



Curso Multimedia Home Platform 1.1.2

Java Media Control

Presentando media con Java Media Framework

Curso Multimedia Home Platform 1.1.2

Copyright 2008 © Enrique Pérez Gil

Licensed under the ***Creative Commons Attribution-Non-Commercial-No Derivative Works 3.0 Unported License***. You may not use this file except in compliance with the License. You may obtain a copy of the License at:

<http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode>

This is a human-readable summary of the License applied:

(<http://creativecommons.org/licenses/by-nc-nd/3.0/>)

You are free to Share, to copy, distribute and transmit the work **Under the following conditions:**

- **Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Noncommercial.** You may not use this work for commercial purposes.
- **No Derivative Works.** You may not alter, transform, or build upon this work.

For any reuse or distribution, you must make clear to others the license terms of this work. Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.

Introducción

- Se elige JMF 1.0 (MHP no pretende hacer nuevos APIS)
- JMF 1.0 se centra en presentar contenido que proviene tanto de servers como de local, sin embargo en el contexto en el que nos encontramos es necesario efectuar algunos ajustes porque, por ejemplo, no estamos ante reproducción de contenido estático, no podemos parar, rebobinar...etc..

Esquema general de funcionamiento

- El elemento que se encarga de **decodificar** es la clase: **javax.media.Player**
- Imaginemos un reproductor de DVD casero, el componente **javax.media.Player** sería la parte del mismo que se dedica estrictamente a decodificar y presentar. A un Player se le pueden añadir **controles** para por ejemplo, parar la presentación, o que salte a un punto, etc...estos controles en JMF se denominan **javax.media.Control**
- Y ¿ qué es lo que lee un **javax.media.Player** ? Al igual que un reproductor de DVD lee de un DVD un Player leerá un **javax.media.protocol.DataSource** (algo parecido a las conexiones JDBC de algunos componentes visuales de la época...☺)
- En JMF existen **dos tipos de DataSources** dependiendo de si el DataSource tiene que solicitar los datos para que se los manden o por el contrario una vez que se “enchufa” a la fuente estos llegan sin mas. Los tipos son **PULL** y **PUSH**. **Broadcast sería PUSH**.
- En JMF esta distinción entre **PUSH** y **PULL** Datasources **no se manifiesta al programador**, o mejor dicho, **se oculta su complejidad**

Locators. Recordemos

- Locator de JMF: **javax.media.MediaLocator**. Ya vimos “algo” al respecto:
 - Grupo MEDIA: Sólo se usa para control de presentación de Media. La clase de davic se implementa para unirlo con el Grupo TV: así se puede tomar un Locator del Grupo TV y construir fácilmente un Player para verlo
 - Para referenciar contenido Media por parte de Java Media Control API usaremos
 - > **javax.media.MediaLocator**
 - > **org.davic.media.MediaLocator** extends javax.media.MediaLocator

Locators Classes. Recordemos

- Existen dos grupos diferenciados:

| <p style="text-align: center;">Grupo MEDIA</p> <p>Sólo se usa para control de presentación de Media por parte de Java Media Control API: JMF</p> <p style="text-align: center;">Para Services, Audio y VideoDrips</p> | <p style="text-align: center;">Grupo TV</p> |
|---|--|
| <p>javax.media.MediaLocator</p> | <p>javax.tv.locator.Locator</p> <p> puede referirse a cualquier contenido Media que se distribuya en un DTV: TS, Service, File</p> |
| <p>org.davic.media.MediaLocator</p> <pre>public class MediaLocator extends javax.media.MediaLocator { public MediaLocator(org.davic.net.Locator loc) }</pre> | <p>Para elementos referenciables como dvb://....</p> <p>org.davic.net.Locator (implements javax.tv.locator.Locator)</p> <p>org.davic.net.dvb.DvbLocator (extends Locator)</p> <p>org.davic.net.dvb.DvbNetworkBoundLocator (ext.DvbLocator)</p> |
| <p style="text-align: center;">La clase org.davic.media.MediaLocator conecta el Grupo TV con el Grupo MEDIA puede construirse con un org.davic.net.Locator: así se puede tomar un Locator del Grupo TV y construir fácilmente un Player para verlo con JMF!!!</p> | |

Creando los Players. `javax.media.Manager`

- ¿ **Cómo se crean los Players y los Datasources?** A través de `javax.media.Manager`.
- **Ni los Players ni los DataSource** se pueden crear directamente! Ambos se obtienen usando `javax.media.Manager`.
- También `javax.media.Manager` es el punto de acceso al system `javax.media.TimeBase`: una referencia constante de tiempo, digamos que es el reloj de referencia.

