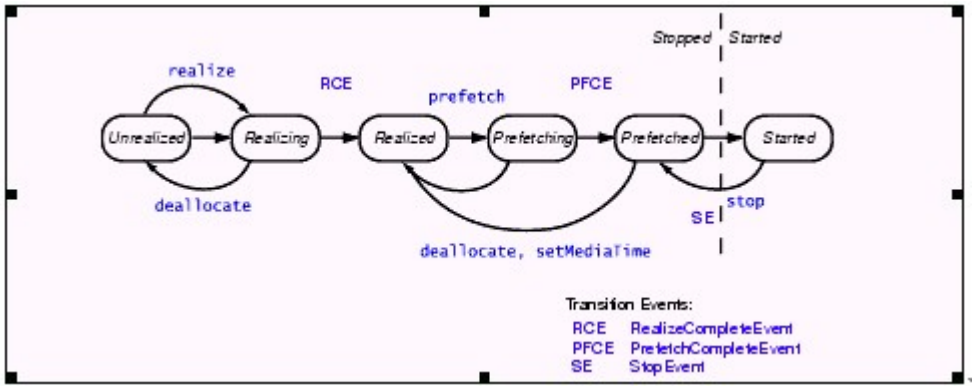
제 2 강좌 : Player, Processor를 통한 멀티미디어 JukeBox 만들기

지난 달 강좌를 통하여 JMF의 구조와 설치방법, 그리고 간략한 예제에 대하여 살펴보았다. 이번 강좌에서는 JMF 구동의 기본인 Player와 Processor의 이벤트처리를 다루어보고, Swing과 결합된 JMF JukeBox 만들기를 구현하여 JMF의 상태처리 방법과 상태정보에 따라 발생하는 이벤트획득 및 처리방법에 대하여 논의한다. 또한 JMF에서 지원되는 기본적인 GUI Component를 이용하지 않고서 사용자가 직접 GUI Component를 개발하여 추가하는 방법과 미디어 데이터 재생정보를 반영하여 GUI Component에 적용하는 방법을 구현하여 본다. 본 강좌에 대한 문의 사항이나 관련된 프로그램은 아래의 홈페이지를 참조하기 바란다.

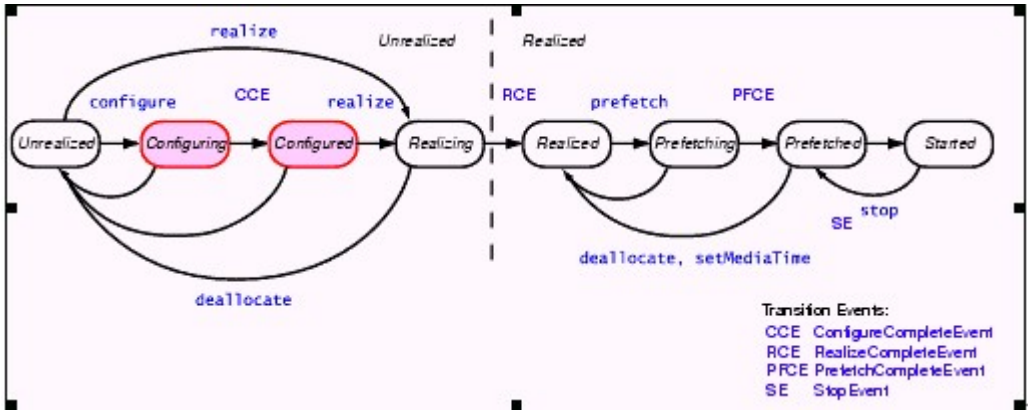
이재훈 전임연구원
삼성테크윈 정밀기기연구소(Samsung Techwin)
kingseft@samsung.co.kr
http://myhome.naver.com/kingseft

자바의 기본 구동 모델 : Player와 Processor

이번에는 Player와 Processor가 어떠한 상태정보를 나타내면서 구동되는지를 알아본다. 지난 예제는 애플릿의 기본구조를 가지고 있지만, 특별한 이벤트 처리 부분이 있다. JMF에서는 Player와 Processor에 대하여 현재 Player와 Processor가 어떤 상태에 있는지에 대한 상태정보를 이벤트화하여 이벤트 정보를 전달해준다. JMF 프로그램을 구동할 때 내부적으로 프로그램을 초기화하고, 관련 데이터소스를 얻어와 정보를 파악하고, 각종 컴포넌트 정보를 제공하고, 최종적으로 화면에 보여줄 준비를 하고, 마지막으로 화면에 보이는 작업을 수행한다. 이러한 각 단계에 대해 JMF에서는 각 상태별로 이벤트가 발생되도록 정의하였다.



[그림 1] Player의 상태변환구조



[그림 2] Processor의 상태변환구조

[그림 1] 및 [그림 2]를 살펴보면 실제로 화면에서 멀티미디어 데이터를 재생하여 보기까지에는 상당히 많은 내부적인 단계를 거쳐야 하는 것을 알 수 있다. 각 단계별로 JMF는 내부적인 처리를 어떻게 하는지 Player를 기준으로 알아보도록 하자.

Unrealized

- JMF 프로그램을 처음 구동시키면 바로 이상태로 진입한다. 즉, Player가 생성된 직후의 단계로서 이때 Player는 재생하고자하는 멀티미디어 데이터에 대한 정보를 가지고 있지않고 단순히 Player라는 객체만 생성된 단계이다.

Realizing

- Unrealized 상태에서 realize()가 호출되었을 때 이 상태로 이동한다. 그러나 주의할점은 JMF에서 모든 상태로의 이동은 바로 호출되지 않고, 일정한 지연시간을 갖는다는 점이다. 그러므로 Realizing 상태로 이동했는지 반드시 확인하고 그에따른 작업을 수행해야 한다.

Realized

- Realizing 상태로부터 RealizedCompleteEvent가 발생했을경우 이 상태로 진입한다. 이 상태에서 Player는 재생하고자하는 멀티미디어 데이터에 대한 정보를 완전히 파악하고 있는 단계이다. 이 단계에서 화면에 보이고자하는 각종 컴포넌트들의 정보를 획득할수 있다.

Prefetching

- Realized 단계에서 prefetch() 함수가 호출되면 이상태로 진입하게 된다. 이 상태에서는 현재재생중인 미디어 데이터의 시간변경 정보와 재생비율을 변경할때에도 이용된다.

Prefetched

- PrefetchingCompleteEvent가 발생되면 이 단계로 진입한다. 실제로 이 단계에서는 모든

사전준비를 마치고, 완전하게 재생을 위하여 기다리는 단계가 된다.

Started
- 실제로 멀티미디어 데이터를 재생하는 단계이다.

이젠 간략하게 Processor의 상태변화를 살펴보겠다. Processor에서는 Configuring 과 Configured 단계가 추가되었다. 이부분에 대한 설명은 아래와 같다.

Configuring
- Processor가 Unrealized 상태에서 configure() 메쏘드를 호출하면 이 상태에 진입한다. 이상태에 있는 동안에 Processor는 DataSource에 연결을 시도하고, 입력되는 데이터 스트림에 대한 Demultiplexer를찾고 , 입력데이터의 포맷정보를 찾게 된다.

