks.
  - **CaptureDeviceManager**--maintains a registry of available capture devices.
  - **PlugInManager**--maintains a registry of available JMF plug-in processing components, such as Multiplexers, Demultiplexers, Codecs, Effects, and Renderers.
- To write programs based on JMF, you will use the Manager `create` methods to construct the Players, Processors, DataSources, and DataSinks for your application.
- If you're capturing media data from an input device, you'll use the CaptureDeviceManager to find out what devices are available and access information about them.
- If you're interested in controlling what processing is performed on the data, you might also query the PlugInManager to determine what plug-ins have been registered.

Event Model

- Whenever a JMF object needs to report on the current conditions, it posts a `MediaEvent`. `MediaEvent` is sub-classed to identify many particular types of events.
- For each type of JMF object that can post MediaEvents, JMF defines a corresponding listener interface. To receive notification when a `MediaEvent` is posted, you implement the appropriate listener interface and register your listener class with the object that posts the event by calling its `add`*`Listener`* method.
- Controller objects (such as `Players` and `Processors`) and certain Control objects such as `GainControl` post media events.

Data Model

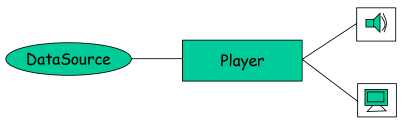
- JMF media players usually use `DataSources` to manage the transfer of media-content. A `DataSource` encapsulates both the location of media and the protocol and software used to deliver the media. Once obtained, the source cannot be reused to deliver other media.
- A `DataSource` is identified by either a JMF `MediaLocator` or a URL (universal resource locator). A `MediaLocator` is similar to a URL and can be constructed from a URL, but can be constructed even if the corresponding protocol handler is not installed on the system. (*Note*: In Java, a URL can only be constructed if the corresponding protocol handler is installed on the system.)
  - A standard data source uses a byte array as the unit of transfer. A *buffer data source* uses a Buffer object as its unit of transfer.
- JMF data sources can be categorized according to how data transfer is initiated:
  - *Pull Data-Source*--the client initiates the data transfer and controls the flow of data from pull data-sources.
  - *Push Data-Source*--the server initiates the data transfer and controls the flow of data from a push data-source. Push data-sources include broadcast media, multicast media, and video-on-demand (VOD).

JMF: Main Functionality

- Presentation
  - Take media content from a `DataSource` and render it.
  - This functionality is contained in the `Controller` interface
    - `Player` extends this interface
- Processing
  - Take media content from a `DataSource`, perform some user-defined processing on it, and output it
  - This functionality is contained in the `Processor` interface
    - `Processor` extends the `Player` interface
- Capture
  - A capturing device can act as a source for multimedia data.
  - Capture devices are abstracted as `DataSources`.
- Media Storage and Transmission
  - A `DataSink` reads media data from a `DataSource` and renders it to some destination (generally a destination other than a presentation device).
  - E.g., A `DataSink` might write data to a file, write data across the network, or function as an RTP broadcaster.

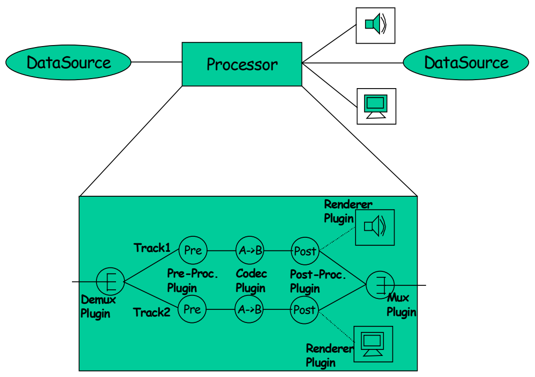
Player

- A `Player` processes an input stream of media data and renders it at a precise time. A `DataSource` is used to deliver the input media-stream to the Player.The rendering destination depends on the type of media being presented.



Processor

- A `Processor` is a `Player` that takes a DataSource as input, performs some user-defined processing on the media data, and then outputs the processed media data.



JMF:Extensibility

One can extend JMF by implementing custom plug-ins, media handlers, and data sources.

- By implementing one of the JMF plug-in interfaces, one can directly access and manipulate the media data associated with a Processor:
  - Implementing the `Demultiplexer` interface enables you to control how individual tracks are extracted from a multiplexed media stream.
  - Implementing the `Codec` interface enables you to perform the processing required to decode compressed media data, convert media data from one format to another, and encode raw media data into a compressed format.
  - Implementing the `Effect` interface enables you to perform custom processing on the media data.
  - Implementing the `Multiplexer` interface enables you to specify how individual tracks are combined to form a single interleaved output stream for a Processor.
  - Implementing the `Renderer` interface enables you to control how data is processed and rendered to an output device.

JMF:Extensibility

- **Implementing MediaHandlers and DataSources**
  - If the JMF Plug-In API doesn't provide the degree of flexibility that you need, you can directly implement several of the key JMF interfaces: `Controller`, `Player`, `Processor`, `DataSource`, and `DataSink`.
    - For example, you might want to implement a high-performance `Player` that is optimized to present a single media format or a `Controller` that manages a completely different type of time-based media.
  - The `Manager` mechanism used to construct `Player`, `Processor`, `DataSource`, and `DataSink` objects enables custom implementations of these JMF interfaces to be used seamlessly with JMF.
    - When one of the `create` methods is called, the `Manager` uses a well-defined mechanism to locate and construct the requested object.
    - Your custom class can be selected and constructed through this mechanism once you register a unique package prefix with the `PackageManager` and put your class in the appropriate place in the predefined package hierarchy.