<JSTORM>

자바 멀티미디어 JMF

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

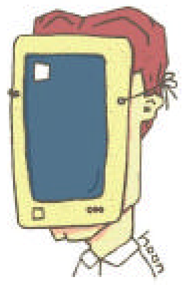
Document Information

Document title:	자바 멀티미디어 JMF
Document file name:	JMF1_심빈구.doc
Revision number:	<1.0>
Issued by:	<심빈구> raica@nownuri.net
	정리 : <윤준호> junoyoon@orgio.net
Issue Date:	<2000/8/25 >
Status:	final

Content Information

Audience	마이크로 소프트웨어 독자
	자바 멀티미디어 파운데이션 클래스(JFC)의 사용방법을 알고 싶어하는 중급 프로그래머
Abstract	JMF 프로그래밍
	- 인터넷 방송 프로그램 제작을 위한 사전단계로 간단한 Media 플레이어와 Media Converter를 만들어 봄으로써 JMF의 기초를 익힌다.
Reference	
Benchmark information	

Document Approvals

고문	Signature 	date
지위	Signature	date

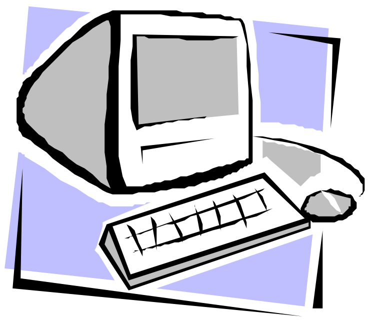
Revision History

Revision	Date	Author	Description of change

Table of Contents

자바 멀티미디어 JMF

요사이 자바 진영에서 새롭게 발표하는 제품들은 그 수가 날로 늘고 있다. 실제로 자바 사이트를 자주 방문하는 사람이라면 누구나 동감할 것이다. 그렇다고 해서 발표되는 제품들이 모두 질이 떨어지거나 흥미가 떨어지는 주제들 또한 아니다. 발표되는 제품 하나 하나를 볼 때마다 모두 한번쯤은 다루어 보고 싶은 충동을 갖도록 하는 제품들이다. 이번 연재에서는 그 중에 최근에 2.0 버전이 발표된 JMF에 대해 다루어 보겠다.



JMF 첫걸음, JMF의 구조를 파해쳐 보자.

JMF의 배경

JMF는 1998년 중반 썬과 SGI(Sillicon Graphics) 그리고 인텔사가 같이 1.0버젼의 스펙을 디자인 했었다. 그러나 초기버전이 나오기 전에 SGI와 인텔사가 프로젝트에서 떨어져 나가게 됐다. 그 후부터 당분간은 썬 혼자만이 JMF의 개발 맡게 되었다. 하지만 썬사 혼자서 개발을 진행시키기엔 그 분야에서 무리가 있었다. 썬사에 축적된 멀티미디어 관련 기술은 그다지 없기 때문이다.

하지만 다행히도, 1998년 말경에 IBM이 JMF에 흥미를 갖고 개발에 참여하게 되었다. 당시에는 독자들도 알겠지만 IBM에는 Jikes나 IBM JDK, RMI for IE등등 자바에 관련된 제품들을 발표하면서, 자바에 관심을 보였던 시기였다. 덕분에 JMF개발은 무사히 진행될 수 있었으며, 그 후로 무사히 개발을 진행시킬 수 있었다. 그 결과 1998년 JMF1.1을 발표할 수 있게 되었다. 그 후 버전이 업 되어 작년 11월 말경에는 JMF2.0 FCS 버전이 발표되었다. JMF2.0버전에서는 아래와 같은 기능이 추가되었다.

- 오디오&비디오 캡춰
- 오디오와 비디오를 스트림 형태로 관리(따라서 스트림을 이용한 모든 서버클라이언트 모델에 추가할 수 있게 됐다)
- Player에서의 Pluggable Codec 지원

JMF패키지 종류와 설치

JMF는 총 3가지의 패키지로 제작되었는데, Solaris용과 Window용 그리고, Cross-Platform 용 이렇게 세가지다. JMF는 그 특성상 시스템의 하드웨어를 다루는 작업을 많이 하기 때문에 Cross-Platform이외에 OS에 맞게 최적화 해서 제작된 패키지를 배포하고 있다. 실제로 캡쳐 기능은 Cross-Platform에서는 지원하지 못한다. 따라서 제대로된 JMF의 기능을 이용하기 위해선 독자의 OS에 맞는 패키지를 설치하기 바란다. 이 외에 JDK와의 호환성도 조금 문제가 되는데, 자세한 사항은 박스기사를 참조하기 바란다.

지원하는 미디어 타입

http://java.sun.com/products/java-media/jmf/2.0/supported.html

프로젝트 개요

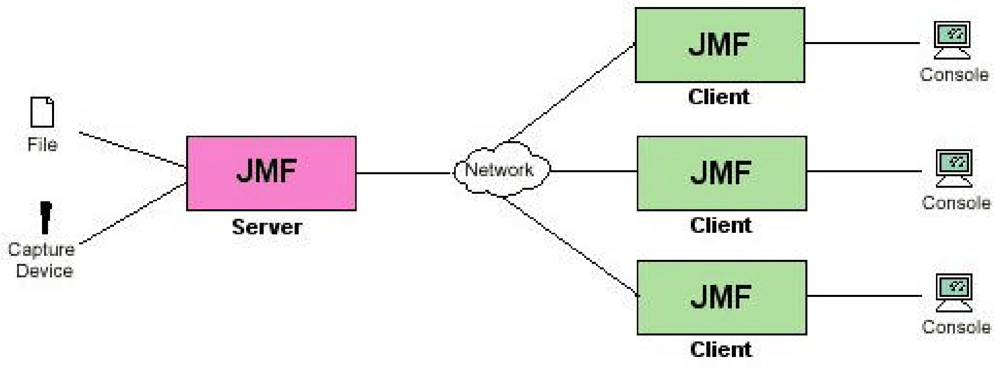


그림 1 방송국 프로그램 구조

이번 연재의 목표는 인터넷 방송 프로그램 만들기이다. JMF 캡춰기능을 사용하여 방송할 미디어 자료를 받아들이고, JMF RTP API를 사용하여 미디어 자료를 실시간으로 전송하고, 클라이언트는 마찬가지로 JMF RTP API를 사용하여 서버쪽에서 방송하는 내용을 실시간으로 받아서 모니터나 스피커에 그 내용을 뿌려주게 된다. JMF RTP API란 네트웍에서 실시간 전송을 지원하는 프로토콜인 RTP(Real-time Transport Protocol)에 대해 JMF에서 지원하는 API이다. 제작하려는 프로그램의 스펙을 다시 정리하면 아래와 같다.

- ✍✍ 서버
  - ✍✍ 방송 포트를 제공한다.
  - ✍✍ 컴퓨너에 설치된 마이크를 이용하여 방송할 내용을 녹음한다.
  - ✍✍ 녹음된 내용을 방송한다.
- ✍✍ 클라이언트
  - ✍✍ 서버의 해당 포트에 연결
  - ✍✍ 서버에서 방송되고 있는 내용을 청취한다.
  - ✍✍ 그 외의 미디어 파일들을 플레이한다.

하지만 본격적인 개발은 다음 연재로 미루고, 우선 이번 연재에서는 우선 JMF의 구조를 중점적으로 다루어 볼 것이다. 그 후에는 클라이언트쪽 스펙의 3번째에 해당하는 미디어 파일 플레이 부분에 대한 프로그램을 짜 볼 것이다.

아마 이번 연재만 이해한다면 다음연재 부터는 아주 손쉽게 따라 올 수 있을 것이다.

JMF 구조를 이해하자.