Configured
- ConfigureCompleteEvent가 발생되면 진입하게 된다. 이 상태에서는 Processor가 완전히 DataSource에 연결이되었고, 내부적으로 이용할 데이터의 포맷이 결정되어진 상태이다.

JMF 프로그래밍 만들기
이제부터는 실제적인 예제를 제작하면서 관련된 사항을 분석하자. 첫번째로 만들 JMF 예제는 애플릿을 통하여 동영상 파일을 재생하는 프로그램이다. 예제 프로그램은 자바 애플릿 프로그램이므로 HTML 파일과 Java 파일이 필요하다. 먼저 예제 파일들과 컴파일 및 실행방법을 알아보록하자.

[그림 3] SimplePlayerApplet의 실행 화면

본 예제는 지난시간에 간단히 살펴본 예제와 동일한 형태이지만, 애플릿에서 JMF를 구동하는 방법과 더불어서 JMF의 Player의 내부 이벤트 처리부분에 대한 좀더 세부적인 사항을 설명하려고 한다. 먼저 HTML 소스파일부터 살펴보도록 하자.

```
<applet code=SimplePlayerApplet width=320 height=300>
<param name=file value="bluescreen2.mov">
</applet>
```

위의 소스코드에서 보듯이 일반 자바 애플릿 프로그램과 다른점이 없다. 한가지 주의할점은 param 형식으로 플레이할 파일의 이름을 적어서 value값에 주면 자바 프로그램에서 이를 읽어들이는 방법을 취하였다는 것이다. 이 부분에서 원하는 파일 이름으로 value의 값을 바꾸어주시면 된다. 물론 플레이를 하고자 하는 파일과 HTML파일 및 SimplePlayerApplet 클래스 파일은 동일한 디렉토리에 있어야 한다. 다음으로 실제 SimplePlayerApplet.java 파일을 살펴보도록 하자.

HTML의 param을 이용한 파라미터 입력

먼저 프로그램에서는 HTML의 param 형태로 읽은 미디어데이터 파일이름을 얻어와야 한다. 이 부분은 아래와 같다.

```
if ((mediaFile = getParameter("FILE")) == null)
    Fatal("Invalid media file parameter");
    try {
 url = new URL(getDocumentBase(), mediaFile);
 mediaFile = url.toExternalForm();
    } catch (MalformedURLException mue) {
}
void Fatal (String s) {
 throw new Error(s);
 }
```

소스에서 보듯 HTML의 file 파라미터의 value 값을 String으로 읽어들였다. 그 값이 없을경우 해당에러 처리를 하고 프로그램을 종료한다. 이렇게 얻은 파일이름과 HTML 도큐먼트의 정보로부터 URL을 만들어야한다. 다음으로 해야할 작업은 MediaLocator의 정보를 생성하는 부분이다. MediaLocator에는 미디어데이터에 대한 위치정보가 들어가게 된다. 아래에서 보는 것 처럼 URL 정보로부터 MediaLocator를 생성한다.

```
if ((mrl = new MediaLocator(mediaFile)) == null)
    Fatal("Can't build URL for " + mediaFile);
```

Player를 만드는 부분

위와 같은 과정을 통해서 미디어데이터에 대한 위치정보를 획득하였다. 이제는 Player를 만들어야한다. Player를 만드는 방법은 Manager 클래스를 이용하여 아래와같이 3가지의 방법중 하나를 이용하게 된다.

```
public static Player createPlayer(MediaLocator sourceLocator)
                          throws java.io.IOException,
                                 NoPlayerException

public static Player createPlayer(java.net.URL sourceURL)
                          throws java.io.IOException,
                                 NoPlayerException

public static Player createPlayer(DataSource source)
                          throws java.io.IOException,
                                 NoPlayerException
```

위의 3가지 방법을 간략히 살펴보면, 데이터의 URL 정보로 player를 생성하는 방법, DataSource로 player를 생성하는 방법, 그리고 예제 프로그램에서 이용한 MediaLocator를 통해 player를 생성하는 방법이다. 자. 그럼 실제 프로그램에서 적용한 방법을 보도록 하자.

```
try {
 player = Manager.createPlayer(mrl);
 } catch (NoPlayerException e) {
 System.out.println(e);
 Fatal("Could not create player for " + mrl);
 }
```

Manager클래스의 createPlayer를 이용하여 Player를 생성하였으며, 3가지의 Player 생성방법중에서 MediaLocator를 이용하였다.

Player의 이벤트처리 구조

이제는 생성된 Player에 대하여 각종 해당되는 이벤트를 처리하는 부분이다. 일반적인 자바 프로그램과 비슷한 방법으로 JMF에서는 특별한 방법을 이용한다. 첫째, 우리가 만든 자바애플릿은 ControllerListener를 구현상속해야 한다. 즉, Player의 모든 이벤트처리는 바로 ControllerListener를 상속받아 해당 이벤트 처리를 해주어야 한다. 둘째로 생성된 Player에게 이벤트 처리를 등록시켜야 한다. 마지막으로 이렇게 ControllerListener 이벤트를 처리할 때 반드시 구현해야 하는 메쏘드가 있다.

```
public synchronized void controllerUpdate(ControllerEvent event) {  }
```

위와같이 메쏘드의 이름은 controllerUpdate이고, 이 메쏘드의 인자인 ControllerEvent의 종류에 따라서 Player의 상태를 알수 있고, 상태에 따른 해당작업을 수행할 수 있다. 그럼 이와 같은 controllerUpdate 메쏘드내부에서의 이벤트 처리과정을 살펴보도록 하자.

```
if (event instanceof RealizeCompleteEvent) {
} else if (event instanceof CachingControlEvent) {
} else if (event instanceof EndOfMediaEvent) {
} else if (event instanceof ControllerErrorEvent) {
}
```

controllerUpdate 메쏘드의 파라미터인 ControllerEvent는 위와같이 instanceof를 이용하여 이벤트의 종류를 판별할수 있다. 이외에도 많은 이벤트들이 있으니 여러분이 직접 API 문서를 참조하시기 바란다.

