<JSTORM>

자바 멀티미디어 JMF

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

Document Information

Document title:	자바 멀티미디어 JMF - 3
Document file name:	JMF3_심빈구.doc
Revision number:	<1.0>
Issued by:	<심빈구> raica@nownuri.net
	정리 : <윤준호> junoyoon@orgio.net
Issue Date:	<2000/9/1>
Status:	final

Content Information

Audience	마이크로 소프트웨어 독자
	자바 멀티미디어 파운데이션 클래스(JFC)의 사용방법을 알고 싶어하는 중급 프로그래머
Abstract	JMF 프로그래밍
	- 지난 연재에 이어 방송국 프로그램에 살을 붙이고 여러 컴포넌트들을 만들어 본다,.
Reference	
Benchmark information	

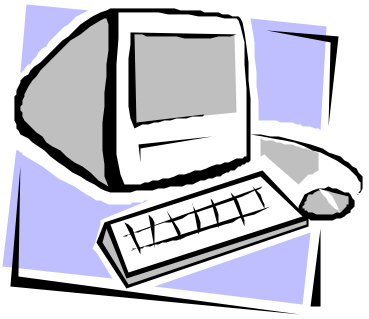
Document Approvals

고문	Signature	date
		
지위	Signature	date

Revision History

Revision	Date	Author	Description of change

Table of Contents



방송국 프로그램에 살을 붙이자!

이번 연재에서는 저번에 예고했던 대로 방송국 프로그램에 살을 붙이는 부분을 하게 된다. 튜너(선국장치)와 새로운 컨트롤 컴포넌트 추가라는 두 개의 커다란 주제를 두고 이번 연재를 진행하게 될 것이다. 막상 이번 연재에 사용할 프로그램을 만들고 나서 그 동안 하나 하나씩 추가된 클래스들을 보니 상당한 양의 프로그램이 되어있어 필자도 약간 놀랐었다. 따라서 먼저 이들 클래스들의 관계를 살펴보고 넘어 가는 과정이 필요할 것 같다.

방송국 프로그램에 사용된 클래스 구분

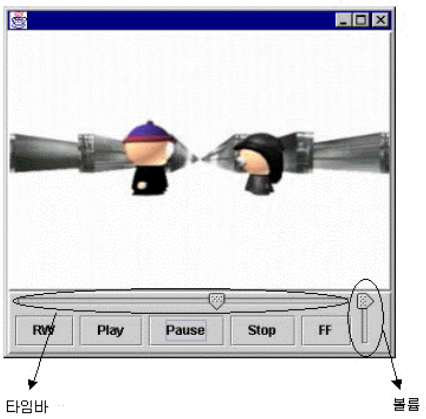
우선 양이 많아진 클래스들을 패키지로 나누어 구분하는 작업을 해 보았다. <표 1>은 그러한 패키지 구분 기준을 나타내었다. 패키지 시작 부분은 Internet Broadcasting System(인터넷 방송국)의 약자인 ibs로 나타내었다. IBS는 이번 연재에서 가끔 나오게 되니 알아두기 바란다.

패키지명	내용
ibs.core	인터넷 방송 프로그램의 주된 클래스들
ibs.core.rtp	RTP에 관계된 클래스들
ibs.extcontrol	커스텀 컨트롤 컴포넌트에 관계된 클래스들
ibs.tuner	튜너에 관계된 클래스들

표 1 패키지 구분 기준

컨트롤 컴포넌트 만들기

저번 연재까지는 JMF에서 제공하는 컨트롤 컴포넌트를 그냥 사용했었다. 워낙 이 컴포넌트가 어디에 붙어도 무난하기 때문에 별로 다른 컴포넌트로 바꾸고 싶은 생각이 들지 않았을 것이다. 하지만 아무리 좋은 보기 좋더라도, 자주 보게 되면 식상해 지기 쉽듯이, 여기서는 JMF에서 제공하는 컨트롤에 질린 독자들을 위해, 새로운 컨트롤 컴포넌트를 만드는 방법을 제공하려고 한다.



필자가 새로운 컨트롤 컴포넌트를 만들려고 했을 때, 가장 먼저 고려했던 부분은 바로 확장성이다. 한가지의 컨트롤 컴포넌트만을 만들려고 했다면, 확장성 같은 것은 문제삼지 않았을 것이다. 하지만 여러 가지 종류의 컨트롤 컴포넌트를 만들어 보려고 생각했었기 때문에 이 부분의 구현에 대해 그렇게, 간단히 넘어가지는 않았다.