아래 코드는 JMF를 사용해 미디어를 플레이하기 위한 부분 코드이다. 부분 코드라고는 했지만 실제로는 아래의 코드에 JMF의 미디어 플레이에 관한 핵심 부분이 다 들어 있다고 해도 과언이 아닐만큼 JMF는 쉽게 만들어 졌고 추상화 또한 잘 되어 있다.

```
// 미디어를 플레이 해준다.
Player              player;
URL                 mediaURL = new URL("file:/c:/dear.mpg");
MediaLocator        mrl = null;

// 해당하는 미디어가 없다면..
if ((mrl = new MediaLocator(mediaURL)) == null){

}
// 해당하는 미디어가 있다면..
else{
    DataSource src = null;

    try{
        src = Manager.createDataSource(mrl);
        player         = Manager.createPlayer(src);

        // player을 realize 시키고 Realized 상태까지 기다린다.
        player.realize();
        while(!(processor.getState() == Processor.Realized)){

        // Realized 상태에서 할 일들을 넣는다.

        // 플레이 속도를 2배로 한다.
        player.setRate(2.0);

        // player을 prefetch 시키고 Prefetched 상태까지 기다린다.
        player.prefetch();
        while(!(processor.getState() == Processor.Prefetch)){

        // Prefetch 상태에서 할 일들을 넣는다.

        // TimBase에 동기적으로 플레이 시작
        player.syncStart();
        }catch(IOException e){
        }catch(javax.media.NoDataSourceException e){
        }catch(NoPlayerException e){
        }
    }
```

이제부터 본격적으로 JMF의 전반적인 구조와 중요한 클래스들을 살펴볼 것이다. 이 부분을 처음부터 쭉 따라 보기란 다소 지루한 면이 있을 것이다. 새로 접하는 것이 다 그렇듯이, 어느 정도 적응이 될 때까지는 그것에 대한 흥미가 공부를 하는데 많은 도움을 줄 것이다. 그러나 이제부터 다루려고 하는 내용은 그 흥미를 다소 누그러트리는 부분이 될 것이다. 하지만 앞으로 나오는 내용을 그저 그대로만 보지 말고, 그것이 자신이 하려고 하는 또는, 하고 싶은 부분에 어떻게 쓰일 수 있는지? 좀더 응용해서 생각해보아 완벽히 이해하길 바란다.

기본 구조(추상적인)

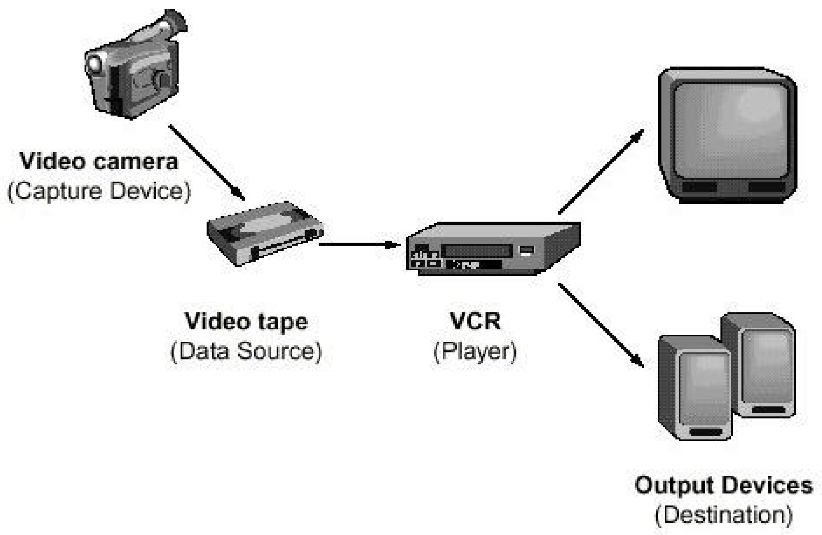


그림 2 JMF의 추상적 구조

JMF가 제공하는 기능 중에서 RTP부분만을 제외시킨다면 JMF의 구조는 거의 90%가 [그림 2]에 매치 된다고 할 수 있다. 그만큼 JMF는 추상화가 잘 되어있는 패키지 이다. 테잎(MediaSource)을 재생하기 위해선 테잎을 비디오플레이어(Player)에 넣고, Play 버튼을 누르면 된다. 그리고, 정지시키기 위해선 Stop버튼을 누르면 된다. 여기까지는 누구나 알고 있는 비디오의 사용방법일 것이다. JMF의 사용 방법 또한 이와 거의 흡사하다. 앞으로 나오는 내용을 보고 난 후에 다시 이 [그림 2]을 본다면 여러분들도 이 말에 동감할 것이다.

중심 클래스 구조

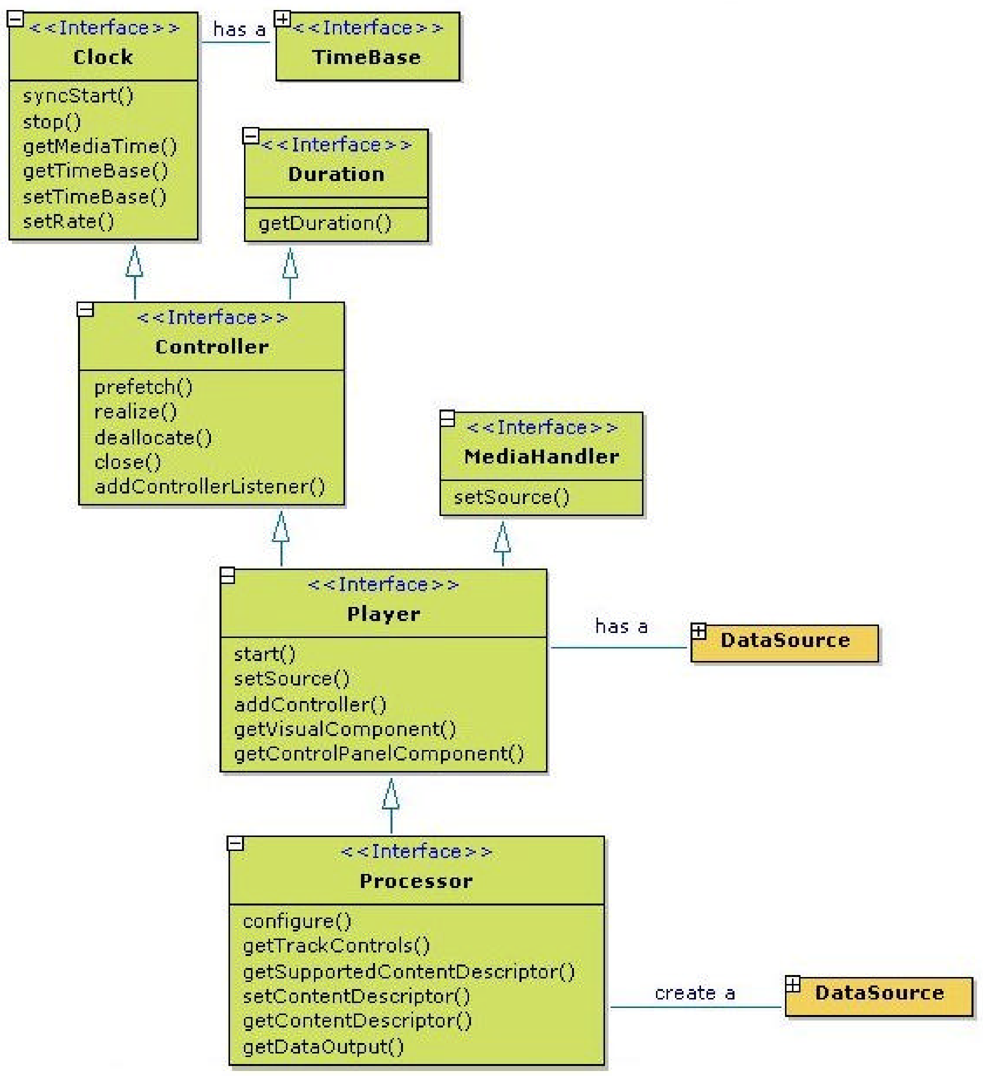


그림 3 JMF의 클래스 구조

[그림 3] 의 클래스 다이어그램은 JMF에서 핵심적인 클래스들의 관계를 나타낸 것이다. 앞으로 몇 페이지 동안 이 클래스들을 설명할 것인데, [그림 3]를 지속적으로 참고하면서 읽기 바란다.