애플릿의 기본동작과 Player

애플릿 프로그램의 기본적인 메쏘드인 init(), start(), stop(), destroy()등의 메쏘드 내부에서 Player를 제어할 수 있다. 애플릿이 시작하는 start() 메쏘드 수행시에 Player를 시작시키고, 애플릿이 중단되는 stop()메쏘드에서 Player를 중단시키며, 애플릿이 종료되는 destroy() 메쏘드내에서 Player를 종료시킬수 있다.

```
public void start() {
     if (player != null)
     player.start();
}
public void stop() {
 if (player != null) {
     player.stop();
        player.deallocate();
    }
}
public void destroy() {
 player.close();
}
```

JMF의 Component 얻어오기

JMF의 컴포넌트들은 Player가 완전히 Realized 상태, 즉 RealizeCompleteEvent 가 발생한후에 이용할 수 있다. 이러한 컴포넌트는 Visual Component와 ControlPanel Component가 있으며, 이러한 컴포넌트들을 획득한후에 적절한 Layout Manager를 이용하여 원하는 위치에 표현할 수 있다. Component를 얻는 메쏘드는 2가지 종류가 있으며, 실제 이부분에 대한 구현 소스는 아래와 같다.

```
if (event instanceof RealizeCompleteEvent) {
 int width = 320;
 int height = 0;
 if (controlComponent == null)
 if (( controlComponent = player.getControlPanelComponent()) != null) {
```

```
    controlPanelHeight = controlComponent.getPreferredSize().height;
    panel.add(controlComponent);
    height += controlPanelHeight;
 }
 if (visualComponent == null)
  if (( visualComponent = player.getVisualComponent())!= null) {
      panel.add(visualComponent);
      Dimension videoSize = visualComponent.getPreferredSize();
      videoWidth = videoSize.width;
      videoHeight = videoSize.height;
      width = videoWidth;
      height += videoHeight;
      visualComponent.setBounds(0, 0, videoWidth, videoHeight);
  }
     panel.setBounds(0, 0, width, height);
     if (controlComponent != null) {
   controlComponent.setBounds(0, videoHeight, width, controlPanelHeight);
   controlComponent.invalidate();
     }
 }
}
```

대용량 파일의 다운로드 과정표기

실제로 웹에서 대용량의 멀티미디어 파일을 플레이하기 위해서 다운로드를 받는경우, 현재 어느정도까지 다운로드를 받았는지, 앞으로 어느정도의 시간이 더 필요한지에 대한 정보를 시각적으로 제공해준다면 유저가 프로그램을 이용하기에 한결 편할것이다. JMF에서도 이러한 방법을 제공해준다. ControllerEvent의 종류중에서 CachingControlEvent 를 통해서 접근을 할 수 있으며, 획득한 이벤트의 getCachingControl() 메쏘드를 통하여 다운로드 컨트롤을 얻을 수 있다. 실제 구현된 소스는 아래와 같다.

```
if (event instanceof CachingControlEvent) {
    if (player.getState() > Controller.Realizing)
   return;
    CachingControlEvent e = (CachingControlEvent) event;
    CachingControl cc = e.getCachingControl();
```

```
    if (progressBar == null) {
     if ((progressBar = cc.getControlComponent()) != null) {
      panel.add(progressBar);
      panel.setSize(progressBar.getPreferredSize());
      validate();
    }
   }
}
```

미디어의 반복 재생부분

미디어 파일의 재생을 모두 마치면, 기본적으로 JMF는 반복재생을 하지 않는다. 그렇기 때문에 미디어 데이터의 재생종료를 알리는 이벤트 처리를 해주어야 한다. 이렇듯 재생 종료시 발생하는 이벤트는 EndOfMediaEvent 이다. 이 이벤트가 발생했을 때 Player의 MediaTime을 0 값, 즉 최초의 시작시간으로 옮겨주고 다시 Player를 시작시키면 미디어데이타의 재생이 종료되는 마지막시점에서 다시 처음부터 재생을 시작하게 된다. 아래는 구현된 소스이다.

```
if (event instanceof EndOfMediaEvent) {
     player.setMediaTime(new Time(0));
     player.start();
}
```

Controller의 상태처리에 관한 이벤트

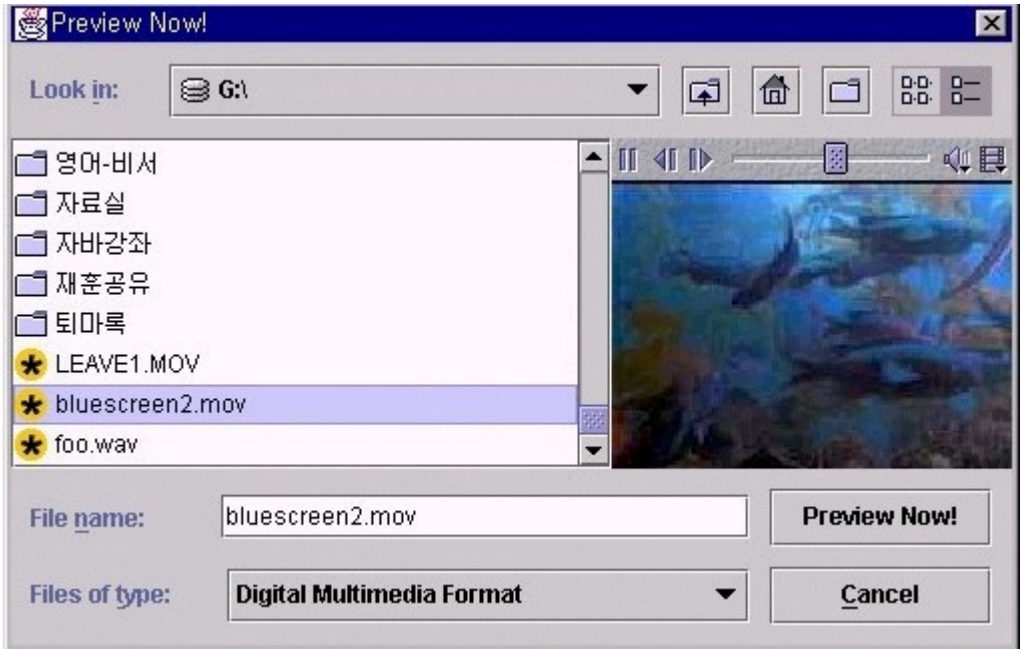
Controller의 특정 상태 이벤트처리도 가능하다. 이때에는 ControllerCloseEvent 및 ControllerErrorEvent 이벤트를 처리해주면 된다. 관련된 소스는 아래와 같다.

```
if (event instanceof ControllerErrorEvent) {
 player = null;
 Fatal(((ControllerErrorEvent)event).getMessage());
} else if (event instanceof ControllerClosedEvent) {
     panel.removeAll();
  }
}
```

이상과 같이 전체적인 프로그램 소스에 대하여 기능별로 분석을 시도하였다. 완전한 소스코드는 지면관계상 생략을 하고, 부록 CD를 참조하기 바란다.

두번째 예제 : Swing을 이용한 JukeBox 만들기

이전 예제에서는 Caching Control, State 처리, Remote 멀티미디어 파일 처리등에 대한 처리가 부족했다. 두번째 예제는 스윙과 프리뷰기능을 이용한 멀티미디어 다이얼로그를 만들어 보자. 스윙의 JFileChooser 를 이용한 파일선택 다이얼로그 박스를 만드는 것과 거의 비슷하지만, JMF의 state 처리와 적절한 Caching control 기능을 추가하여 프로그램을 구현한다. 이 프로그램에서는 mpg, mp3, mov 등의 멀티미디어 데이터를 재생하고 파일 선택이 가능하도록 구성되어있다. 프로그램의 초기 구조로서 버튼을 생성하고, listener를 추가시켰으며, 하단부에 output area를 생성해서 사용자가 파일을 선택한 경우 사용자에게 선택한것에 대한 응답을 표시한다. 초기 애플리케이션에서 Preview 버튼을 누르면 actionPerformed() 이벤트 처리 함수가 실행되고, 이 부분에서 우리가 만드는 변형된 JFileChooser 다이얼로그 프로그램이 수행되게 된다. 프로그램의 개발단계는 일단, 일반적인 JFileChooser 다이얼로그 박스를 생성하고, 이렇게 생성된 다이얼로그 박스에 파일필터기능, 파일 뷰어 기능, 파일 프리뷰 기능을 추가하여 완성된 프로그램으로 개발하게 된다. 먼저 [그림 4]를 확인하여 우리가 만들고자 하는 프로그램의 외형부터 살펴보도록 하자