Veamos el API

Creando los Players. javax.media.Manager

- **API. Creando Players:**

- public static javax.media.Player **createPlayer**(javax.media.protocol.**DataSource** source)
 - Creará el Player a partir del Datasource y el tipo de este: DataSource.getContentType().
 - **Si NO ENCUENTRA un player** que sepa manejar el tipo de contenido que se va a recibir lanzará una Exception.
- public static javax.media.Player **createPlayer**(java.net.URL sourceURL)
 - Creará el Player a partir de la URL,
 - **OJO: en realidad NO es así ->**
 - crea internamente un MediaLocator con la URL.
 - Intenta crear el Player partiendo del Locator, pero....ver siguiente método
- public static javax.media.Player **createPlayer**(javax.media.MediaLocator locator)
 - Creará el Player a partir de un Locator
 - **OJO: tampoco es exactamente así:**
 - Primero buscará una clase DataSource que sepa obtener los datos con el protocolo utilizado en el Locator
 - Instanciará un DataSource a partir de la clase encontrada y le asignará el Locator.
 - Intentará crear el Player internamente con el DataSource.

- **Por lo tanto: el importante es el que recibe el DataSource.**

Creando los Players. javax.media.Manager

- **API. Creando Datasources:**

- public static javax.media.protocol.DataSource **createDataSource**(java.net.URL url)
 - Creando un DataSource a partir de una URL
- public static javax.media.protocol.DataSource **createDataSource**(
javax.media.MediaLocator locator)
 - Creando un DataSource a partir de un Locator

OJO: En ambos métodos **si NO hay DataSource** que sepa manejar el contenido lanzará una Exception.

- **API. Accediendo al TimeBase**

- public static javax.media.TimeBase **getSystemTimeBase**()

Donde TimeBase es:

```
public interface TimeBase {  
    public Time getTime(); -> tiempo actual.  
    public long getNanoseconds(); -> tiempo actual en nanos.  
}
```

Creando los Players. javax.media.Manager

- Como hemos visto, para que **javax.media.Manager** nos pueda ofrecer un Player necesitará:
 - Disponer de uno que entienda el tipo de información que va a presentar(audio, dvbservice..)
 - Disponer de un Datasource que sepa como obtener los datos del contenido usando el protocolo que le han dicho.(dvb, file, http...)
- Los métodos de Manager siguientes nos permiten saber para un protocolo y un tipo de contenido los tipos de **Datasources** y **Handlers/Players** existentes respectivamente. (Aunque en el caso de los Handlers y al menos para el deco Strong 5510 MHP no está muy claro, como veremos).
 - public static java.util.Vector **getDataSourceList**(java.lang.String protocol)
 - Ofrece una lista de Clases Manejadoras de Datasources en función del nombre del **protocolo** usado.
 - public static java.util.Vector **getHandlerClassList**(java.lang.String contentType)
 - Ofrece una lista de clases manejadoras de **tipo de contenido**.

Ejercicios Bloque JMF-1

Creando los Players. javax.media.Manager

- ¿ Cual será el Handler/Player para un DVB Service ? Recordemos de JavaTV Selection

API javax.tv.service.selection.ServiceContext

- public ServiceContentHandler[] **getServiceContentHandlers()**
 - Ofrece la lista de los ServiceContentHandler del servicio: **devolverá uno del tipo ServiceMediaHandler que se encarga de la toda la presentación de tipo Media, si es que se está presentando media, y uno o varios encargados de gestionar los Xlet en ejecución, si es que los hay**

- Y **javax.tv.service.selection.ServiceMediaHandler** es un **Player!!!**

```
public interface ServiceMediaHandler extends  
    javax.media.Player, javax.tv.service.selection.ServiceContentHandler{  
}
```

Creando los Players. javax.media.Manager

- Como hemos visto para construir un player para presentar un contenido se debe conocer por un lado “como viene”, el protocolo, y por otro el tipo del mismo: audio, DVB Service....
- Un DataSource deberá se capaz de identificar el content type a partir de los datos de que disponga: extensión de fichero, signalling(content_type_descriptor), URL.getContentType(), abriendo el contenido y analizándolo...