Clock, Duration, TimeBase

JMF의 시간 관리는 TimeBase와 Clock 두 인터페이스의 동기화를 통해서 이루어 진다. Clock은 미디어 시간(Media Time)을 관리하고, TimeBase는 실제 시간(Time-Base Time)에 대응되는데, Clock은 TimeBase에서 시간을 받아 현재 미디어의 시간(정확히 말하면, Media Stream 에서의 현재 시간의 위치)을 정한다. 이때 JMF내에서 시간을 관리쪽에서 전반적으로 사용되는 Time객체를 시간 값으로 받는데, 이 객체는 나노초 단위까지 정확히 시간값을 갖고 있다. 따라서 JMF는 시간을 나노초 단위까지 정확히 관리하게 된다.

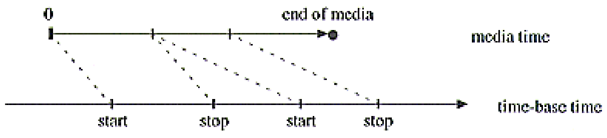


그림 4 Time-Base Time과 Media Time

앞에서 Clock의 미디어 시간 관리에 대해 잠깐 언급을 했었는데, 이에 대해 자세히 살펴보자. 미디어의 시간과 실제 시간은 [그림 4]와 같은 형태로 매핑 된다. 미디어의 시간이 멈춰 있을 때에도 여전히 실제 시간을 흘러가고 있는 것이다. 실제로 TimeBase 인터페이스를 살펴보면 시간을 받는 메소드(getNanoSecond(), getTime())를 제공할 뿐이고, 시간을 멈추거나 다시 시작시키는 조작작업을 제공하지는 않는다. Time-base시간은 거의 시스템 시간에 기반을 두고 있다.

이에 반해 미디어 시간은 마치 스톱워치처럼 시간을 멈추거나 다시 시작시킬 수 있다. 그리고, [그림 4]을 보면 알 수 있겠지만 미디어 시간이 0이라 함은 미디어 스트림의 시작위치를 말하는 것이고, 미디어 시간의 최대치는 미디어 스트림이 끝나는 시점이다. 미디어 스트림의 최대치는 Duration인터페이스를 통해 얻을 수 있다.

여기서 한가지 생각할 수 있는 것이 바로 미디어의 재생 비율이다. 만약 두 배의 속도로 미디어를 재생하고 싶다면 어떻게 해야 할까? 앞의 내용을 어느 정도 이해한 독자라면, 간단히 생각날 것이다. 바로 미디어의 시간을 두 배로 흐르게 해주면 될 것이다. 이를 위해 미디어의 시간을 관리하는 인터페이스인 Clock에서는 setRate() 메소드를 제공해 준다. 또한 getRate() 메소드를 통해 현재의 재생비율을 알 수 있다. 하지만 불행히도, 미디어 파일의 특성상 지정해준 비율로 재생을 할 수 없는 파일들도 있는데, 이런 미디어들은 setRate() 메소드를 통해 비율을 지정해 주어도 지정해준 비율로 정확히 재생해 주지 못 하고, 자체적으로 해당 비율에 최대한 가능한 비율로 재생해 주는데, 이러한 실제 재생비율은 setRate()메소드의 리턴 값으로 얻을 수 있다.

DataSource

이 클래스는 JMF 전반에서 사용되는데, 이름에서도 알 수 있듯이 미디어 자료를 나타내는 클래스다. 또한 미디어의 위치 정보, 프로토콜과 미디어의 전송에 관한 작업을 캡슐화 한다. 나중에 다루겠지만, JMF를 사용하여 캡쳐된 미디어 자료 또한 DataSource 형태로 변환되어 사용되어 진다.

DataSource는 또한 SourceStream 객체들의 집합을 관리하는데, 가장 기본적인 data source는 전송단위로는 바이트 배열을 사용한다. 이외에 Buffer객체를 전송단위로 사용하는 DataSource도 있다.

	버퍼 사용 안 함	버퍼 사용
Pull방식	PullDataSource	PullBufferDataSource
Push방식	PushDataSource	PushBufferDataSource

표 1 DataSource의 종류

이와는 별도로 JMF는 Data 전송이 어떻게 시작되어 지는가에 따라 크게 DataSource PullDataSource와 PushDataSource로 분류된다. 클라이언트 쪽에서 데이터의 전송을 초기화시키고, 데이터의 흐름을 조절하는 것을 PullDataSource라고 한다. 여러분들이 잘 알고 있는 HTTP 프로토콜이나 FILE이 Pull 방식의 Data이다. 이와는 반대로 서버쪽에서 데이터 전송을 초기화하고, 흐름을 조절하는 것을 PushDataSource라고 하는데, VOD, RTP(Real-time Transport Protocol)이 이에 해당한다.

위에서 제시했던 DataSource 이외에 JMF에서는 cloneable DataSource와 merging DataSource라는 두 가지 특별한 DataSource를 제공한다. cloneable DataSource는 SourceCloneable 인터페이스를 Implement한다. 이 인터페이스는 createClone()메소드를 정의하는데, 이 메소드를 사용하여 해당 DataSource객체를 복사할 수 있다. 이때 복사된 DataSource객체 또한 clonable DataSource이다. merging Datasource는 여러개의 DataSource를 합쳐 하나의 DataSource를 생성한다. 이때 생성된 DataSource의 Duration 값은 가장 큰(긴) Duration값을 갖는다. 한가지 주의할 점은, merging할 DataSource들 형식이 같아야 한다는 것이다. 예를 들면 PullDataSource와 PushDataSource간의 merging은 할 수 없다. 하지만 테스트해 본 결과 아직 이 부분은 제대로 돌아가지 않는 듯 했다.

Format에 대한 처리

미디어 파일은 각 파일마다의 특별한 Format을 갖고 있다. MP3와 WAVE파일은 둘다 오디오 파일이지만 각각 포맷이 달라 서로 다르게 decoding을 해야 한다. JMF에서는 이러한 포맷의 정보를 Format이라는 클래스를 사용하여 나타낸다. 그러나 이 객체는 해당 포맷에 대한 인코딩정보는 갖고 있지 않다. 단지 해당 포맷의 인코딩 이름과 타입만을 갖고 있는다.

Player ,Controller

[그림 3]에서 보듯이 Player 인터페이스는 Controller인터페이스를 상속받는데, Controller에는 JMF 전반에 걸쳐 사용되는 State를 다루는 메소드 들을 제공한다. Controller는 언제나 위의 6개의 상태중에서 하나의 상태에 속해 있는데 현재의 상태는 Controller의 int getState()메소드에 의해 알 수 있으며, 해당 상태의 값은 Controller인터페이스에 static변수로 선언되어 있다.

UnRealized상태에서 Prefetched상태까지의 5개의 상태는 크게 Stopped상태로 나타낼 수 있는데, 이는 이미 보았던 Clock인터페이스에 정의되어 있다. 그러면 두 가지 상태 정보만 갖고도 나타낼 수 있는 상태를 Player에서는 6개의 상태로 분류해 놓았을까? 간단히 말하자면 자원관리의 편리함을 위해서 라고 말할 수 있다. 미디어를 재생하기 위하여 자원을 수집하는 과정은 1차적으로 컴퓨터 내부적으로 미디어 재생을 위한 자원들을 수집하고, 2차적으로는 미디어를 재생하기 위한 하드웨어에 대한 자원을 얻는 과정들을 거치게 된다. 이 과정을 얻기 전, 얻는 중, 얻은 후 의 상태들로 나눈 결과 이와 같은 상태들이 분류된 것이다.

Controller의 6개 상태를 살펴보자.

Unrealized

Player 객체가 처음 생성되었을 때 갖는 상태이다. 이 상태에서는 아직 미디어에 대한 아무런 정보도 갖고 있지 않다.