[그림 4] Swing과 결합된 JMFFileChooser 프로그램

JMFFileChooser 프로그램의 구조

먼저 JMFFileChooser를 살펴보면, 스윙을 이용하기 위해서 import 문을 이용해 스윙 패키지를 임포트하고, JFrame을 상속받았다.

```
import javax.swing.*;
import javax.swing.filechooser.*;
public class JMFFileChooser extends JFrame{
```

그럼 다음 단계로 JMFFileChooser의 생성자 부분을 한번 보기로 하자.

```
public JMFFileChooser()
{
        super("JMF Media Previews by kingseft");
        JButton sendButton = new JButton("Preview");
        sendButton.addActionListener(new SendListener());
        log = new JTextArea(5,20);
        log.setMargin(new Insets(5,5,5,5));
        JScrollPane logScrollPane = new JScrollPane(log);
        Container contentPane = getContentPane();
        contentPane.add(sendButton, BorderLayout.NORTH);
        contentPane.add(logScrollPane, BorderLayout.CENTER);
}
```

이부분에서는 프리뷰 기능을 수행하기 위해 선택 버튼을 하나 만들고, 사용자 선택에 대한 결과를 표시하기 위한 영역을 지정한다. 생성된 버튼과 스크롤페인을 부착시킴으로서 사용자 선택과 파일선택 결과표시를 위한 GUI 생성을 수행하였다. 다음 단계는 생성된 파일선택 다이얼로그에 필터를 연결하는 부분이다. 예를들어서 *.mpg만 선택하거나... *.mp3만 선택하는경우 .. 이러한 방식으로 나열된 파일들에 대해 필터 처리를 하는 부분을 구현한다.

```
JFileChooser filechooser = new JFileChooser();
filechooser.addChoosableFileFilter(new MultimediaFilter());
filechooser.addChoosableFileFilter(new MultimediaFilter());
```

먼저 파일선택 다이얼로그 박스를 만들고 멀티미디어 필터를 추가하여서 원하는 멀티미디어 파일만 표시되게 한다. 또하나 필요한 작업 과정은 파일 선택 박스에 넣을 사용자 정의 선택 리스트를 만드는 부분이다. 이때 이용하는 메쏘드가 바로 addChooserFileFilter() 이다. 다음 단계는 필터링을 거쳐서 출력되는 파일정보들에 대해 User Interface 를 추가하는 부분이다. 이때는 JFileChooser 의 setFileView() 메쏘드를 사용한다. 위의 부분은 쉽게 이야기하자면, 필터링된 파일들중에서 mp3 파일들의 아이콘을 동일하게 사용자

정의 아이콘으로 바꾸고, mov 파일들의 표시 아이콘들도 다른 동일한 아이콘들로 바꾸는등 보기 편하게 만드는 기능을 구현하는 부분이다. 여기서 구현할 사항은 멀티미디어 프리뷰 클래스를 만들고, 이 클래스 오브젝트를 파일선택 다이얼로그 박스에 추가하는 작업이다. 아래의 코드는 실제적으로 멀티미디어 프리뷰 기능을 하기위한 클래스를 이용하는 부분이다.

```
MultimediaPreview mmpreview = new MultimediaPreview(filechooser);
mmpreview.setLayout( new BorderLayout());
filechooser.setAccessory(mmpreview);
```

자.. 파일선택 다이얼로그 박스를 만드는 부분은 모두 마쳤고, 그러면 실제적으로 유저화면에 보여주는 부분의 구현이 남았다. 먼저 기본 다이얼로그 박스를 보여주는 부분을 구현하고 사용자가 선택한 파일의 이름을 출력하는 부분을 구현한다. 마지막으로는 사용했던 멀티미디어 리소스를 풀기위해서 cleanup 메쏘드를 이용합니다. 구현 코드는 아래와 같다.

```
int returnVal = filechooser.showDialog(JMFFileChooser.this, "Preview Now!");
if (returnVal == JFileChooser.APPROVE_OPTION)
{
       File file = filechooser.getSelectedFile();
       log.append("You chose: " + file.getName() + "." + newline);
}
else
{
       log.append("You quit without selecting a file." + newline);
}
mmpreview.cleanup();
```

순진한 우리 개발자들에게 주의!!!
이미 알고 있겠지만, Player 는 결코 동기적으로 수행되지 않는다. 먼저 아래의 코드를 살펴보자. 이와 같은 프로그램을 작성한다면 정말 예상하지 못할 결과가 초래된다. JMF 는 현재의 상태정보에 따라서 이벤트를 발생시키고, 해당이벤트가 발생되기 전까지는 우리가 원하는 상태에 진입했다고 단정지을 수 없다.

```
if( player != null){
       player.stop();
       player.deallocate();
```

```
        player.close();
        player = null;
        removeAll();
}
```

위의 코드는 개념적으로는 당연한 코드라고 생각할수있다. 그러나, 강좌 첫 부분에서도 언급했듯이 player 는 player state 라는 5 단계를 거치게 된다. 그래서 위와같은 프로그램을 수행할때 정말로 player 가 현재 어떤 상태에서 수행중인지를 먼저 검사해야 한다는 것이다. player 의 현재 상태에 따라서 우리가 해야하는 작업도 각기 달라지게된다. player 의 deallocate() 함수는 만약 player 가 Started 상태에서 호출된다면 Exception 을 발생시키게 되며 또한 Stop 은 비동기적으로 호출되기 때문에, 실제로 player 가 StopEvent 를 받기전까지는 player 가 Started 상태를 벗어났다고 단정할 수 없다. 위와같은 코드는 player 가 알맞은 상태에 진입하고 탈출할때까지 기다리지 못하기 때문에 player 가 stop 되기도 전에 deallocate 되는 위험이 포함되어 있다. 자. 그럼 해결 시나리오를 한번 구상해 보자. 지금 수행되고 있는 파일선택 프리뷰 다이얼로그 박스를 갑자기 종료했다고 가정하자. 그러면 다이얼로그 박스만 종료시키면 모든 것이 다 끝이라고 생각하면 안된다. 다이얼로그 박스를 종료할때 player 역시 종료 시켜주어야 하며, 그렇지 않다면, 결국 메모리의 리소스가 유출되어 memory leak 가 발생하게 되고, 심지어 다이얼로그 박스가 종료된 후에도 우리가 선택한 멀티미디어 파일이 계속해서 프리뷰된 상태로 지속된다는 것이다. 실제적인 디바이스를 셧다운 시키기 위해서 cleanup() 메쏘드를 이용하였다. cleanup()함수에서는 ClosePlayer() 함수를 내부에서 호출하고 바로 ClosePlayer() 함수가 메모리 유출을 막는 구현부분이다.

```
private void ClosePlayer( )
{
        if ( player != null )
        {
                bclosing = true;
                bprcoessedevent = false;
                beforeTime = System.currentTimeMillis();
                player.stop();
                synchronized (this )
                {
                        try
                        {
                                if ( bprcoessedevent != true )
```

```
                {
                        wait();
                }
        }
        catch( InterruptedException e)
        {
        }
}

afterTime = System.currentTimeMillis();
System.out.println( "Stop took " + (afterTime - beforeTime) + " ms");
beforeTime = System.currentTimeMillis();
bprcoessedevent = false;
player.deallocate();
synchronized (this )
{
        try
        {
                if ( bprcoessedevent != true )
                {
                        wait();
                }
        }
        catch( InterruptedException e)
        {
        }
}
afterTime = System.currentTimeMillis();
System.out.println( "Deallocate took " + (afterTime - beforeTime) + "
ms");
beforeTime = System.currentTimeMillis();
bprcoessedevent = false;
player.close();
synchronized (this )
{
        try
```

```
                        {
                                if ( bprcoessedevent != true )
                                {
                                        wait();
                                }
                        }
                        catch( InterruptedException e)
                        {
                        }
                }
                afterTime = System.currentTimeMillis();
                System.out.println( "Close took " + (afterTime - beforeTime) + " ms");
                bclosing = false;
                 player = null;
                removeAll();
        }
}
```