그러면 구체적으로 문제점을 짚어 보자, 필자가 만들려고 하는 컨트롤 컴포넌트와 여러분들이 만들려고 하는 컨트롤 컴포넌트는 그 외양만 다르다 뿐이지 기본적인 컨트롤 기능은 모두 같을 것이다. 어떤 사람은 JButton형식의 네모난 버튼을 누르면 Play가 되는 컨트롤 컴포넌트를 만들 것이다(필자가 그렇다). 또 다른 사람은 윈 엠프(Winamp)처럼 삼각형 모양의 버튼을 누르면 Play가 되는 컨트롤 컴포넌트를 만들 것이다. 앞에서 말한 두 개의 컨트롤 컴포넌트는 모니터 상으로 봤을 때에는 확연히 다른 컴포넌트라고 느끼겠지만, 실제 구현 쪽으로 들어가 보면 둘의 코드 패턴은 그다지 다르지 않을 것이라는 것은 누구나 조금만 생각해 보면 알 수 있을 것이다. 당연히 둘 다 사용자의 입력을 받아서 결국엔 Player.start()메소드를 호출하여 미디어를 시작시킬 것이다. 다른 부분도 자세히 살펴본다면 상당히 많은 부분이 중복되어 버린다는 것을 알 수 있을 것이다. 따라서 이렇게 중복되어지는 코드부분을 따로 떼어내는 작업이 우선적으로 필요하게 된다.

필자는, 이 부분에 대해 여러분 스스로가 많은 생각을 해 볼 것을 권한다. 지금 이 순간에 '이렇게 중복되는 부분을 어떤 방식으로 떼어낼 것인가?', 또 '떼어내진 부분은 만들려고 하는 컨트롤 컴포넌트와 어떤 식으로 상호작용을 하게 만들까?'를 여러분이 먼저 한번 생각해 보고 이 다음 부분을 읽어보길 바란다. 실제로 이러한 능력이 프로그램을 얼마나 멋있게 잘 짜는가? 보다 중요하게 여겨지고 있는 능력이다. 따라서 필자가 말하고 싶은 것은 구조적인 프로그램을 짜는 능력을 길으라는 것이다.
필자는 공통부분을 추출하여 다음과 같이 분류해 보았다.

재생위치 표시(타임 바)

- 재생중인 미디어의 재생위치를 사용자에게 알려주어야 한다.
- 사용자의 재생위치 변경 명령을 재생중인 미디어에 반영해야 한다.

볼륨 표시

- 재생중인 미디어의 볼륨 상태를 사용자에게 보여 주어야 한다.
- 사용자의 볼륨 변경을 반영시켜야 한다.

그외 기본 기능

- Play
- Stop
- Pause
- FastForward (FF)
- Rewind (RW)

이것으로 공통부분 추출은 끝났다. 이제 위에서 추출한 공통적인 기능들의 구현을 생각해 보자.

먼저 간단 한 것부터 생각을 해 보자. Play나 Stop, Pause 기능들은 단발성으로 사용자의 액션에 맞추어 주기만 하면 된다. 이 부분의 구현 부분은 <예제 1> 의 생성자 밑 부분의 play(), stop(), pause(), .. 등의 부분이다. 여기서 한가지 주의해야 할 부분은 stop()메소드와 pause()메소드 부분인다. player.stop()메소드는 실제로 미디어를 중지 시키긴 하지만 미디어 타임은 재생되어졌던 곳에서 안 변하기 때문에 stop() 메소드 안에서는 미디어 타임을 0으로 지정해 주는 부분이 추가되어졌다.

이제 타임바의 구현을 살펴볼 차례이다. 앞에서 보았던 기능은 UI쪽에서 일방적인 방향으로 흘러가기 때문에 쉽게 구현이 가능했다. 하지만 타임 바의 구현은 UI쪽에서 사용자가 변경하는 경우와, 재생중인 미디어의 재생시간을 끈임 없이 UI에 반영 시켜 주어야 하는 두가지 경우를 동시에 생각해야 한다. 아래에서 각각의 과정을 나누어 살펴보자.

- UI의 사용자의 변경을 재생중인 미디어에 반영
  - 앞에서의 간단한 기능과 같이 그 부분의 메소드를 제공한다.
- 재생중인 미디어의 시간을 UI에 반영
  - 이 부분은 자바 1.1 이벤트 모델과 비슷한 방법으로 구현했다. 따라서 관계되는 리스너를 정의 했는데, 이 리스너는 아래와 같이 하나의 메소드를 갖고 있다.

```
public interface MediaTimeChangeListener {
   void mediaTimeChanged(long mediaTime, long duration);
}
```

여기서 필자는 Event객체를 넘기지 않고, 바로 변수를 넘기는 방법을 사용했는데, 이와 같이 간단한 정보를 굳이 새로운 Event 객체를 정의하면서 까지 넘기지 않아도 되었기 때문이다.

이러한 MediaTimeChangeListener는 <예제 1>의 (1)부분에 의해 BasicMediaControl에 등

록되어 진다. 그런 후에, 어디에서든 미디어의 시간이 변경되었을 때 불려지는 (2)부분의 fireMediaTimeChange 메소드에 의해 시간 변화를 보고 받게 된다.