Realizing

Unrealized 상태에서 realize 메소드가 호출되었을 때 갖는 상태이다. 여기서 주의할 점은 Unrealized상태에서 realize메소드 호출 후 바로 Realized상태로 전이되지 않는다는 점이다. realize메소드는 비동기적으로 실행되기 때문이다.
이 상태에서는 랜더링시에 필요한 자원을 계산하여 얻는다. 이때 독점적인 자원(Exclusive-use resource)들은 제외 되는데, 독점적인 자원이란, 한번에 오직 하나의 Player만이 사용할 수 있는 Hardware장치같은 제한된 자원을 말한다. 독점적 자원은 prefetching 단계에서 얻어진다.

Realized

realizing 작업이 끝났을 때 갖는 상태이다. 이 상태에서는 보여줄 미디어에 대한 정보를 알고 있다. 따라서 필요한 컴포넌트들을 제공한다. getControlPanelComponent()메소드를 통해 미디어의 재생을 컨트롤 할 수 있는 컴포넌트를 얻을 수 있다. 또한 재생하려는 미디어가 비디오를 포함하고 있다면, getVisualComponen11t()메소드는 널 값이 아닌 비디오를 보여줄 컴포넌트를 넘겨준다.

Prefetching

Realized상태에서 prefetch메소드가 불렸을 때 갖는 상태이다. 이 상태에서는 미디어를 보여주기 위한 준비를 한다. 그러기 위해 미디어 데이터를 읽어드리고, 독점적(Exclusive)인 자원 또한 얻는다. 이 단계는 재생중인 미디어의 위치(시간적)가 바뀌었거나, 재생 비율이 바뀌거나 할 때에도 거치는 단계이다.

Prefetched

Prefetching단계가 끝났을 때 갖는 상태이다. 이 단계에서는 모든 준비를 마치고 플레이를 시작하기만을 기다리는 단계이다.

Started

미디어를 플레이하고 있는 상태다.

예제 1 : MediaPlayer

[소스코드 1]

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.media.*;
import javax.media.protocol.*;
import java.net.*;
import java.io.*;

public class MediaPlayer extends JFrame implements ControllerListener, ActionListener{
   public static final String OPEN_MENU_ACTION_COMMAND     = "Open";
   public static final String EXIT_MENU_ACTION_COMMAND     = "Exit";
   public static final String RATE_1_ACTION_COMMAND        = "1 배속";
   public static final String RATE_2_ACTION_COMMAND        = "2 배속";
   public JFileChooser fileDlg = new JFileChooser();

   // 미디어를 플레이할 플레이어
   Player      player = null;
   // 현재 재생중인 미디어의 URL
   String      mediaURLString = null;
   float       playRate = (float)1.0;

   // 비디오가 플레이될 컴포넌트
   Component visualComponent = null;
   // 미디어에 대한 컨트롤(시작,정지 등.)을 제공하는 컴포넌트
   Component controlComponent = null;

   // visualComponent를 포함할 프레임
   JFrame displayFrame;

   public MediaPlayer(){
      super("Media Player");
```

```
        // Swing과 같이 동작하게 하기 위해
        // LightWeight Renderer으로
        Manager.setHint(Manager.LIGHTWEIGHT_RENDERER, new Boolean(true));
        initComponent();
        this.pack();
        this.show();
    }
    public void initComponent(){
        JMenuBar mainMenuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");

        fileMenu.add(makeMenuItem(OPEN_MENU_ACTION_COMMAND));
        fileMenu.addSeparator();
        fileMenu.add(makeMenuItem(EXIT_MENU_ACTION_COMMAND));
        mainMenuBar.add(fileMenu);

        JMenu rateMenu = new JMenu("Rate");

        rateMenu.add(makeMenuItem(RATE_1_ACTION_COMMAND));
        rateMenu.add(makeMenuItem(RATE_2_ACTION_COMMAND));
        mainMenuBar.add(rateMenu);

        this.setJMenuBar(mainMenuBar);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                exit();
            }
        });
    }

    // ------------------------------------------------------------------------
    private JMenuItem makeMenuItem(String commandString) {
        JMenuItem item = new JMenuItem(commandString);
        item.addActionListener(this);
        return item;
    }

////////////////////////////////////////////////////////////////////////////////
// ActionListener 구현 부분
    // ------------------------------------------------------------------------
    public void actionPerformed(ActionEvent e){
        String actionCommand = e.getActionCommand();
        if      (actionCommand.equals(OPEN_MENU_ACTION_COMMAND))    {
            openMediaFile();
        }else if(actionCommand.equals(EXIT_MENU_ACTION_COMMAND))    {
            exit();
        }else if(actionCommand.equals(RATE_1_ACTION_COMMAND))   {
            setPlayRate(1);
        }else if(actionCommand.equals(RATE_2_ACTION_COMMAND))   {
            setPlayRate(2);
        }
    }
    // -----------------------------------------------------------------------
    private void openMediaFile(){
            if(fileDlg.showOpenDialog(this) == JFileChooser.APPROVE_OPTION){
            try{
                        startMedia(fileDlg.getSelectedFile().toURL());
            }catch(Exception e){
            }
        }
    }
    // -----------------------------------------------------------------------
    private void exit(){
        System.exit(-1);
    }
    // -----------------------------------------------------------------------
    private void setPlayRate(float rate){
        this.playRate = rate;
        if(player != null){

            if(!(player.getState() == Controller.Unrealized ||
                 player.getState() == Controller.Realizing))
                player.setRate(playRate);
        }
    }

///////////////////////////////////////////////////////////////////////////////
    // -----------------------------------------------------------------------
    private void startMedia(URL mediaURL){
            if(player != null){
                        player.stop();
```

```
                    player.deallocate();
                    player.close();
                    player = null;
            }
            try {
               // 지정된 미디어(mediaURL)를 위한 Player의 인스턴스를 생성시킨다.
               try {
                    player = Manager.createPlayer(getDataSource(mediaURL));
               } catch (NoPlayerException e) {
              fatal("지정하신 화일이 없습니다");
               }
               // Add ourselves as a listener for a player's events
               System.out.println("미디어를 시작시킨다.");
               player.addControllerListener(this);
               player.start();
            } catch (MalformedURLException e) {
               fatal("Invalid media file URL!");
            } catch (IOException e) {
            }
    }
///////////////////////////////////////////////////////////////////////////////
// ControllerListener 구현부분

public synchronized void controllerUpdate(ControllerEvent e) {
            System.out.println("업테이트");
            if (player == null)
               return;
            if (e instanceof RealizeCompleteEvent) {
            System.out.println("RealizeCompleteEvet - 컴포넌트들을 넣는다.");
           float real = player.setRate(playRate);

               JPanel pane = new JPanel();
               pane.setLayout(new BorderLayout());
                    if (( controlComponent =
                         player.getControlPanelComponent()) != null) {
                        pane.add(controlComponent, BorderLayout.SOUTH);
                    }
                    if (( visualComponent =
                         player.getVisualComponent())!= null) {
                       pane.add(visualComponent, BorderLayout.CENTER);
                    }
            if (controlComponent != null) {
                    controlComponent.invalidate();
            }
            if(displayFrame == null)
              displayFrame = new JFrame();

           displayFrame.setTitle(mediaURLString);
               displayFrame.setContentPane(pane);
               displayFrame.doLayout();
               displayFrame.pack();
               displayFrame.show();
            } else if (e instanceof EndOfMediaEvent) {
               System.out.println("EndOfMediaEvent.");
               player.setMediaTime(new Time(0));
               player.start();
            } else if (e instanceof CachingControlEvent) {
               System.out.println("CachingControlEvent");
               player.setMediaTime(new Time(0));
               player.start();
            } else if (e instanceof ControllerErrorEvent) {
               System.out.println("ControllerErrorEvent");
               player = null;
               fatal(((ControllerErrorEvent)e).getMessage());
        } else if (e instanceof ControllerClosedEvent) {
               System.out.println("ControllerClosedEvent");
            }
    }
    private DataSource getDataSource(URL mediaURL){
           mediaURLString = mediaURL.toString();
            MediaLocator mrl = null;

            if ((mrl = new MediaLocator(mediaURL)) == null){
           fatal("지정하신 화일이 없습니다");
           return null;