위의 프로그램에서 먼저 Player를 멈추도록 stop()이벤트를 호출하고 Player가 stop() 단계로 가서 stopEvent가 발생되도록 유도한다. stop()메쏘드만 불렀다고 해서 Player가 stop이 바로 되지 않기 때문에 stopEvent를 기다려야 한다. stopEvent를 받은후에 bprocessedevent 변수를 true가 되게 했기 때문에 다음부분이 실행되고, 이제 우리가 사용했던 리소스를 풀어주어야 한다. 이부분 역시 deallocate() 함수를 호출하고 기다려야 한다.
이후에는 deallocateEvent 를 만날때까지 기다려야 하고, Player를 종료하도록 한다. 그후에 closeEvent 이벤트를 만날때까지 또 기다려야 하고 최종적으로 쓸모없는 메모리 영역을 garbage collected 되도록 설정합니다. 프로그램 중간중간에 실제 경과 시간을 표시하도록 했으니 관심있는 독자들은 JMF의 각 상태에 따른 이벤트 이동시간이 얼마나 경과하는지를 검토해보는것도 좋을 듯 하다.

Player의 이벤트 처리부분

이번에는 Player가 재생되기 위해서 발생하는 각종 이벤트를 처리하는 부분을 살펴봐야한다. 먼저 살펴볼것은 파일선택 다이얼로그 박스에서 특정한 하나의 파일을 선택했을경우 그 파일에 대해서, 올바른 멀티미디어 파일인지를 판별하고, 알맞은 파일이라면, 기존의 player를 제거하고, 새로운 player를 생성하고, 이벤트 핸들러를 등록하는 과정이다. 일단 파일 다이얼로그 박스에서 파일 선택에 대한 이벤트를 처리하기위해서 PropertyChangeListener를 구현해야 하며, 당연히 ControllerListener도 구현해야 한다. 실제 구현 클래스의 구조

는 아래와 같다.

```
public class MuitimediaPreview extends JComponent implements
PropertyChangeListener, ControllerListener{
```

이부분에서 한가지 의문이 발생한다. 왜 JComponent를 상속받을까? 파일 선택 다이얼로그를 만들고 그 다음 우리가 프리뷰 기능을 파일선택 다이얼로그 박스의 악세서리 기능으로 추가가 된다고 하였다. 그래서 우리가 생성하는 클래스가 JComponent를 상속받고 나중에 파일선택 다이얼로그 박스의 setAccessory(mmpreview)와 같이 등록이 되는 것이다.

파일선택에 따른 이벤트의 처리
이제 남은 부분은 파일 선택이 되었을때의 이벤트 처리 부분이다.

```
public void propertyChange(PropertyChangeEvent e)
{
	String prop = e.getPropertyName();
	if (prop == JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
	{
		ClosePlayer();
		f = (File) e.getNewValue();
		try
		{
			String mediaFile = "file:///" + f.getPath();
			URL mediaURL = new URL( mediaFile );
			beforeTime = System.currentTimeMillis();
			player = Manager.createPlayer(mediaURL);
			afterTime = System.currentTimeMillis();
			 System.out.println( "Create player took " + (afterTime -
beforeTime) + " ms");
			player.addControllerListener(this);
			beforeTime = System.currentTimeMillis();
			player.prefetch();
		}
		catch(Exception except)
		{
			System.err.println("Exception: " + except );
```

```
            }
        }
}
```

먼저 파일 선택 다이얼로그 박스에서 파일선택 변환시 발생한 이벤트에서 이벤트의 프러퍼티 이름을 얻어오고, 이벤트의 프러퍼티 이름이 SELECTED_FILE_CHANGED_PROPERTY 이면 파일의 프리뷰작업을 시작한다. 그렇다면 이전까지, 혹은 지금현재 진행중인 player를 종료하는 과정을 수행한다. 파일 종료후에는 새로 선택된 파일 이름을 받아오고 선택된 파일을 통하여 URL 정보를 얻는다. 이후에 선택된 파일에 대한 프리뷰 기능을 위해서 Player를 생성한후에 플레이어가 이벤트에 반응하도록 이벤트 등록을 한다. 마지막으로 prefetch 단계에 이르도록 하여 실제로 이용가능한 상태에 진입하도록 한다.

Player의 상태변화에 따른 이벤트처리방법

이제는 player의 이벤트 처리에 대해 알아보자. 아래 코드를 보면 상당히 많은 이벤트 종류가 있다는 것을 알 수 있다. 이러한 이벤트들은 instanceof 를 통하여 그 이벤트 종류를 구분한다.

```
public synchronized void controllerUpdate(ControllerEvent event)
{
        if (event instanceof CachingControlEvent)
        {
                CachingControlEvent e = (CachingControlEvent) event;
                CachingControl cc = e.getCachingControl();

                if (jmfProgress != null)
                {
                      if ( cc.getContentProgress() == cc.getContentLength() )
                        {
                                if (player.getState() > Controller.Realizing)
                                {
                                        remove( jmfProgress );
                                        jmfProgress = null;
                                        validate();
                                }
                  }
                      else
```

```
                        {
                                long progress = ( cc.getContentProgress() /
(cc.getContentLength()/100));
                                jmfProgress.setValue((int) progress);
                        }
                        repaint();
                }
                else
                {
                        jmfProgress = new JProgressBar();
                        jmfProgress.setMinimum(0);
                        jmfProgress.setMaximum(100);
                        jmfProgress.setValue(0);
                        add(jmfProgress, BorderLayout.SOUTH);
                        validate();
                        repaint();
                }
        }
        else if (event instanceof RealizeCompleteEvent)
    {
                player.prefetch();
        }
        else if (event instanceof PrefetchCompleteEvent)
        {
                afterTime = System.currentTimeMillis();
                System.out.println( "Prefetch player took " + (afterTime - beforeTime)
+ " ms");
                Component alohaGUI = player.getVisualComponent();
                Component alohaControl = player.getControlPanelComponent();
              if ( alohaControl != null )
                {
                        add(alohaControl, BorderLayout.NORTH);
                }
                if ( alohaGUI != null )
                {
                        add(alohaGUI, BorderLayout.CENTER);
```

```
        }
        player.start();
        validate();
        repaint();
        setVisible(true);
}
else if (event instanceof EndOfMediaEvent)
{
        player.setMediaTime(new Time(0));
}
else if (event instanceof ControllerErrorEvent)
{
        player = null;
}
else if (event instanceof ControllerClosedEvent)
{
        synchronized (this )
        {
                if ( bclosing == true )
                {
                        bprcoessedevent = true;
                        notify();
                }
        }
}
else if (event instanceof DeallocateEvent)
{
        synchronized (this )
        {
                  if ( bclosing == true )
                {
                        bprcoessedevent = true;
                        notify();
                }
        }
}
```

```
        else if (event instanceof StopEvent)
        {
                synchronized (this )
                {
                        if ( bclosing == true )
                        {
                                bprcoessedevent = true;
                                notify();
                        }
                }
        }
}
```