이제 시간 변경을 전달 할 수 있는 방법은 만들었지만, 시간의 변경을 어떻게 찾아 낼 수 있을까?

필자는 가장 쉽고도 확실하게, Thread를 생성하여 지속적으로 미디어 시간 변경을 검사하게끔 했다. (3) 부분은 이와 같이 루프를 돌면서 미디어 시간 변경을 체크하는 부분이다.

볼륨의 경우도, 미디어 시간과 동일하게 이루어지니 살펴 보기 바란다. 참고로 볼륨의 경우 GainControl이라는 볼륨 조절 방법을 제공하는 컨트롤을 사용하는데, 이 컨트롤에 대한 자세한 내용은 박스 기사를 참고하기 바란다.

<예제 1> BasicMediaControl

```
public class BasicMediaControl implements Runnable {
  public static final int FF_RW_PERCENT = 3;
   private long  mediaTime = 0;
  private long  duration  = 0;
  private Player  player;
  private GainControl gain;
  private Vector  volumnChangeListenerList  = new Vector(1);
  private Vector  mediaTimeChangeListenerList = new Vector(1);

  public BasicMediaControl (){ }

  /**
   * 컨트롤 하려는 Player 객체를 받는다.
   * 넘겨지는 Player객체는 Realized 상태이어야 한다.
   */
  public BasicMediaControl (Player player) {
    this.player = player;
    gain = (GainControl)player.getControl("javax.media.GainControl");
    duration = player.getDuration().getNanoseconds();
  }

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

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

```
public void stop() {
  player.stop();
  player.setMediaTime(new Time(0));
}
public void ff(){
  long pos = mediaTime + (duration/100)*FF_RW_PERCENT;
  if(pos > duration)
    pos = duration;

  this.setMediaTime(pos);
}

public void rw(){
  long pos = mediaTime - (duration/100)*FF_RW_PERCENT;
  if(pos < 0)
    pos = 0;

  this.setMediaTime(pos);
}

public long getMediaTime() {
  return mediaTime;
}

public void setMediaTime(long time){
  this.mediaTime = time;
  player.setMediaTime(new Time(time));
}

public int getPlayerState(){
  return player.getState();
}

public long getDuration() {
  return duration;
}

public GainControl getGainControl() {
  return gain;
}
```

```
public void setGain(float f){
  gain.setLevel(f);
}

public Player getPlayer(){
  return player;
}

public void addVolumnChangeListener(VolumnChangeListener vcl){    <---- (1)
  volumnChangeListenerList.addElement(vcl);
}

public void addMediaTimeChangeListener(MediaTimeChangeListener mtcl){
  mediaTimeChangeListenerList.addElement(mtcl);
}

protected void fireMediaTimeChange() {                              <---- (2)
  MediaTimeChangeListener mtcl;
  for(int i=0; i < mediaTimeChangeListenerList.size(); i++){
    mtcl = (MediaTimeChangeListener) mediaTimeChangeListenerList.elementAt(i);
    mtcl.mediaTimeChanged(mediaTime, duration);
  }
}

protected void fireGainChange() {
  VolumnChangeListener vtcl;
  for(int i=0; i < volumnChangeListenerList.size(); i++){
    vtcl = (VolumnChangeListener) volumnChangeListenerList.elementAt(i);
    vtcl.volumnChanged(gain.getLevel());
  }
}

public void controllerUpdate(ControllerEvent ce) {
  if (ce instanceof DurationUpdateEvent) {
    this.duration = ((DurationUpdateEvent)ce).getDuration().getNanoseconds();
  }
}

/**
 * 실제로 쓰레드가 돌고 있으면서, 지속적으로 미디어 시간을 체크해
```

```
    * 시간이 변경 되었을때 알려 준다.
    */
   public void run(){
    while (true) {
     if (player != null) {
      long newMediaTime = player.getMediaTime().getNanoseconds();
      if(mediaTime != newMediaTime) {
       mediaTime = newMediaTime;
       fireMediaTimeChange();                                    <------ (3)
      }
     }
    try {
    Thread.currentThread().sleep(500);
    } catch (InterruptedException ie) {
    }
    }
   }
}
```

BasicMediaControl을 이용하여 컨트롤 컴포넌트 만들기

<예제 2>는 <그림 1>에서 보는 컨트롤 컴포넌트의 구현 코드를 조금 편집 한 코드이다. <예제 2>를 보면서 위에서 만들어 놓은 BasicMediaControl을 어떻게 사용하여 컨트롤 컴포넌트를 만드는지 알아 보자.

우선 (1)이라고 표시한 부분에서는. SampleControlComponent가 앞에서 정의한 VolumnChangeListener과 MediaTimeChangeListener을 Implements 한다고 선언되었다. 이들 리스너 들은 (3), (4) 부분에서 컴포넌트에 맞게 구현 되었다. 여기에서는 넘겨진 값을 각각의 뷰에 반영시키는 일을 하게 된다. 그리고, 마지막으로 (2)부분에서는 BasicMediaControl의 인스턴스인 control에 두 리스너를 등록한다.

그 다음은 (5) 부분을 보자. 버튼의 액션을 다루는 ActionListener 구현 부분인데, 이 부분에서는 사용자의 입력에 맞게 단순히 BasicMediaControl의 해당 메소드를 호출하기만 한다.

이것으로 새로운 컨트롤 컴포넌트의 작성은 끝났다. 수치상의 계산이 좀 복잡했을 뿐이지 그리 어렵지 않게 만들 수 있었다. 여러분도 여러분만의 컨트롤 컴포넌트를 쉽게 만들어낼 수 있을 것이다.

<예제 2> SampleControlComponent