¿ Qué vamos a poder gestionar con JMF ?

- Recordemos los APIS que consumen media en el contexto de DVB-J:

Table 139: Media type support required in Enhanced and Interactive Broadcast profile 1

| Media type | Reference | API(s) |
|---------------------|----------------|---|
| Downloadable fonts | 7.4 | <code>org.dvb.ui.FontFactory</code> |
| Audio from file | 7.1.4 | <code>org.havi.ui.HSound</code> JMF only with references to files |
| MPEG I frame images | 7.1.1 | <code>org.havi.ui.HBackgroundImage</code> |
| PNG images | 7.1.1.3 | <code>java.awt.Image</code> |
| JPEG images | 7.1.1.2 | <code>java.awt.Image</code> |
| DVB service | EN 300 468 [4] | JMF only with references to DVB services for playback direct from the network |
| Video "drips" | 7.1.3 | <code>org.dvb.media.DripFeedDataSource</code> |
| Text files | 7.1.5 | All Java APIs supporting reading or writing text files. |
| Subtitles | 7.2.3 | JMF only as part of a DVB service |

¿ Qué vamos a poder gestionar con JMF ?

- En resumen:
 - **Audio (MPEG-1 Audio Layers 1, 2):**
 - Desde ficheros y
 - Proveniente del Broadcast como parte de un DVB Service
 - **MPEG Video:** sólo proveniente del Broadcast **como parte de un DVB Service**
 - **DVB Service** (quien sabe cual es el Player de este? ☺): como un todo
 - **Subtitles:** sólo proveniente del Broadcast como parte de un DVB Service.
 - **Videodrips:** sólo desde memoria. En la capa de Video únicamente.
- **Es decir, los Players presentarán sólo:**
 - **DVB Services.** (Protocolo dvb)
 - **Audio Files desde Ficheros.** (Protocolo file)
 - **VideoDrips desde memoria y solo en la capa de Video.** (Protocolo dripfeed:)
- **Respecto al sonido:** Fuera del contexto de JMF es posible reproducir sonido proveniente de una URL con el componente **org.havi.ui.HSound** (siempre y cuando tengáis los permisos adecuados de socket)

Veamos los APIS ahora que sabemos lo que hay.

Los APIS. javax.media.protocol.DataSource

- El API.
 - Los 3 métodos siguientes **no los usaremos directamente**, recordemos que un **DataSource se crea con Manager y sólo pasándole una URL o un javax.media.MediaLocator**, del cual hereda nuestro magnífico **org.davic.media.MediaLocator**:

```
public DataSource() (ayuda para la instanciación dinámica)
public DataSource(javax.media.MediaLocator source)
public void setLocator( javax.media.MediaLocator source) Solo se puede establecer una vez (es lógico si tenemos en cuenta la complejidad de su creación: compatible con la fuente de datos y con el Player
```
 - `public javax.media.MediaLocator getLocator()`
 - `public String getContentType();`
 - Descripción del tipo de contenido que se obtiene con este datasource.

javax.media.protocol.DataSource

- El API. Obteniendo los Control..... **javax.media.Control**

public class DataSource implements **javax.media.protocol.Controls**, javax.media.Duration

- Implementa este Interface el cual proporciona **dos métodos** interesantes que vemos a continuación.
 - Object[] getControls()
 - Nos ofrecerá la lista de objetos de tipo **javax.media.Control** con los que podremos “manejar” la reproducción del contenido. Mas adelante los veremos con más detalle.
 - Object **getControl**(String controlInterface)
 - Nos ofrecerá un **javax.media.Control** para manejar el elemento DataSource sobre el que hacemos la llamada que implemente el interface pasado.
 - Este método es mas bien de uso interno.

javax.media.protocol.DataSource

- El API. Ciclo de Vida.....
 - public void **connect()** throws java.io.IOException;
 - Inicia la comunicación. **OJO: reserva recursos caros. Hacedlo cuando sea oportuno**
 - public void **disconnect()**;
 - Desconecta y **libera recursos!**
 - public void **start()** throws java.io.IOException;
 - **Después de connect se llama a start para que comience la transmisión de datos**
 - public void **stop()** throws java.io.IOException;
 - Detiene la transmisión de datos

javax.media.protocol.DataSource

- El API. ¿ cuanto dura el fichero de audio cargado ?.....

public class DataSource implements javax.media.protocol.Controls, **javax.media.Duration**

- Implementa este Interface el cual proporciona información respecto a la duración de la presentación del contenido.