먼저 살펴볼 이벤트는 CachingControlEvent 이다. 이 이벤트는 멀티미디어 파일을 네크워크 등에서 다운로드 받을때 다운로드 받는 시간이 오래걸린다면 ProgressBar를 표시하고 다운로드경과를 표시해 주는 역할을 한다. 물론 다운로드가 끝나면 당연히 ProgressBar 를 제거해야한다. 이때 발생하는 이벤트가 CachingControlEvent 이고, 받아오는 컨트롤이 CachingControl 입니다. ProgressBar가 존재하지 않는다면 추가하고 현재 상태가 이미 Realized 되었다면 다운로드 ProgressBar 제거하며, 현재 다운로드 중이라면 현재 몇 % 진행되었는지를 계산하고 그 값을 progressBar에 지정해준다. progressBar의 값이 갱신됨에 따라 progressBar의 위치도 새롭게 그려야 한다. RealizeCompleteEvent 이벤트 단계에서는 수행할 작업이 없으니 player.prefetch() 를 호출하여 다음 이벤트 상태로 옮기도록 명령한다. PrefetchCompleteEvent 이벤트는 Prefetch 단계까지 완전히 마친후에 즉, start 직전단계에 다다랐을때 를 의미하므로 이 부분에서 player에 연결된 VisualComponent와 ControlPanelComponent를 획득하여 부착시키는 작업을 수행한다. 또한 이 부분에서 PREVIEW 하게 되면 바로 playback 되도록 기능을 추가시키었다. EndOfMediaEvent 이벤트 발생시에는 실행한 미디어의 마지막 위치에 왔을 때 이므로 미디어의 현재 위치를 맨 처음으로 되돌린다. ControllerErrorEvent 이벤트가 발생시에는 예기치 못한 에러의 발생임을 가정하고 null 처리를 해준다. ControllerClosedEvent 이벤트 발생시에는 올바른 종료를 위하여 일단은 대기 상태에 들어가도록 변수값만 세팅한다. DeallocateEvent 이벤트 발생시에는 사용자원에 대한 리소스를 제거할때 발생되는 이벤트 처리 를 수행해준다. 마지막으로 StopEvent 처리에는 player의 작업 중단을 알리는 기능을 첨부한다.

커스텀 사용자 정의 GUI 만들기

이번부분은 Player를 이용할 때 JMF에서 기본적으로 제공하는 GUI를 대체하여 새롭게 사용자가 정의한 GUI Component를 추가하고, 미디어 데이터 재생시 재생되는 정보를 GUI

Component들과 어떻게 서로 연동시켜야 하는지에 대하여 알아본다. 먼저 실행되는 결과 화면부터 살펴보자 [그림 5]를 살펴봄으로서 JMF의 표준 GUI와 비교하여 보기바란다.

[그림 5] Custom GUI를 가지는 미디어 재생기

[그림 5]에서 살펴본바와 같이 Custom UI를 만들기 위해서 먼저 화면에 보이는 부분이 필요하다. 기본적으로 제공되는 UI를 이용하지 않는다면. play 버튼, stop 버튼, 일시정지 버튼 등에 해당하는 모든 이미지를 전부 프로그래머가 그려 주어야 한다. 두번째로 생각해야 할 점은 JMF로 파일을 재생하는 경우 파일 재생 시간을 알려주는 Slider 컨트롤과 Volume을 나타내는 Slider 컨트롤 부분의 구현으로서 재생하는 도중에 계속해서 이 두가지 컨트롤을 업데이트 해주어야 합니다. 반대로 생각하면 사용자가 위의 두가지 컨트롤을 마우스로 움직인다면, 그 결과가 현재 재생되고 있는 파일에도 영향을 미쳐서 볼륨 조정이라든지.. 특정 원하는 위치에서부터 재생한다든지 하는 작업이 필요하다. 접근하기가 쉽지만은 않지만, 언급한 작업들을 일단하자. 연속적으로 미디어 파일의 정보를 업데이트 해야 하니까 일단 쓰레드를 하나 만들고 이 쓰레드를 통해서 볼륨과 미디어타임의 표시를 계속해서 갱신해야한다. 또한 프로그램에서는 사용자의 입력 정보를 판단하는 부분이 있어야 하며, 현재 마우스위치가 알맞은 User Interface상에 있는지도 판단되어야 한다.

프로그램의 구조 설계하기

[그림 5]는 Frame 기반의 User Interface 대신에 마치 WinAmp와 같은 모습의 컨트롤을 만든다. 재생되는 화면은 Border와 Menu가 없는 구조로 바로 Window를 상속받아서 경계선이나 메뉴가 없는 윈도우가 생성한다. 하단부의 UI는 하나의 JPEG 그림이다. 실제로 여러분이 원하는 부분 부분 전부 입맛에 맞게 새로 그려서 추가를 할 필요도 있을것이다. 중요한 것은 play 버튼의 위치가 어디이고, stop 버튼의 위치가 어디이고, close 버튼의 위치가 어딘가 즉, 각 UI에 대한 절대적인 좌표의 위치를 알아야 한다는 것이다. 이 부분은 프로그램의 소스부분에서 좀더 설명을 하도록하겠다. 먼저 프레임을 만들고 우리가 새롭게 만들 클래스를 호출하는 부분이다. 이 프로그램은 내부에서 다시 MainWindow를 생성하도록 되어있고 모든 실제적인 기능은 결국 MainWindow.java에서 구현되어 있습니다

```
public class Main extends Frame {
    public Main(String [] args) {
        new MainWindow(this, args);
    }
     public static void main(String[] args) {
        Main main = new Main(args);
```

```
        main.invokedStandalone = true;
    }
    private boolean invokedStandalone = false;
}
```

위에서 보듯이 이 프로그램 즉, Main.java는 결국 내부에서 다시 MainWindow를 생성하도록 되어있습니다. 모든 프로그램의 기능은 결국 MainWindow.java에서 구현되어 있습니다