```
public class SampleControlComponent extends JPanel implements
                VolumnChangeListener, MediaTimeChangeListener,      <----- (1)
                                        ActionListener{

   public static final String PLAY_BTN_ACTION_COMMAND      = "Play";
   public static final String PAUSE_BTN_ACTION_COMMAND     = "Pause";
   public static final String STOP_BTN_ACTION_COMMAND      = "Stop";
   public static final String FF_BTN_ACTION_COMMAND        = "FF";
   public static final String RW_RTP_BTN_ACTION_COMMAND    = "RW";


   JSlider mediaTime;
   JSlider volumn;
   BasicMediaControl control;
   Thread  controlThread;

   public SampleControlComponent(Player player){
      control = new BasicMediaControl(player);
      initComponent();
      control.addMediaTimeChangeListener(this);      <------(2)
      control.addVolumnChangeListener(this);
      controlThread = new Thread(control);
      controlThread.start();
   }
   public void initComponent(){
      .....
      버튼들을 초기화 한다.
      .....

      // 타임 바(mediaTime) 초기화
      mediaTime = new JSlider(JSlider.HORIZONTAL);
      mediaTime.setPaintTicks(false);  // 눈금을 표시하지 않는다.
      mediaTime.setMaximum((int)(control.getDuration()/1000000000L));
      mediaTime.setValue(0);
       // 사용자의 타임바 변경을 받는다.
      mediaTime.addChangeListener(new ChangeListener(){
         public void stateChanged(ChangeEvent e){
            int changedValue = mediaTime.getValue();
```

```
            int oldValue = (int)(control.getMediaTime()/1000000000L);
            if(!(changedValue > oldValue -2 && changedValue < oldValue + 2)){
               control.setMediaTime((long)(changedValue*1000000000L));
            }
         }
      });

      // 볼륨조절 초기화
      volumn = new JSlider(JSlider.VERTICAL);
      volumn.setPaintTicks(false);
      volumn.setMaximum(10);
      volumn.setValue((int)(control.getGainControl().getLevel()*10));
      volumn.setPreferredSize(new Dimension(30, 40));
      // 사용자의 볼륨 조절을 받는다.
      volumn.addChangeListener(new ChangeListener(){
         public void stateChanged(ChangeEvent e){
            int changedValue = volumn.getValue();
            control.setGain(((float)changedValue)/10);
         }
      });
      .......
      컴포넌트들 추가
      .......
   }

   public void volumnChanged(float changedVolumn){<--- (3)
      volumn.setValue((int)(changedVolumn*10));
   }

   public void mediaTimeChanged(long mediaTime, long duration){  <----(4)
      int dura = (int)(duration/1000000000L);
      int mTime = (int)(mediaTime/1000000000L);
      mediaTime.setMaximum(dura);
      mediaTime.setValue(mTime);
   }
   public void actionPerformed(ActionEvent e){         <----(5)
      String actionCommand = e.getActionCommand();

      if (actionCommand.equals(PLAY_BTN_ACTION_COMMAND))    {
         control.play();
      }else if(actionCommand.equals(STOP_BTN_ACTION_COMMAND))    {
```

```
            control.stop();
        }else if(actionCommand.equals(PAUSE_BTN_ACTION_COMMAND))   {
            control.pause();
        }else if(actionCommand.equals(FF_BTN_ACTION_COMMAND))      {
            control.ff();
        }else if(actionCommand.equals(RW_RTP_BTN_ACTION_COMMAND))  {
            control.rw();
        }
    }

}
```

SampleControlComponent 추가 시키기

위에서 만들어진 컨트롤 컴포넌트를 말 그대로 컴포넌트이기 때문에 생성을 하여 원하는 곳에 추가시키기만 하면 된다. 아래 코드는 MediaPlayer클래스에서 SampleControlComponent를 추가 시키는 부분의 코드 이다.