– public javax.media.Time getDuration()

Pero....¿ realmente vamos a usar el API Datasource ?

NO

en realidad el API que acabamos de ver va dirigido más a implementadores, porque con este API quien se va a comunicar mayormente de manera interna es el PLAYER. Que es con quien nosotros hablaremos

Sigamos

javax.media.Player

- Sabemos qué es un DataSource, sabemos los Locators que necesitamos, sabemos que tipo de contenido se puede mostrar, veamos ahora el elemento que lo muestra y qué podemos hacer con él (controls...)

javax.media.Player

Player implements javax.media.**MediaHandler**, javax.media.**Controller**, javax.media.**Duration**

- La clase Player sólo ofrece los siguientes métodos, que describimos brevemente.
 - public Component **getVisualComponent()**;
 - Me devuelve un componente visual si tiene contenido visual que presentar.
 - public GainControl **getGainControl()**;
 - Me devuelve un GainControl, que me ofrece un API para manejar el contenido Audio si lo tiene.
 - public Component **getControlPanelComponent()**;
 - Me devuelve un “Panel de Control” gráfico para controlar x aspectos del Player.
 - public void **start()**;
 - Pide que arranque la reproducción lo antes posible....(asíncrono)
- Estos dos métodos sirven para controlar desde este Player los Controller que se añadan. Ahora veremos que es un Controller.
 - public void **addController**(Controller newController)
 - public void **removeController**(Controller oldController);

javax.media.Player

Player implements javax.media.**MediaHandler**, javax.media.**Controller**, javax.media.**Duration**

- Por no dejar cabos sueltos. API de GainControl, bastante obvio y sencillo

```
public interface GainControl extends Control {  
    public void setMute(boolean mute);  
    public boolean getMute();  
    public float setDB(float gain);  
    public float getDB();  
    public float setLevel(float level);  
    public float getLevel();  
    public void addGainChangeListener(GainChangeListener listener);  
    public void removeGainChangeListener(GainChangeListener listener);  
}
```

- **OJO: extends Control.** Nos dan un shortcut para obtener un Control que maneja el Sonido del Player.
- Pero ¿ qué es un Control ?

Antes de seguir : javax.media.Control

- ¿ Qué es un Control ?

```
public interface Control {  
    public java.awt.Component getControlComponent();  
}
```

- Nos ofrece **un posible componente gráfico** para gestionar los valores a los que se refiere, p.e. un slider para el volumen...
- **Y además de este método podrá ofrecer otras funcionalidades específicas a nivel de API no gráfico.**

javax.media.Player

Player implements javax.media.**MediaHandler**, javax.media.**Controller**, javax.media.**Duration**

- El **Player** implementa el Interfaz de **javax.media.Controller**, el cual para empezar vemos que es un Reloj: es decir, implementa la gestión de reproducción temporal especificada por Java Media: **javax.media.Clock**. Por ejemplo objetos que reproducen MPEG Movies!

```
public interface Controller extends javax.media.Clock, javax.media.Duration
```

- También **Controller** implementa **javax.media.Duration** el cual nos va a indicar que los objetos ofrecen una duración temporal (ya lo implementa por doble camino!!!)
- **Controller**, en definitiva, proporciona una gestión de recursos, estados y eventos y también permite obtener nuevos Controls (javax.media.Control) que pueden ser usados con el componente que está siendo gestionado por el Player.
- Es en Controller donde reside la mayor parte del API de Player.

javax.media.Player

Player implements javax.media.**MediaHandler**, javax.media.**Controller**, javax.media.**Duration**

- El Player implementa el Interfaz de **javax.media.MediaHandler** el cual lo único que ofrece (que no es poco), es la posibilidad de establecer el DataSource!
- Como ya vimos, este método será más bien de uso interno.
public void **setSource**(javax.media.protocol.DataSource source)