        }else{
           DataSource src = null;
           try{
              src= Manager.createDataSource(mrl);
           }catch(IOException e){
            }catch(javax.media.NoDataSourceException e){
            }
            return src;
        }
    }

    void fatal (String s) {
             System.err.println("FATAL ERROR: " + s);
             throw new Error(s);
    }

    public static void main(String args[]){
        new MediaPlayer();
    }
}
```

이제 어느 정도 기본 지식에 대한 설명은 끝났다. 다소 열거 식이라 쭉 읽기는 힘들었을 것이다. 하지만 앞 부분을 잘 이해하였다면 이제부터는 고개를 끄덕이며 쭉 훑어 내려가기만 하면 된다.

그러면 이제부터 본격적으로 프로그래밍으로 들어가 보겠다. 만들어 볼 프로그램은 간단한 '미디어 재생기'이다. 이 프로그램의 실제 코드 길이는 200라인 정도뿐이 안되지만, 거의 모든 종류(재생가능한 미디어 참조)의 미디어 파일을 지원할 정도로 기능은 막강하다.

미디어를 재생하기 위해선, 미디어 자료확보, 컨트롤 컴포넌트(비디오 자료일 경우 비주얼 컴포넌트 포함) 획득, 재생 순의 단계을 거치는데 이를 각 단계별로 살펴보기로 하자.

DataSource 생성

DataSource클래스는 아래와 같은 방법으로 생성 할 수 있다. DataSource의 생성자에 MediaLocator클래스가 인수로 들어가는데, MediaLocator는 미디어 자료에 대한 위치 정보를 나타낸다. MediaLocators는 URL로부터 생성된다. 따라서 MediaLocator로부터 URL을 얻는 것도 가능하다. 그러나 URL과는 달리 URLStreamHandler이 시스템에 설치되지 않아도 생성될 수 있다. 따라서 MediaLocator는 해당 파일에 대한 단순한 위치정보라고 볼 수 있다.

```
MediaLocator mrl = new MediaLocator(mediaURL);
DataSource   mediaSource = new DataSource(mrl);
```

Player 생성

Player의 인스턴스는 Manager클래스의 아래의 메소드를 사용하여 생성할 수 있다.

```
static Player  Manager.createPlayer(java.net.URL sourceURL)
static Player  Manager.createPlayer(MediaLocator  sourceLocator)
static Player  Manager.createPlayer(DataSource source)
```

첫 번째 생성자는 단순히 넘겨진 URL을 갖고 MediLo cator을 생성하여 두 번째 생성자에게 넘겨버린다. 두 번째와 세 번째 메소드들은 상당히 복잡한 알고리즘을 통해 Player을 생성해 낸다. JMF의 해당 메소드 설명 부분에 알고리즘이 나와 있으니 관심있는 독자는 찾아보기 바란다.

미디어 재생

앞에서 생성된 Player객체는 Unrealized 상태이다. 미디어를 재생 시키려면 Player를 Started상태로 변경해야 한다. 방법은 두 가지가 있는데, 하나는 Player.start()메소드를 사용하는 방법이고, 나머지 하나는 realize(), prefetch() 메소드를 호출하여 상태를 차례로 변경시키는 방법이다. 앞에서도 언급했지만 Controller 인터페이스는 상태가 변할 때 마다 ControllerEvent를 발생시킨다(발생되는 이벤트는 [그림 6] 참조). Player.start()메소드를 사용하는 방법을 선택하였다면, ControllerEvent를 받는 ControllerListener를 Player인스턴스에 추가하여 상태 변화를 추적할 수 있다. 따라서 ControllerListener(ControllerUpdate 메소드)에서 발생한 이벤트를 분석하여 상태에 따른 작업을 지정해 줄 수 있다. 이벤트은 아래와 같이 instanceof 연산자를 사용하여 분석할 수 있다.

```
if (e instanceof RealizeCompleteEvent) {
            ....
} else if (e  instanceof PrefetchCompleteEvent){
            ....
} else if (e  instanceof ControllerClosedEvent) {
            ....
```

컴포넌트 얻기

Realized 상태에서는 Player객체을 통해 미디어의 재생을 조절하는 컨트롤 컴포넌트를 얻을 수 있다. 만약 미디어가 비디오정보도 갖고 있다면 화상정보를 보여주는 비주얼 컴포넌트도 얻을 수 있다. 얻어진 컴포넌트를 자신의 GUI에 보기 좋게 배열만 시켜 놓으면, 미디어를 재생하기 위한 모든 기본적인 작업은 끝났다. 이제 Started상태로만 변경되기만을 기다리면 된다.