```
displayFrame.setControlComponent(
        new SampleControlComponent(player));
```

이렇게 만들어진 클래스들은 <그림 2>와 같은 관계를 갖는다.

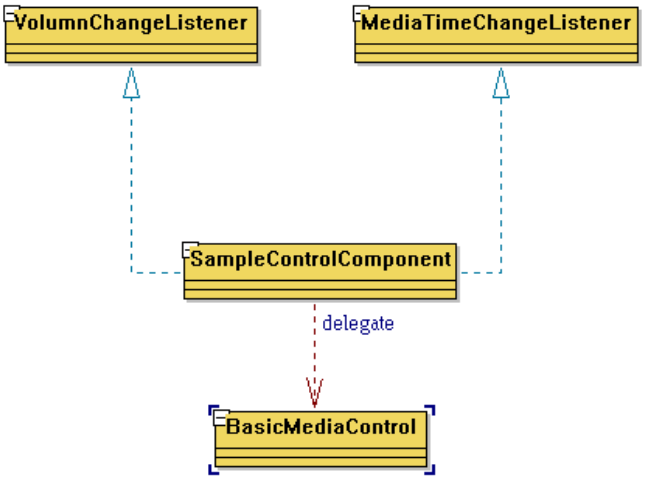


그림 1 컨트롤 컴포넌트를 구성하는 클래스들

선국장치 만들기

이번에 다루려고 하는 부분은 JMF 패키지와는 그렇게 상관이 없는 부분이다. 하지만 클라이언트 측의 편의를 위해서는 빼놓을 수 없는 부분이기도 하다. 바로 방송을 선택할 수 있는 선국 장치에 대한 부분이다.

실제로 현재까지 구현되어진 방송국 프로그램은 방송을 듣기 위해선 방송국의 서버와 포트를 사용자가 직접 입력해 주어야 했었다. 이러한 작업은 상당히 불편한 방법이기 때문에, 사용자에게 <그림 3>과 같이 방송을 목록화 하여 보여주고 그 중에서 원하는 방송을 선택하여 볼 수 있는 기능을 제공하려고 한다.

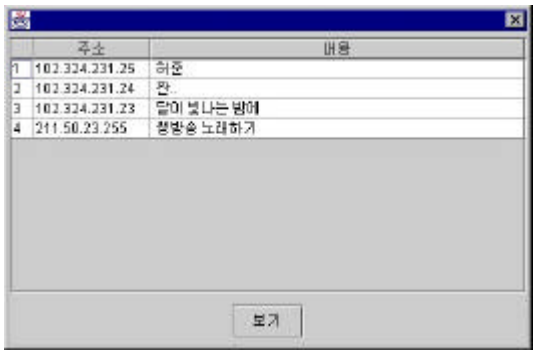


그림 2 튜너에 등록된 방송국 리스트

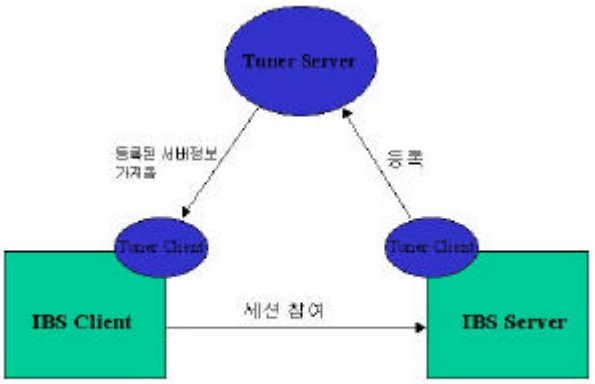


그림 3 IBS에서 사용된 Tuner들의 관계

선국 장치의 구성은 <그림 4>에서 보는 바와 같다. IBS 서버와 클라이언트는 튜너 클라이언트를 이용하여 튜너 서버와의 통신을 하게 된다.

아래는 튜너와 IBS간의 작업을 나누어 각 작업별로 이루어지는 과정을 순차적으로 짚어 보았다. 각 과정의 괄호 안에는 해당 과정이 이루어지는 영역을 표시해 두었다.

방송 등록

✍✍ **새로운 방송 시작**(IBS Server) -> Tuner Server**에 방송 등록 요청**(Tuner Client) -> **요청된 방송을 등록**(Tuner Server)

방송 삭제

✍✍ **방송 종료**(IBS Server) -> Tunser Server**에 방송 삭제 요청**(Tuner Client) -> **요청된 방송 삭제**(Tuner Server)

등록된 방송 보기

✍✍ **등록된 방송 보기**(Tunser Client) -> **등록된 방송 리스트 요청**(Tuner Client) ->**요청된 방송 리스트 전송**(Tuner Server) -> **전송된 방송 리스트 획득**(Tuner Client)

위에서 보듯이 튜너 서버와 클라이언트 간의 통신 내용은 상당히 간단하기 때문에 구현 방법으로 소켓(Socket)**이라는 방법을 선택했다. 소켓을 선택했기 때문에 먼저 통신 규약**(Protocol)**을 설정할 필요가 있는데,** <표 2>**은 튜너 서버와 클라이언트 사이에 주고받는 스트림의 형식을 정의해 놓은 표이다.**

스트림 형식	작업명:메시지
메시지 형식	방송주소,방송설명|방송주소,방송설명|방송주소,방송설명|...
예	1:211.50.63.255,최신가요|211.50.64.255,짱구는 못말려

표 2 선국장치 프로토콜

표에서 보는 바와 같이 전송되는 스트림은 작업을 지시해 주는 부분과, 작업에 대한 정보를 담는 메시지 부분, 이렇게 두 가지 부분으로 나뉘게 된다. 이 두 부분은 ':'에 의해 나누어 진다. 메시지 부분은 작업에 대한 정보를 담는 부분인데, 실제로 작업에 대한 정보란 '어떠한 방송을 등록 시켜 달라', '어떠한 방송을 삭제 시켜 달라'등 방송에 대한 정보이기 때문에 방송을 담을 수 있게 만들었다. 각각의 방송은 '|'에 의해 나누어지고, 방송은 세부적으로 ','에 의해 방송 주소와, 방송 설명으로 나누어 진다.

이제 이런 형식에 의해 전송된 스트림을 나누어주는 작업이 필요한데, 그 작업은 OpCode**부분을 떼어 내는 부분 메시지를 떼어 내는 부분 그리고, 떼어낸 메시지를 갖고 방송국 정보를 만드는 부분 이렇게 세 가지 부분으로 나뉘게 된다. 이러한 부분들은 예제** 2**에 나타낸** TunerProtocol**에서 구현해 놓았다.**

예제 2: TunerProtocol