- Ahora que sabemos de quien hereda, vayamos desgranando de manera ordenada todo lo que ofrece javax.media.Player

javax.media.Player API

| Visual Components | |
|--|--|
| <code>public java.awt.Component getControlPanelComponent();</code> | <ul style="list-style-type: none">• Devuelve un Control Panel con las opciones necesarias para gestionar este Player!• No es habitual, y generalmente devuelve null. Ocurre lo mismo que con <code>getVisualComponent()</code> |
| <code>public java.awt.Component getVisualComponent();</code> | <ul style="list-style-type: none">• Devuelve el componente Visual del Player (si lo tiene y si no reproduce únicamente sonido) donde el contenido media “visual” es renderizado.• Normalmente no devolverá nada pues el video se suele presentar en la Capa de Video, aunque es posible obtener su referencia para presentarlo dentro de una jerarquía AWT (recordamos <code>HVideoComponent?</code>) Lo vemos más adelante |

javax.media.Player API

| Controls | |
|--|--|
| public javax.media. GainControl getGainControl(); | <ul style="list-style-type: none"> Devuelve el Control para manejar el Audio, si es que este Player está emitiendo Audio. |
| public javax.media.Control[] getControls(); [Controller] | <ul style="list-style-type: none"> Devuelve la lista de Control que permiten gestionar aspectos de este elemento: un slider para subir/bajar sonido... (si no hay, como suele ser muy habitual, array.length=0) |
| public javax.media.Control getControl(String forName); [Controller] | <ul style="list-style-type: none"> Devuelve un Control que soporte el interface de Control pasado. |

javax.media.Player API

| DataSource (Interface MediaHandler, de uso interno) | |
|--|--|
| public void setSource (javax.media.protocol.DataSource source) [MediaHandler] | <ul style="list-style-type: none">• Permite establecer el DataSource |

javax.media.Player API

| Métodos relacionados con el Estado | |
|--|---|
| public Time getStartLatency() [Controller] | <ul style="list-style-type: none"> Lo más que podría tardar en presentar el primer frame de contenido |
| public int getState() [Controller] | <ul style="list-style-type: none"> Estado en el que se encuentra: (en Controller) Unrealized, Realizing, Realized, Prefetching, Prefetched, Started |
| public int getTargetState(); [Controller] | <ul style="list-style-type: none"> Estado siguiente. Al que estamos yendo. |
| public void realize(); [Controller] | <ul style="list-style-type: none"> Lo vemos en detalle después. |
| public void prefetch(); [Controller] | <ul style="list-style-type: none"> Lo vemos en detalle después. |
| public void deallocate(); [Controller] | <ul style="list-style-type: none"> Lo vemos en detalle después. |
| public void close(); [Controller] | <ul style="list-style-type: none"> Lo vemos en detalle después. |
| public void stop(); [Clock] | <ul style="list-style-type: none"> Lo vemos en detalle después. |

javax.media.Player API

| Métodos relacionados con Notificación de Estados del Player | |
|--|---|
| public void addControllerListener (javax.media.ControllerListener listener); [Controller] | <ul style="list-style-type: none">• Recibe los eventos de cambios de estado y sucesos. Veremos el detalle después |
| public void removeControllerListener (javax.media.ControllerListener listener); [Controller] | <ul style="list-style-type: none">• De-suscripción |

javax.media.Player API

| Métodos relacionados con Controllers añadidos al Player | |
|---|--|
| public void addController (javax.media.Controller newController) | <ul style="list-style-type: none">• Añade un Controller al Player (Puede ser otro Player!!!!) |
| public void removeController (javax.media.Controller oldCont); | <ul style="list-style-type: none">• Elimina el Controller |

Usado generalmente para reproducir
sincronizadamente varios Players.