```
player.getControlPanelComponent()
player.getVisualComponent()
```

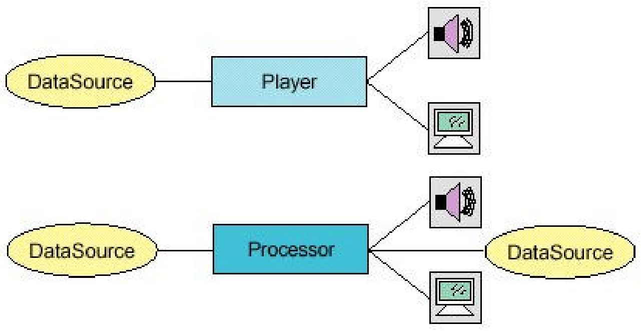
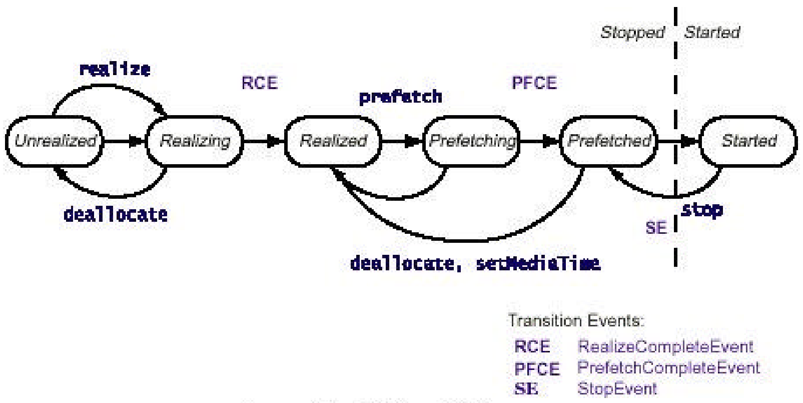
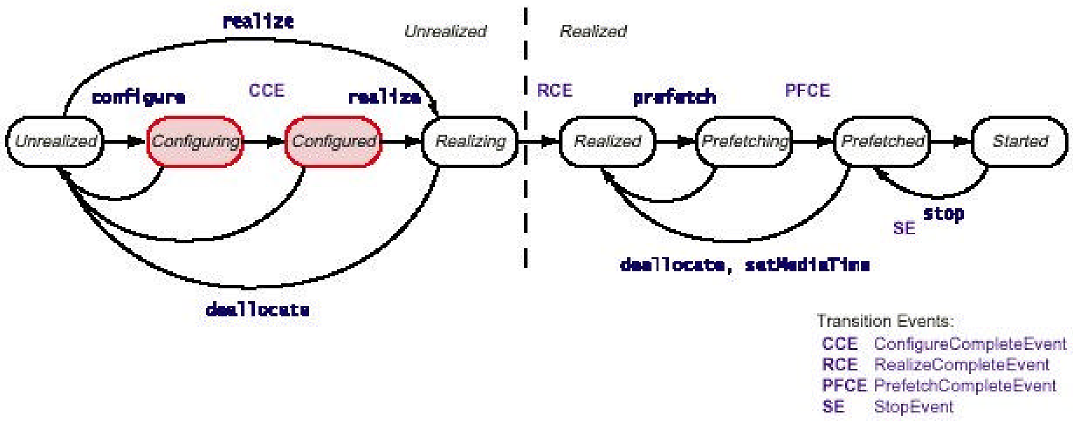


그림 5 Player와 Processor의 Model 비교



player의 State 변화



processor의 State 변화

그림 6 Player와 Processor의 State변화 비교

Method	Unrealized Player	Realized Player	Prefetched Player	Started Player
addController	NRE	적합	적합	CSE
deallocate	적합	적합	적합	CSE
getControlPanelComponent	NRE	적합	적합	적합
getGainControl	NRE	적합	적합	적합
getStartedLatency	NRE	적합	적합	적합
getTimeBase	NRE	적합	적합	적합
getVisualComponent	NRE	적합	적합	적합
mapToTimeBase	CSE	CSE	CSE	적합
removeController	NRE	적합	적합	CSE
setMediaTime	NRE	적합	적합	적합
setRate	NRE	적합	적합	적합
setStopTime	NRE	적합	적합	이전시간 입력시 StopTimeSetError
setTimeBase	NRE	적합	적합	CSE
syncStart	NRE	NPE	적합	CSE

<표 2> Player의 상태별 메소드 호출 제약

NRE – NotRealizedError

NPE - NotPrefetchedError

CSE - ClockStartedError

Method	Unrealized Processor	Configuring Processor	Configured Processor	Realized Processor
getControls	적합	적합	적합	적합
getDataOutput	NRE	NRE	NRE	적합
getContentDescriptor	NCE	NCE	적합	적합
getSupportedContentDescriptors	적합	적합	적합	적합
getTrackControls	NCE	NCE	적합	FCE
realize	적합	적합	적합	적합
setContentDescriptor	NCE	NCE	적합	FCE

<표 3>Processor의 상태별 메소드 호출 제약

NRE – NotRealizedError

NCE - NotConfiguredError

FCE - FormatChangeException

Processor

이제 [그림 3]에서 아직 설명하지 않은 Processor인터페이스를 살펴보기로 한다. Processor도 Player와 마찬가지로 미디어 데이터를 보여줄 수 도 있다. [그림 3]에서 보듯이 Player인터페이스를 상속하기 때문이다. 하지만, Processor은 내부적으로 입력된 미디어 스트림을 처리하는 과정을 한번 거친다. [그림 5]에서 보듯이 Processor은 Player와는 다르게 새로운 DataSource를 Output으로 생성할 수 있다. 이것은 Processor내에서 사용자가 정의한 처리 과정을 거쳐 생성된 결과물(새로운 DataSource)을 다른 플레이어나, Processor들이 이용할 수 있도록 제공하는 기능이다. 또한 이 결과물은 다른 전달과정을 거쳐 네트워크로 보내지거나 파일에 저장되어질 수도 있다.

다음은 Processor과 관련된 사항들을 정리한 것이다.

- ✍✍ ProcessorModel을 이용하여 특정 형식의 입력과 출력을 갖는 Processor을 만들 수 있다.
- ✍✍ TrackControl 의 setFormat메소드를 이용하여 해당 트랙의 format변환을 명시한다.
- ✍✍ Processor의 setOutputContentDescriptor메소드를 이용하여 Multiplex된 데이터 포맷을 명시한다.
- ✍✍ TrackControl의 SetCodecChain메소드를 이용하여 Processor에서 사용할 Effect나 Codec Plug-in을 선택할 수 있다.
- ✍✍ TrackControl의 setRender메소드를 사용하여 Processor에서 사용할 Render을 선택할 수 있다.

Processor의 내부 처리과정은 다음과 같은 순서로 처리된다. 우선 미디어 데이터를 트랙별로 나눈후에 각각의 트랙을 해당하는 포맷에 맞추어 디코딩 한다. 그런 후에 지정된 포맷으로 인코딩하는 과정을 거친다. 각각의 디코딩 인코딩 과정에는 정해진 Effect 알고리즘을 적용시킨다.

JMF plug-in

JMF는 아래와 같은 5가지의 Plug-in 기본 인터페이스를 제공한다. 이들 인터페이스 하나하나가 모두 미디어 데이터를 직접 다루는데 사용되는 인터페이스들이기 때문에 이러한 인터페이스 제공은 JMF 사용자들에게 상당한 유연성을 제공해 준다. 만약 여러분이 새로운 미디어 포맷을 개발하였다면 그에 대한 처리는 아래의 인터페이스만 구현하여 넣어주면(Plug In)끝난다. 이러한 작업은 PlugInManager 클래스의 addPlugIn메소드를 통해 이루어 지는데, 그러고 나면 Player이나 Processor들을 통해 해당 미디어를 재생할 수도 있고, 아니면, 다른 파일로 변환하거나, 네트워을 통해 전송 또한 가능해 진다.

- ✍✍ Demultiplexer - 입력 스트림을 파싱한다. 여러개의 트랙을 갖고 있는 스트림을 분리 해주는 작업을 한다. 예를 들면, mpeg를 demultiplexing하면 비디오와 오디오트랙으로 나눠주는 일이 Demultiplexer가 하는 작업이다.
- ✍✍ Multiplexer - Demultiplexing과는 반대로 여러 개의 트랙들을 하나의 출력 스트림으로 변환하는 작업을 한다.
- ✍✍ Effect - 해당 트랙에 대해 특별한 효과를 나타내는 처리를 한다. Codec 인터페이스를 상속한다. (TrackControl 참조)
- ✍✍ Codec - 데이터를 인코딩과 디코딩을 한다. 이 인터페이스는 입력데이터형과 출력데이터형 모두 Buffer객체를 사용한다. 따라서 이 인터페이스를 구현하려면 Buffer를 참고하기 바란다. (TrackControl 참조)
- ✍✍ Renderer - 미디어 데이터를 처리하여 최종적으로 화면이나 스피커로 전달하는 일을 한다.

TrackControl

트랙에 대한 처리과정을 조정하는 인터페이스이다. Processor.getTrackControl메소드를 통해 얻을 수 있다. 이 인터페이스를 사용하여 원하는 Effect나 Codec을 적용시킬 수 있다.

또한 TrackControl을 통해 할 수 있는 일 중 주목할 점은 포맷의 변환 기능이다. TrackControl인터페이스는 FormatControl인터페이스를 상속받았는데, FormatControl인터페이스의 setFormat()메소드를 통해 포맷의 변경이 가능하다. 여기서 말하는 포맷의 변경은 출력물의 포맷 변경을 말하는 것이지 원래 DataSource의 포맷이 변경을 말하는 것은 아니다. 따라서 원래 파일의 형식이 변경되는 것은 아니다. 이와는 다르게 포맷을 변경할 수도 있는데, 바로 ProcessorModel을 사용하는 방법이다. ProcessorModel을 사용하여 Processor의 출력형식을 지정해 줄 수 있다. ProcessorModel을 정의한 후에는 Manager.createRealizedProcessor()메소드를 사용하여 모델에 맞는 Processor을 생성할 수 있다. 이름에서도 알 수 있듯이 이 메소드는 Realized상태의 Processor을 생성하는데, Realized상태에서는 포맷 변경이 불가능하기 때문이다. 따라서 지정된 ProcessorModel에 맞는 Processor만이 끝까지 남게된다.

Data의 저장& 전송

DataSink인터페이스는 출력장치가 아닌 주로 다른 곳으로의 미디어 데이터 전송에 사용된다. 주로 파일이라던가 네트웍으로의 전송을 위해 사용되는데, 다음 연재에 다루게 될 RTP에서도 사용되게 된다.

필자는 DataSink에 대해 얼핏 듣고 나서 이 인터페이스를 써 보려고 JMF스펙을 찾아 보았다. 딱 보니 인터페이스라서, 어디서 생성을 할까? 하고 여러 군데를 뒤지다 혹시나 하고 보았던 것이 Manager클래스이었다. 역시 Manager클래스는 DataSink를 생성하는 메소드 또한 제공하고 있었다. JMF의 Manager클래스는 그 이름답게 참 많은 일을 해내고

있었다. 만약, 여러분들도 혹시나 이와 비슷한 궁금증이 생긴다면 꼭 Manager클래스를 먼저 찾아보기 바란다.

예제2 :MP3 추출기

앞에서 미디어의 트랙 분리. 포맷 변환. 등등 기본적인 기능들을 다루어 보았다. 앞에서 다룬 내용은 정말 기본적인 것이지만, 이러한 내용을 바탕으로 많은 것들을 응용해 낼 수 있어야 한다. 적어도 필자의 의도는 그러했었다.

이번에 만들어보려고 하는 프로그램은 그 응용 중에서 한가지다. mpg 파일을 입력으로 받고, mp3를 출력으로 하는 프로그램이다. 여기서 입력과 출력은 모두 파일을 통해 일어난다. 간단히 말하자면, mpg 파일에서 오디오트랙만 추출하여 mp3파일로 저장하는 프로그램이다.

이 예제를 단계별로 살펴보자.

Processor 생성

Processor을 생성하려면 Player과 마찬가지로 해당 미디어에 대한 MediaLocator를 Manager클래스의 createProcessor메소드에 넘겨주면 된다. 이 예제는 실행시에 인수로 입력 파일과 출력할 파일의 이름을 받는다. 입력파일은 URL형식의 스트링으로 받게 되는데, 이 값을 갖고 URL과 MediaLocator을 차례로 생성한다. 그 후 Manager의 createProcessor(MediaLocator ml) 메소드를 이용하여 Processor객체를 얻는다.