```
public class TunerProtocol{
    public static final String OP_DELIMITER = ":";
    public static final String BROADCASTERINFO_DELIMITER = "|";
    public static final String DESCRIPTION_DELIMITER = ",";
```

```
    // 넘겨받은 스트링에서 Operation 부분을 떼어 낸다.
    public static String parseOPCode(String str){
        StringTokenizer st = new StringTokenizer(str, OP_DELIMITER);
          if(st.hasMoreElements()){
            return st.nextToken();
        }
        return null;
    }

  // 넘겨받은 스트링에서 Message 부분을 떼어 낸다.
    // opcode:message.....
    public static String parseMessage(String str){
        StringTokenizer st = new StringTokenizer(str, OP_DELIMITER);
        if(st.hasMoreElements()){
            st.nextToken();
            if(st.hasMoreElements()){
                return st.nextToken();
            }
        }
        return "";
    }

    // 넘겨받은 스트링을 분석해 방송 정보를 담는 BroadcasterInfo 에 넣어 리턴한다.
    // address,description|address,description|address,description
    public static Vector parseBroadcasterInfo(String str){
        StringTokenizer mainST = new StringTokenizer(str, BROADCASTERINFO_DELIMITER);
        Vector result = new Vector(2);
        BroadcasterInfo bInfo;
        // 방송국 정보가 없을때 까지
        while(mainST.hasMoreElements()){
            StringTokenizer subST = new StringTokenizer(mainST.nextToken(),
                                        DESCRIPTION_DELIMITER);
            bInfo = new BroadcasterInfo();
            bInfo.setAddress(subST.nextToken());
            bInfo.setDescription(subST.nextToken());
            result.addElement(bInfo);
        }
        return result;
    }
}
```

튜너 서버

이제 위에서 정의한 TunerProtocol을 이용해 튜너에서는 어떻게 통신을 하는지 알아 볼 것이다. 아래 코드는 위에서 정의한 TunerProtocol을 이용해 실질적으로 통신을 제어하는 부분이다. 우선 소켓에서 스트림을 읽어온 후, parseOPCode() 메소드를 호출하여 OP 코드를 추출해 낸다. 그 후에는 OP코드에 맞추어 분기하여 작업을 수행하게 된다.

```
String message = "";
try{
   message = input.readUTF();
}catch(Exception e){
   e.printStackTrace();
}

String opCode;
opCode = TunerProtocol.parseOPCode(message);
if(opCode != null){
   if(opCode.equals(TunerServer.OP_ADD_SERVER)){
      addServer(TunerProtocol.parseMessage(message));
   }
   else if(opCode.equals(TunerServer.OP_REMOVE_SERVER)){
      removeServer(TunerProtocol.parseMessage(message));
   }
   else if(opCode.equals(TunerServer.OP_GET_SERVER_INFO)){
      getServerInfo();
   }
}
```

튜너 클라이언트

튜너 클라이언트의 기본 개념은 가벼운 네트웍 연결에 있다. 튜너 클라이언트는 그 특성상 컨넥션을 계속 갖고 있을 필요가 없다. 따라서 필요할 때마다 컨넥션을 열고, 작업이 끝나면 컨넥션을 끊는 형식으로 작동하게 된다.

아래 코드는 튜너 클라이언트의 몇 개의 메소드 중에서 서버의 정보를 받아 오는 부분이다. 먼저 openSocket()메소드를 호출하여 소켓을 열고, 서버 정보를 보내 달라는 메시지를 보낸다. 이 메시지는 OP코드만이 필요하기 때문에 OPCode에 ':' 붙여서 서버 쪽에 보냈다. 그 다음으로는 서버쪽에서 답신한 메시지를 받아서 parseMessage()메소드를 이용해 메시지 부분을 분리한 뒤 parseBroadcasterInfo()메소드를 이용해 서버쪽에서 넘겨준 방송 정보를 Vector형으로 획득한다.