Problemática de la gestión del Tiempo: Clock, TimeBase y MediaTime

- TimeBase es la referencia de tiempo sobre la que funciona el Reloj.
- El MediaTime es la referencia de Tiempo respecto a la cual “se mueve” el contenido presentado, y se define respecto al TimeBase y a un **ratio**: si **1.0** se va al unísono con TimeBase, si 0.0 estamos parados, si 2.0, vamos al doble de velocidad que TimeBase.
- Conviene hacer caso omiso de estos valores cuando estamos reproduciendo contenido Broadcast pues no hay manera de estar seguros de a qué tiempo se refiere el que se da.
- En un contexto de Broadcasting no es posible establecer con claridad la duración de lo que se está reproduciendo de manera que el API relacionado con el Media Time que veremos a continuación es prácticamente inútil para este tipo de contenido. Como se ve en el capítulo DSMCCC Stream Events & NPT, es mediante NPT, Normal Play Time, como se resuelve esta problemática para contenido Broadcast.
- Tan solo cuando reproduzcamos audio-clips, para los cuales sabemos cual es el comienzo y cual el final podremos hacer un uso de él.

javax.media.Player API

| Gestión del Tiempo | |
|--|--|
| public long getMediaNanoseconds() ; [Clock] | <ul style="list-style-type: none"> Media Time actual en nanosegundos |
| public Time getDuration() [Duration] | <ul style="list-style-type: none"> Duración del clip teniendo en cuenta el ratio. Si no la puede determinar devuelve DURATION_UNKNOWN |
| public void setStopTime (javax.media.Time stopTime); [Clock] | <ul style="list-style-type: none"> Establecemos el momento en que queremos que se pare la reproducción. Para resetear: establecer el valor Clock.RESET. Una vez que ha arrancado sólo se puede establecer 1 vez. |
| public javax.media.Time getStopTime() ; [Clock] | <ul style="list-style-type: none"> Ultimo StopTime establecido |
| public void setMediaTime (javax.media.Time now); [Clock] | <ul style="list-style-type: none"> Establece el Media Time, es decir: "salta a este punto". Sólo si stopped |
| public javax.media.Time getMediaTime() ; [Clock] | <ul style="list-style-type: none"> Media Time Actual (basado en su TimeBase) |

javax.media.Player API

| Gestión del Tiempo | |
|---|--|
| public void syncStart (javax.media.Time at); [Clock] | <ul style="list-style-type: none"> Establece el momento en que se debe arrancar la reproducción con respecto al TimeBase. Sólo si está parado. |
| public javax.media.Time getSyncTime (); [Clock] | <ul style="list-style-type: none"> Devuelve Media Time actual, al igual que getTime() o bien, cuando se usa syncStart(...) lo que falta para que lleguemos al momento indicado en el anterior, a partir de cuyo instante vuelve a funcionar igual que getTime() |
| public void setTimeBase (javax.media.TimeBase master) [Clock] | <ul style="list-style-type: none"> Establece el TimeBase, aunque se establece uno por defecto. Solo si el Clock está parado |
| public javax.media.TimeBase getTimeBase (); [Clock] | <ul style="list-style-type: none"> Devuelve el TimeBase |

- Recordemos TimeBase

```
public interface TimeBase {
    public Time getTime();
    public long getNanoseconds();
}
```

Problemática de la gestión del Tiempo: Clock, TimeBase and Media Time

- Para clarificar el significado de las marcas de tiempo vemos a continuación lo que se obtiene cada 5" reproduciendo un clip de audio: **la duración es desconocida** y El TimeBase parece un reloj que arranca con el Xlet.

```
[2#1:1] getMediaNanoseconds in secs: 4
[2#1:1] getRate                1.0
[2#1:1] getDuration   in secs  9.223372036854776E9
[2#1:1] getMediaTime in secs  5.0230000640000005
[2#1:1] getStopTime  in secs  9.223372036854776E9
[2#1:1] getSyncTime  in secs  5.042999808
[2#1:1] getTimeBase  in secs  328
[2#1:1] -----
[2#1:1] getMediaNanoseconds in secs: 10
[2#1:1] getRate                1.0
[2#1:1] getDuration   in secs  9.223372036854776E9
[2#1:1] getMediaTime in secs  10.09299968
[2#1:1] getStopTime  in secs  9.223372036854776E9
[2#1:1] getSyncTime  in secs  10.113000448000001
[2#1:1] getTimeBase  in secs  334
[2#1:1] -----
[2#1:1] getMediaNanoseconds in secs: 15
[2#1:1] getRate                1.0
[2#1:1] getDuration   in secs  9.223372036854776E9
[2#1:1] getMediaTime in secs  15.162999808
[2#1:1] getStopTime  in secs  9.223372036854776E9
[2#1:1] getSyncTime  in secs  15.184000000000001
[2#1:1] getTimeBase  in secs  339
```