```
// 해당 URL에 맞는 Player 을 만든다.
processor = Manager.createProcessor(mrl);
```

StateHelper 생성

이 클래스는 JMF의 예제 클래스 중 하나인데, 이름 그대로 Controller나 Player, 그리고 Processor들의 State 변환과정을 도와주는 클래스다. 이번 예제에서는 Processor의 상태 변화에 사용된다. 이 클래스의 소스코드는 [소스코드 2]에 있으니 참조하기 바란다.

Configure

Processor또한 생성되자 마자 갖는 상태는 Unrealized 상태이다. 따라서 미디어에 대한 처리 작업을 하려면 다음의 상태로 넘어가야 한다. ([그림 6] 참조 configure단계에 대한 설명은 realize이나 prefetch단계와 비슷하기 때문에 생략.)

StateHelper을 사용하면 아래와 같이 할 수 있다.

```
if (!sh.configure(10000)){
    System.exit(-1);
}
```

StateHelper 클래스를 쓰지 않고 한다면 아래와 같을 것이다.

```
processor.configure();
// configure 될때까지 기다린다.
while(!(processor.getState() == Processor.Configured)){}
```

트랙 분리& 프로세싱

이제 입력된 mpg 파일에서 트랙을 분리하는 작업을 할 수 있는 상태(Configured 상태)가 되었다. Processor은 트랙에 대한 조작을 지원하는 TrackControl을 제공해 준다. getTrackControls 메소드를 사용하여 미디어에 대한 TrackControl들을 얻자.

```
TrackControl[] trackControls = processor.getTrackControls();
```

얻어진 TrackControl배열은 미디어가 갖고 있는 모든 트랙에 대한 컨트롤들이다. 따라서 현재 얻어진 값은 Mpeg 파일에 맞는 오디오 트랙과 비디오 트랙 두 가지 일 것이다. 이 중에서 비디오 트랙은 setEnable메소드를 사용해 제거해 버리자.

```
}else if(trackControls[i].getFormat() instanceof VideoFormat){
        trackControls[i].setEnabled(false);
}
```

이제 오디오 트랙 하나만 남았다. 이 오디오 트랙을 mp3 형식으로 변환한다.

```
trackControls[audioTrackIndex].setFormat
                              (new AudioFormat(AudioFormat.MPEGLAYER3));
```

마지막으로 최종적(Multiplex 된)인 출력 파일의 형식을 지정해 준다.

```
processor.setContentDescriptor
                 (new FileTypeDescriptor(FileTypeDescriptor.MPEG_AUDIO));
```

이제 우리가 원하는 Processor을 만들기 위한 작업은 모두 끝났다. 이렇게 만들어진 Processor은 오디오 트랙만을 가진 mp3형식의 미디어 자료를 출력할 것이다.

미디어 저장

환된 미디어 자료를 저장하는 일만 남았다. 이런 일을 해 주는 것은 앞에서도 언급했듯이 DataSink라는 인터페이스이다. 이 인터페이스의 객체는 Manager 클래스를 통해 생성할 수 있는데, 이때 운반할 미디어 데이터와, 그 목적지를 넘겨준다.

```
sink = Manager.createDataSink(processor.getDataOutput(), dest);
```

목적지와의 연결을 맺는다.

```
sink.open();
```

목적지로의 데이터 전송을 시작한다.

```
sink.start();
```

이제 Processor을 미디어의 시간만큼만 재생시키면 자동적으로 변환된 미디어가 지정한 파일에 저장될 것이다.

```
sh.playToEndOfMedia((int)(processor.getDuration().getSeconds()) * 1000);
```

연결을 종료한다.

```
sh.close();
sink.close();
```

다시한번 과정을 짚어보면, 우선 Processor은 앞에서 비디오 트랙부분을 제거한후 오디오 트랙을 mp3형식에 맞게 변환하여 출력해 준다. 그러면, DataSink는 Processor의 출력값을 받아서 지정한 위치에 저장해 주게 된다. 여기에서 DataSink의 목적지를 파일이 아닌 네트워크로 설정하면 바로 인터넷 방송이 된다.

[소스코드 2]
testConvert 클래스

```
import javax.media.*;
import java.net.*;
import java.io.*;
import javax.media.control.*;
import javax.media.format.*;
import javax.media.protocol.*;

public class testConvert{
   public static void main(String args[]){

      if(args.length < 2 ){
         System.out.println("Usage: java testConvert <URL> <Destination>");
         System.out.println("ex) java testConvert file:/c:/music/love.mpg love.mp3");
         System.exit(-1);
      }

      URL mediaURL = null;
      URL destURL  = null;
      try{
         mediaURL = new URL(args[0]);
         destURL  = new URL(mediaURL.getProtocol(), mediaURL.getHost(), args[1]);

      }catch(MalformedURLException e){
         e.printStackTrace();
      }
      Processor processor;
      StateHelper sh = null;
     try {
         MediaLocator mrl = null;
         if ((mrl = new MediaLocator(mediaURL)) == null){
            return;
         }

   // 해당 URL에 맞는 Player 을 만든다.
         processor = Manager.createProcessor(mrl);
         sh = new StateHelper(processor);
         if (!sh.configure(10000)){
            System.exit(-1);
         }
```

```
        TrackControl[] trackControls = processor.getTrackControls();
        int audioTrackIndex = 0;
        for(int i=0; i<trackControls.length; i++){
            if(trackControls[i].getFormat() instanceof AudioFormat){
                audioTrackIndex = i;
            }else if(trackControls[i].getFormat() instanceof VideoFormat){
                trackControls[i].setEnabled(false);
            }
        }

        trackControls[audioTrackIndex].setEnabled(true);
        trackControls[audioTrackIndex].setFormat(new
                     AudioFormat(AudioFormat.MPEGLAYER3));
        processor.setContentDescriptor(new
                     FileTypeDescriptor(FileTypeDescriptor.MPEG_AUDIO));
        if (!sh.realize(10000)){
            System.exit(-1);
        }
        DataSink sink = null;
        MediaLocator dest = new MediaLocator(destURL);
        try{
            sink = Manager.createDataSink(processor.getDataOutput(), dest);
            sink.open();
            sink.start();
        }catch(Exception e){
            e.printStackTrace();
        }
        // 미디어의 시간까지 플레이
        sh.playToEndOfMedia((int)(processor.getDuration().getSeconds()) * 1000);
        sh.close();
        sink.close();
        System.exit(-1);

    } catch (NoProcessorException e){
    } catch (MalformedURLException e) {
    } catch (IOException e) {
    }
  }
```

StateHelper 클래스