```
String message = "";
try{
        Socket socket = openSocket();
        this.input = new DataInputStream(socket.getInputStream());
        this.output = new DataOutputStream(socket.getOutputStream());
        output.writeUTF(TunerServer.OP_GET_SERVER_INFO+":");

        // 서버 정보를 받아 온다.
        message = input.readUTF();
        socket.close();
}catch(Exception e){
        e.printStackTrace();

        return new Vector();
  }
Vector bInfos = TunerProtocol.parseBroadcasterInfo(
                        TunerProtocol.parseMessage(message));

return bInfos;
```

튜너의 구현에서 생성된 클래스들의 관계를 클래스 다이어그램으로 나타내 보면 아래와 같다.

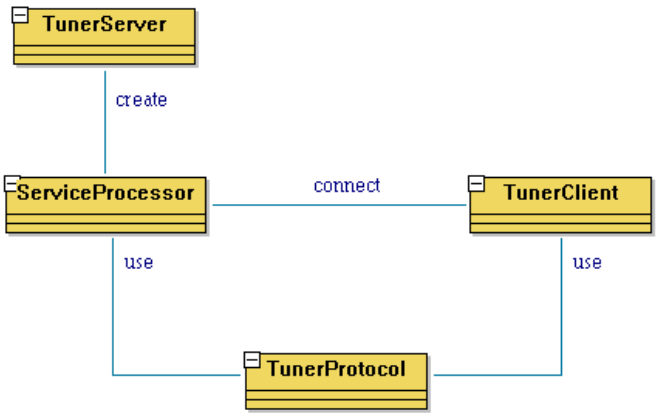


그림 4 튜너를 구성하는 클래스들

JMF의 다음 버전에서는..

연재를 진행하면서 개인적으로 JMF를 더 깊게 알 수 있는 기회가 되었지만 한편으로는 2.0이라는 버전에 걸맞지 않게 불안정한 JMF에 아쉬움도 남는다. JAVA의 멀티미디어 쪽에 공개된 패키지 중 유일한 패키지라는 것이 그 아쉬움을 더 하게 한다. 하지만 Sun의 JMF 개발팀은 IBM이라는 든든한 파트너를 만나게 되었으니 다음 버전에서는 좀더 안정적이고, 확장된 JMF패키지를 내 놓지 않을까 기대한다. 실제로 JMF 홈페이지에 얼마전 올라온 대화 내용에는 이러한 부분에 대한 언급도 되어 있으니 기대 해 볼만하다.

연재를 마치며

3회에 걸친 연재를 이번으로 마친다고 생각하니 시원함보다는 아쉬움이 많이 남는다. 처음 생각했었던 바와는 달리 방송국이라는 프로그램을 만들면서 JMF에 대한 내용을 진행시키다 보니 연관되지 않은 JMF의 나머지 부분은 너무 소홀히 다룬 것 같다. 이번 연재에서 미처 다루지 못 한 내용은 필자가 속한 스터디 그룹인 JStorm 홈페이지 (http://www.jstorm.pe.kr)에 계속 올릴 예정이니 자주 찾아와 주기 바란다. 또한 메일로의 질문도 언제든지 환영한다. 멀티미디어라는 아주 흥미로운 주제를 더욱 더 흥미롭게 접할 수 있었으면 하는 바램이다.

JMF의 Control들 정리

JMF에서는 특정한 객체에 어떠한 값을 세팅하거나, 또는 객체의 속성 값을 얻어올 수 있는 방법을 Control이라는 인터페이스를 통해 제공한다. 몇몇 컨트롤들은 UI 컴포넌트를 제공하여 그 값들을 컴포넌트를 통하여 제어할 수 있게도 한다. Controller 객체, DataSource 객체, DataSink 객체등, 많은 JMF 객체들은 Control들을 제공한다. 아래 표는 JMF의 javax.media.control 패키지의 Control들을 간단한 설명과 함께 정리한 표이다

Control 명	설명
BitRateControl	비트 전송률을 조절한다.
BufferControl	버퍼의 길이와 Threshold등 버퍼링을 조절한다.
FormatControl	지원하는 포맷등을 세팅 하기 위해 제공된다.
FrameGrabbingControl	비디오에서 장면(프레임)을 잡아내는데 이용된다.
FramePositioningControl	비디오 내에서 프레임을 지정하거나 하는 데 이용된다.
FrameProcessingControl	
FrameRateControl	단위 시간동안 보여줄 프레임 수를 조정한다.
H261Control	H.261 비디오 코덱을 위한 컨트롤을 제공
H263Control	H.263 비디오 코덱을 위한 컨트롤을 제공
KeyFrameControl	Key 가 되는 프레임의 간격 조절에 관한 기능 제공
MonitorControl	캡쳐 하거나 encoding 할 때 그 상황을 모니터 하는 기능 제공