자. 그러면 이제 실제 구현되야할 클래스를 살펴보도록 하자. 먼저 클래스 선언부 이다.

```
public class MainWindow extends Window implements MouseMotionListener,
    MouseListener, ControllerListener, Runnable {
```

Window 에서 확장되었기 때문에 일반적은 Frame이 아니다. Window는 메뉴나 경계선이 없는 구조를 형성한다. 그리고 Mouse 버튼을 누르는경우를 감지해야 하니까 MouseListenr를 구현해야 하고, Volume 컨트롤 또는 미디어 타임을 나타내는 컨트롤을 사용자가 마우스로 누르고 Drag 하는 경우를 생각해야 하기 때문에 MouseMotionListener 를 구현해야한다. 이 클래스의 생성자부터 보도록 하자.

```
public MainWindow(Frame frame, String [] args) {
        super(frame);
        this.frame = frame;
        parseArgs(args);
        createGUI();
        setBounds(200, 400, WIDTH, HEIGHT);
        setVisible(true);
        addNotify();
        if (movieURL != null)
            loadMovie(movieURL);

  timeThread = new Thread(this);
  timeThread.start();
    }
}
```

생성자에서 입력되는 Command Line String을 분석한다. 이 부분이 parseArgs()이다.

createGUI() 부분을 통해 실제 화면에 나타날 각종 그래픽부분의 세팅을 하며 화면에 놓일 위치를 결정한다. 다음은 입력파일이 정상적이라면 파일을 읽어들이는 loadMovie 작업을 수행하는 부분이다. 마지막으로 Time Slider와 Volume Slider를 지속적으로 갱신하기 위해 쓰레드를 생성해주어야 한다. 입력되는 Command Line을 판단하는 부분이다

```
private void parseArgs(String [] args) {
        if (args.length > 0)
            movieURL = args[0];
    }
}
```

다음은 입력되는 URL이 올바른 경우 player를 생성하는 부분이다. 내부적으로 주어진 입력에 protocol이 명시되지 않았다면 "file: "를 앞에 추가합니다. 입력되는 URL을 가지고 Player를 생성하고, ControllerListener를 첨가시키고 realize를 과정을 구현한다.

```
private void loadMovie(String movieURL) {
  if (movieURL.indexOf(":") < 3)
      movieURL = "file:" + movieURL;
  try {
      player = Manager.createPlayer(new MediaLocator(movieURL));
      player.addControllerListener(this);
      player.realize();
        } catch (Exception e) {
            System.out.println("player 생성오류가 발생했습니다.");
            return;
        }
    }
}
```

이제 구체적인 player의 동작은 controllerUpdate에서 실행된다. 아래에 이 부분에 대한 소스 이다.. 이부분은 두번째 예제 부분에서 많은 설명이 있었으므로, 독자 스스로가 분석해 보기 바란다.

```
  public void controllerUpdate(ControllerEvent ce) {
  if (ce instanceof RealizeCompleteEvent) {
      if (videoPanel == null)
```

```
            videoPanel = new Window(frame);
        else
            videoPanel.removeAll();

          Component vis = player.getVisualComponent();
              if (vis != null) {
                  videoPanel.add(vis);
      videoPanel.setSize(vis.getPreferredSize());
      centerVideoPanel();
      videoPanel.setVisible(true);
        }
       player.start();
          } else if (ce instanceof ControllerClosedEvent) {
              if (closing)
                  System.exit(0);
              else {
      timex = -1;
      gain = null;
      if (videoPanel != null) {
          videoPanel.dispose();
          videoPanel = null;
      }
              }
        player = null;
          } else if (ce instanceof EndOfMediaEvent) {
        rewind();
        play();
    } else if (ce instanceof PrefetchCompleteEvent) {
         gain = (GainControl) player.getControl("javax.media.GainControl");
        repaint();
    }
      }
}
```

다음은 실제적인 GUI를 그리고 위치시키는 작업과정이다. 먼저 고정적인 위치 결정을 위해 레이아웃을 null로 지정하였으며 , 마우스 리스너와 마우스모션리스터 등록시킨다. 추

가적으로 배경이 되는 비디오플레이어 판넬 그림 읽어들이는 작업을 수행한다. 그리고 실제 컨트롤의 그림을 읽어들인다.

```
private void createGUI() {
      setLayout(null);
        addMouseListener(this);
        addMouseMotionListener(this);
    MediaTracker mt = new MediaTracker(this);
  image = loadImage("/jamp.jpg", this, true);
  mt.addImage(image, 1);
  try {
      mt.waitForID(1);
  } catch (InterruptedException ie) {
  }
    }
}
```

이제 남은 부분은 쓰레드로 구동되는 업데이트 부분과 마우스 처리 부분이다. 먼저 마우스의 버튼눌림에 대한 처리 부분을 살펴보자. 마우스버튼을 누른경우 현재 마우스의 좌표를 얻어와서 그 좌표가 미리지정한 위치내에 들어왔는가를 판단하고 해당작업 수행해야한다. 사용하려는 모든 버튼의 개수만큼 검사하고, 현재의 startPint.x와 startPoint.y가 주어진 좌표내에 들어왔는가를 판단하는 작업을 추가한다. 이 부분에서 중요한 점은 마우스 버튼이 눌려졌을경우 현재 마우스 좌표값이 우리가 원하는 특정 GUI의 위치내부에 들어왔는지를 x, y 좌표값으로 판단해야 한다는 것입니다. 마우스 버튼이 눌려진후 그 순간의 마우스 좌표에 따라서 play, stop, pause , close 등등의 기능을 수행하면 된다.

```
public void mousePressed(MouseEvent parm1) {
        toFront();
        startPoint = parm1.getPoint();
      notOutside = false;
  inVolume = false;
  inTime = false;
  for (int i = 0; i < COMPONENTS.length; i++) {
     if (startPoint.x >= COMPONENTS[i][0] &&
          startPoint.y >= COMPONENTS[i][1] &&
          startPoint.x <= COMPONENTS[i][2] &&
```

```
            startPoint.y <= COMPONENTS[i][3] ) {
        switch (i) {
        case PLAY:        play();        break;
        case PAUSE:       pause();       break;
        case REWIND:      rewind();      break;
        case EJECT:       eject();       break;
        case VOLUME:
            inVolume = true;
            volumeDragged(startPoint.x);
            break;
        case MEDIATIME:
            inTime = true;
            timeDragged(startPoint.x);
            break;
        case CLOSE:
            closing = true;
            close();
            if (player == null)
                System.exit(0);
            break;
        case MAXIMIZE:
            maximize();
        }
        notOutside = true;
        break;
          }
     }
        }
}
```