```
import javax.media.*;

public class StateHelper implements javax.media.ControllerListener {
    Player player = null;
    boolean configured = false;
    boolean realized = false;
    boolean prefetched = false;
    boolean eom = false;
    boolean failed = false;
    boolean closed = false;

    public StateHelper(Player p) {
        player = p;
        p.addControllerListener(this);
    }

    public boolean configure(int timeOutMillis) {
        long startTime = System.currentTimeMillis();
        synchronized (this) {

        if (player instanceof Processor)
            ((Processor)player).configure();
        else
            return false;

        while (!configured && !failed) {
            try {
                wait(timeOutMillis);
            } catch (InterruptedException ie) {
            }

            if (System.currentTimeMillis() - startTime > timeOutMillis)
            break;
            }
        }
        return configured;
    }

    public boolean realize(int timeOutMillis) {
        long startTime = System.currentTimeMillis();

        synchronized (this) {
            player.realize();

            while (!realized && !failed) {
            try {
                wait(timeOutMillis);
            } catch (InterruptedException ie) {
            }
                if (System.currentTimeMillis() - startTime > timeOutMillis)
                break;
            }
        }
        return realized;
    }

    public boolean prefetch(int timeOutMillis) {
        long startTime = System.currentTimeMillis();
        synchronized (this) {
            player.prefetch();

            while (!prefetched && !failed) {
            try {
                wait(timeOutMillis);
            } catch (InterruptedException ie) {
            }
                if (System.currentTimeMillis() - startTime > timeOutMillis)
                break;
            }
        }
        return prefetched && !failed;
    }

    public boolean playToEndOfMedia(int timeOutMillis) {
        long startTime = System.currentTimeMillis();
        eom = false;

        synchronized (this) {
            player.start();
            while (!eom && !failed) {
                try {
                    wait(timeOutMillis);
                } catch (InterruptedException ie) {
                }
                if (System.currentTimeMillis() - startTime > timeOutMillis)
                break;
            }
        }
        return eom && !failed;
    }

    public void close() {
        synchronized (this) {
            player.close();

            while (!closed) {
                try {
                    wait(100);
                } catch (InterruptedException ie) {
                }
            }
        }
        player.removeControllerListener(this);
    }

    public synchronized void controllerUpdate(ControllerEvent ce) {
        if (ce instanceof RealizeCompleteEvent) {
            realized = true;
        } else if (ce instanceof ConfigureCompleteEvent) {
            configured = true;
        } else if (ce instanceof PrefetchCompleteEvent) {
            prefetched = true;
        } else if (ce instanceof EndOfMediaEvent) {
            eom = true;
        } else if (ce instanceof ControllerErrorEvent) {
            failed = true;
        } else if (ce instanceof ControllerClosedEvent) {
            closed = true;
        } else {
            return;
        }
        notifyAll();
    }
}
```

이번 연재에서는 JMF의 기본적인 요소들 하나 하나를 될 수 있는 한 아주 자세히 다루어 보았다. 이 부분을 SUN 사의 다른 확장 패키지들의 스펙에 비교 한다면 가장 앞부분에 해당될 것이다. 먼저 해당 패키지의 구조를 이해시키기 위한 자리 부분이 앞부분에 위치하는데, 해당 패키지에 대한 지식을 확장시킬 수 있는 기반을 마련해 주는 곳이기도 하다.

최근에는 노하우('Know How')보다는 노웨어('Know Where')가 더 부곽이 되고 있는데, 이번 연재의 목표는 바로 JMF에 대한 'Know Where‘의 기반을 닦는데 목표를 두었다. 만약 다른 확장 패키지들을 공부하려 하는 독자가 있다면, 이러한 방법으로 접근하기를 필자는 권유한다.

다음 연재에서는 JMF의 Capture와 RTP부분을 살펴 볼 것이다.

JMF와 JDK의 궁합 맞추기

독자들 중에 JMF2.0을 설치 해본 사람이라면, 바탕화면에 등록되는 JMStudio라는 프로그램을 실행 해 보았을 것이다. 필자의 경우에는 JMStudio의 캡춰 기능(File -> Capture)실행시 항상 잘못된 연산이 일어나거나, 아니면 'Couldn't initalize the Capture Device'라는 메시지를 얻었었다. JMF2.0 Early Access 버전에서부터 이 기능을 실행 하려고 하면 이와 비슷한 문제가 생겼었는데, 아무튼 여러 가지 방법을 써 봐도 결과는 마찬가지였다.

그에 대한 해결책을 찾으려고 선 사이트를 뒤지던 도중 비슷한 문제에 대한 질문을 발견할 수 있었고, 며칠 후 그에 대한 개발자의 답변이 올라왔다.

답변에서는 그에 여러 가지 체크해볼 사항을 올려 주었는데, 그 중에 살펴볼 필요가 있는 체크 사항들의 원문과 내용은 아래와 같았다.

1. Did you un-install all previous versions of JMF on your machine?
   (이전 버전의 JMF를 모두 un-install 했느냐?)

2. Are your running the all-java version of JMF? it does not support audio capture.
   (all-java version 의 JMF에서는 (audio)capture 기능을 지원하지 않는다.)

3. Are you using JMF with JDK 1.3 beta? JMF audio(via JavaSound) is not compatible with JDK 1.3 beta
   (JDK 1.3에서 JMF를 사용하는가? JMF(JavaSound를 거쳐) 는 JDK와 호환되지 않는다.)

당시에 필자는 JDK1.3 beta 버전을 사용하고 있었기 때문에 3번 체크사항을 보는 순간 드디어 이 문제가 해결되는가 싶었었다. 곧 바로 JDK1.3 beta버전을 un-install하고 JDK 1.2.2을 설치하고 기쁜 마음으로 JMStudio를 실행 시켰지만, 결과는 이전과 같았다. 하지만 혹시나 하는 마음에 마지막으로 JDK 1.1.8 버전에서 테스트를 한번 해 보았는데, 드디어 성공했었던 것이었다.

결국 JMF는 Capture기능에 한해선 JDK1.3 beta뿐만 아니라 JAVA 2 환경하고도 제대로 호환되지 않는다. 하지만 모든 기능을 테스트 해본 것은 아니지만 그 외의 기능은 1.0이상의 모든 JDK버전에서 제대로 돌아갔다.

성능 향상을 위한 가비지 컬렉터(Garbage Collector) 사용

가비지 컬렉터는 C++ 같이 인터를 사용해 객체를 다루는 과정중에서 생성할 수 있는 Dangling Reference(잘못된 참조)문제를 근본적으로 해결할 수 있는 방안이다. 이들 언어에서는 명시적으로 객체 삭제 방법을 프로그래머에게 제공하기 때문에 하나의 객체에 대한 포인터가 몇 개가 있을지 모르는 상황에서 해당 객체를 어디선가 삭제해 버리면, 남아있는 포인터들은 잘못된 참조 값을 갖고 있게 된다. 이때 남아있는 참조 값들이 Dangling Reference들이다.

자바에서는 객체 삭제 기능을 명시적으로 제공해 주지는 않는 대신 가비지 컬렉터를 제공 하는데, 가비지 컬렉터는 간단히 설명하면, 레퍼런스가 없는 객체들이 차지하고 있는 메모리를 사용할 수 있도록 시스템 자원으로 되돌리는 역할을 한다. 이러한 작업을 가비지 컬렉션(Garbage Collection)이라고 하는데, 가비지 컬렉터는 어플리케이션의 Background에서 어플리케이션과는 비동기 적으로 돌아가며 가비지 컬렉션 작업을 수행한다. 따라서 Dangling Reference는 절대로 남지 않게 된다.

가비지 컬렉션 작업의 시점은 사용 가능한 메모리가 부족할 때라고 보면 되는데, 이러한 시점은 정확히 알 수 있는 것은 아니다. 또한 가비지 컬렉션이 일어나는 시점에서는 실행중인 어플리케이션의 실행 속도에 문제를 줄 수도 있는데, 실제로 이러한 상황이 언제 일어날 지 모른다는 것이 조금은 문제이다.

System 클래스에서는 이런 점을 위해 가비지 컬렉션의 시점을 지정해 줄 수 있는 메소드를 제공한다. 어디에서나 아래의 메소드를 호출하면 가비지 컬렉션을 수행할 수 있다.

```
    System.gc();
```

따라서, 계산이나 많은 메모리작업을 하기 전에 위의 코드를 삽입하여 미리 가비지 컬렉션을 해 두면 작업 중 가비지 컬렉션 작업 때문에 속도가 떨어지는 문제를 해결할 수 있다.