MpegAudioControl	Mpeg 오디오에 대한 컨트롤 제공
PacketSizeControl	패킷의 크기를 정하는 컨트롤 제공
PortControl	입력이나 출력 장치를 조절하는 컨트롤 제공
QualityControl	질을 조절하는 컨트롤 제공
SilenceSuppressionControl	음 소거를 조절하는 컨트롤 제공
StreamWriterControl	DataSink나 Multiplexer의 출력 스트림 양 조절하는 컨트롤 제공
TrackControl	트랙을 조절하는 컨트롤 제공

컨트롤 획득 방법

컨트롤을 갖고 있는 객체는 아래와 컨트롤을 리턴 해주는 아래와 같은 두 가지의 메소드를 갖고 있다.

getControl(String controlType)
(사용 예 : getControl("javax.media.control.PortControl");
getControls()

유용한 컨트롤 정리

BufferControl

버퍼에 한 조절을 한다. Capture나 RTP에서 많이 사용되는 컨트롤 이다.

FrameGrabbingControl

Render나, Player의 getControl() 메소드를 통해 얻을 수 있다. 이름에서 알 수 있듯이 비디오 스트림에서 하나의 프레임(Still Video Frame)을 뽑아 낼 때 사용한다. 이때 넘겨지는 값은 Buffer형 인데 , ImageConverter 클래스를 이용하여 자바에서 사용가능한 이미지 형으로 변경할 수 있다.

PortControl

입력이나 출력 장치를 지정해 주는 컨트롤이다. 마이크, 스피커, 헤드폰, CD등의 입출력 장치들은 PortControl의 스태틱 변수로 선언되어 있다.

QualityControl

0.0에서 1.0 사이의 값으로 미디어의 질을 조절한다. 미디어의 질과 CPU사에는 Trade-off관계가 있음을 명심하자.

GainControl

현재 재생중인 미디어의 오디오 볼륨을 조절하는 기능을 제공한다. setLevel getLevel, setDB, getDB의 메소드를 통해 볼륨을 조절 할 수 있고, setMute 메소드를 통해 음소거 등의 기능을 제공한다.

캡쳐 작업에 유용하게 사용할 수 있는 CaptureDevice 인터페이스

CaptureDevice 인터페이스는 캡쳐된 데이터를 다루는데 편하도록 DataSource에 몇 가지 기능을 추가해서 정의한 DataSource형 객체이다. 아래 표는 CaptureDevice의 메소드를 나타내고 있다.

메소드	설명
connect()	캡쳐 장치를 초기화 한다.
disconnect()	캡쳐 장치를 닫는다.
getCaptureDeviceInfo	캡쳐 장치의 정보를 CaptureDeviceInfo 형으로 리턴한다.
getFormatControl	캡쳐된 미디어 스트림들 각각의 FormatControl객체를 넘긴다.
start	캡쳐를 시작한다.
stop	캡쳐를 멈춘다.

connect() 메소드를 호출하는 순간 캡쳐 장치는 정해진 포맷으로 초기화된다. 이때 에는 캡쳐 장치에 대한 자원 또한 취득하게 되는데, 이 자원은 독점적인 자원이기 때문에, 다른 곳에서 중복하여 캡쳐 장치를 접근할 수 없게 된다. disconnect()메소드는 connect() 메소드에서 잡고 있던 자원을 풀어 주는 역할을 하게 된다. 따라서 이 메소드 호출 후에는 다른 곳에서 캡쳐 장치 접근이 가능하게 된다.

getFormatControls() 메소드는 FormatControl객체를 넘겨 주는데, 이 컨트롤은 캡쳐된 미디어 스트림의 포맷을 결정할 수 있는 메소드를 제공한다.(1회 기사 참조) 따라서 캡쳐된 미디어를 어떤 포맷으로 다룰 것인지를 결정할 수 있게 해 준다. 하지만 이 메소드를 사용할 때 한가지 주의할 점이 있는데, 이러한 포맷 변경 작업은 connect()메소드 호출 전에만 가능하다는 것이다.

마지막으로 start()메소드와 stop()메소드는 캡쳐작업을 조절하는 역할을 하는데, DataSource의 start()메소드와 stop()메소드와 그 역할이 비슷하다. CaptureDevice의 start메소드는 캡쳐를 시작하여 미디어 스트림을 생산해 내고, DataSource의 start메소드는 미디어를 읽어 들여 미디어 스트림을 생산해 낸다. stop 메소드는 이러한 작업을 중지하게 된다.