각 선택조건에 맞게 실제적인 해당작업을 하는 부분은 아래와 같다. 특히 주의할 점은 Eject() 부분이다. Eject 버튼을 누르면 기존의 player를 Close하고 새로운 파일을 연다. 이 부분에서는 파일을 선택하는 다이얼로그 박스가 생성되고 원하는 파일을 선택하게 되어 있다. 특히, player의 close 메쏘드를 호출한다음 ControllerClosedEvent가 발생될때까지 기다려야 한다.

```
  private void close() {
        if (player != null) {
            player.stop();
            player.close();
        }
    }

private void play() {
  if (player != null)
      player.start();
    }

private void pause() {
  if (player != null)
      player.stop();
    }

private void rewind() {
  if (player != null)
      player.setMediaTime(new Time(0));
    }

private void eject() {
  FileDialog fd;
    close();
   while (player != null) {
      try {
    Thread.currentThread().sleep(100);
      } catch (InterruptedException ie) {
      }
  }
  fd = new FileDialog(frame, "Open File", FileDialog.LOAD);
  if (lastDir != null)
      fd.setDirectory(lastDir);
  fd.show();
  lastDir = fd.getDirectory();
```

```
  String filename = fd.getFile();
  if (filename == null)
      return;
  else {
      loadMovie(lastDir + filename);
  }
  fd.dispose();
    }
}
```

사용자가 볼륨이나 미디어타임의 위치를 마우스로 드래깅하는 경우 이값이 다시 플레이어에게 전달되고 Custiom GUI에도 그 결과가 반영되어야 한다. 이 부분은 아래와 같이 구현될 수 있습니다.

```
public void mouseDragged(MouseEvent parm1) {
  if (notOutside) {
      if (inVolume)
    volumeDragged(parm1.getPoint().x);
      else if (inTime)
    timeDragged(parm1.getPoint().x);
      return;
  }
  startLocation = getLocation();
        setLocation(startLocation.x - startPoint.x + parm1.getPoint().x,
                    startLocation.y - startPoint.y + parm1.getPoint().y);

  if (videoPanel != null) {
      centerVideoPanel();
  }
        Toolkit.getDefaultToolkit().sync();
        repaint();
    }
}
```

볼륨과 미디어타임에 대한 각각의 기능은 아래와 같습니다. volume control를 드래그하는 경우 컨트롤 판넬속의 마우스의 움직임을 통하여 볼륨레벨을 결정하며, player로부터

GainControl을 얻어옴으로서 볼륨레벨을 설정한다. 마지막으로 변경된 값을 GUI에 갱신시켜야 한다. 타임 컨트롤을 드래그 하는경우도 이와비슷한 과정을 거친다. 파일의 duration에 기초한 요구되는 미디어 시간을 결정학 컨트롤 판넬에서의 마우스 위치를 결정한다. 마지막으로 실제 미디어타임을 반영시킨다.

```
  private void volumeDragged(int x) {
  if (gain == null)
      return;
  float level = (x - COMPONENTS[VOLUME][0]) / (float) GAINWIDTH;
  if (level < 0f) level = 0f;
  if (level > 1f) level = 1f;
  gain.setLevel(level);
  repaint();
  }
   private void timeDragged(int x) {
  if (player == null)
      return;
  long dura = player.getDuration().getNanoseconds();
  if (dura < 0 || dura > 3 * 3600 * 1000000000L)
      return;
  long nano = (long) ((float) (x - COMPONENTS[MEDIATIME][0]) /
       (COMPONENTS[MEDIATIME][2] -
        COMPONENTS[MEDIATIME][0] + 1) * dura);
  if (nano < 0) nano = 0;
  if (nano > dura) nano = dura;
      player.setMediaTime(new Time(nano));
  repaint();
    }
}
```

이제 남은 부분은 쓰레드로 지속적인 갱신을 시키는 부분이다. 플레이어의 nano second 와 duration을 구하고 시간경과에 따라 계속변하므로 쓰레드로 구현하여 체크한다

```
public void run() {
  while (true) {
      if (player != null) {
```

```
    long nano = player.getMediaTime().getNanoseconds();
    long dura = player.getDuration().getNanoseconds();
    if (dura >= 0 && dura < (long) 3 * 3600 * 1000000000L) {
        timex = (int) (((float) nano / dura) *
         (COMPONENTS[MEDIATIME][2] -
         COMPONENTS[MEDIATIME][0] + 1));
        repaint();
    }
      }
      try {
    Thread.currentThread().sleep(250);
      } catch (InterruptedException ie) {
      }
  }
    }
}
```

마지막으로 paint 부분이다. 실제 그리기작업은 전부 이 부분에서 수행된다. 이 부분의 궁극적인 목적은 볼륨 컨트롤과 미디어타임 컨트롤을 계속해서 업데이트이며, GainLevel 즉 볼륨을 시작적으로 표현하는 부분과 미디어 타임을 시각적으로 표현하는 부분이 구현된다. 플레이어가 미처 생성완료가 되어있지 않았다면 그림에 두줄을 그어 현재 진행이 불가능함을 알린다.

```
public void paint(Graphics g) {
  if (doubleBuffer == null) {
      doubleBuffer = createImage(WIDTH, HEIGHT);
      dg = doubleBuffer.getGraphics();
  }
  dg.drawImage(image, 0, 0, this);
  if (gain != null) {
      int cx = (int) (gain.getLevel() * GAINWIDTH +
       COMPONENTS[VOLUME][0] + 0.5);
      int sy = COMPONENTS[VOLUME][1];
      int ey = COMPONENTS[VOLUME][3];
      dg.setColor(Color.cyan.darker());
      dg.drawLine(cx, sy + 1, cx, ey - 1);
```

```
        dg.drawLine(cx - 1, sy + 2, cx - 1, ey - 2);
        dg.drawLine(cx + 1, sy + 2, cx + 1, ey - 2);
    }
    if (player != null && timex >= 0) {
        int cx = timex + COMPONENTS[MEDIATIME][0];
        int sy = COMPONENTS[MEDIATIME][1];
        int ey = COMPONENTS[MEDIATIME][3];
        dg.setColor(Color.cyan.darker());
        dg.drawLine(cx, sy + 1, cx, ey - 1);
        dg.drawLine(cx - 1, sy + 2, cx - 1, ey - 2);
        dg.drawLine(cx + 1, sy + 2, cx + 1, ey - 2);
    }
    if (player == null) {
        dg.setColor(Color.red);
        dg.drawLine(24, 12, 95, 12);
        dg.drawLine(113, 12, 300, 12);
    }
    g.drawImage(doubleBuffer, 0, 0, this);
      }
}
```

이상으로 JMF 에서 Custom GUI를 구현하는 부분을 살펴보았다. 실제 이 부분은 JMF만이 아니라 일반적인 자바 프로그램에서도 많이 사용되는 방법이기도 하다. 이해의 편의를 위해 두번째 예제인 Swing과 결합되는 부분은 제외하였다. 지난번 예제인 Swing 부분과 이 부분을 결합해 자신만의 커스텀 GUI를 만드는것도 흥미있는 일일것이다..

마치면서

이번 강좌에서는 Player , Processor 이벤트 모델을 처리하기 위한 방법으로 몇가지 예제와 함께 JMF와 Swing의 결합부분, 프로그램내에서의 리소스 해제문제, 그리고 Custom GUI 부분에 대하여 다루었다. 이러한 모든 예제들을 종합하여 여러분 스스로가 좀더 멋진 프로그램으로 완성시켜주기 바란다. 다음 강좌에서는 Processor와 Player를 이용한 오디오와 비디오 데이터의 Capture 기능에 대하여 알아보도록 하자.