javax.media.Player API

| Gestión del Tiempo | |
|--|---|
| public float getRate (); [Clock] | <ul style="list-style-type: none"> • Devuelve el factor de escala temporal actual: define la relación entre el Clock MediaTime y el TimeBase • Un ejemplo: un ratio de 3 indica que el Media Time irá 3 veces más rápido que el TimeBase una vez el Clock empiece. De igual forma si es -2 irá dos veces más lento que el TimeBase. Todos han de soportar al menos el valor 1.0 y el 0.0(parado) y no están obligados a soportar otros ratios! |
| public float setRate (float factor); [Clock] | <ul style="list-style-type: none"> • Establece el factor de escala Media Time – BaseTime • Sólo si el reloj está parado!. |
| public void stop (); [Clock] | <ul style="list-style-type: none"> • Para el reloj y libera recursos (veremos los estados después) |
| public void start (); | <ul style="list-style-type: none"> • Solicita que la reproducción comience cuanto antes. (veremos después los estados)! |

Todo este API no es necesario conocerlo en detalle, gran parte del mismo no lo usaremos, sin embargo conviene tenerlo presente

javax.media.Player API. Estados

- Veíamos en el método `Controller.getState()` que los posibles estados son:
 - Unrealized, Realizing, Realized, Prefetching, Prefetched, Started, Closed(este no aparece en el método pero se trata)
- Sigamos la secuencia de situaciones:
 - Cuando un Player se crea se encuentra en **Unrealized**
 - Cuando se llama a **realize()**: el método es **asíncrono** y retorna inmediatamente quedando el Player es estado **Realizing**. Durante este periodo el Player “construye” elementos colaterales que necesita para poder reproducir su contenido.
 - Cuando ya ha terminado de “realizarse” 😊, pasa a estado **Realized** y lanza un evento de tipo **javax.media.RealizeCompleteEvent** que recibirá el listener del Player:
ControllerListener:

```
javax.media.ControllerListener{  
    public abstract void controllerUpdate(javax.media.ControllerEvent event);  
}
```

javax.media.Player API. Estados

- Seguimos con la secuencia de situaciones:
 - Llamando al método **prefetch()** el Player intentará prepararse al máximo para tardar lo mínimo en realizar la reproducción cuando se le solicite (**start()**). Este método es **asíncrono**, quedando el Player en estado **Prefetching** hasta que pase a **Prefetched**, momento en el cual lanzará un evento: **javax.media.PrefetchCompleteEvent**.
 - Cuando está en **Prefetched** básicamente ya tiene listo el material que va a ser presentado **y ha consumido recursos, incluyendo los caros**.
- *Esta gestión de eventos permite a las aplicaciones comportarse “amigablemente” con el usuario, pues saben cuando una presentación está disponible para comenzar (relojito...)*
 - Cuando se llama a **start()**, se le solicita al Player que comience a ejecutar la presentación lo antes posible, y pasa a estado **Started** una vez empieza. El método es **asíncrono** y notificará que ha empezado lanzando: **javax.media.StartEvent**.
 - Puede ocurrir que no hayamos llamado antes ni a **realice()** ni a **prefetch()** lo cual hará **start()** internamente, y se recibirán los eventos correspondientes a medida que sucedan.

javax.media.Player API. Estados

- Sigamos la secuencia de situaciones:
 - Llamando a **close()** en cualquier estado se para la presentación si aplica, se liberan todos los recursos, y pasamos a estado **Closed**. Se lanzará un evento **javax.media.ControllerClosedEvent**
 - Llamando a **stop()** estando en **Started** se lanza el evento **javax.media.StopEvent** y nos vamos a **Prefetched**. **Es síncrono**.
 - Llamando a **stop()** estando en **Prefetching** se lanza el evento **javax.media.StopEvent** y nos vamos a **Realized**. **Es síncrono**.
 - Las llamadas de **Deallocate()** nos sirven para liberar recursos cuando estamos en **Prefetched**, **Prefetching** o **Realizing**, y pasar a la situación anterior: **Realized** para los dos primeros o **Unrealized** para el último. **Es síncrono** y lanza el evento **javax.media.DeallocateEvent**.

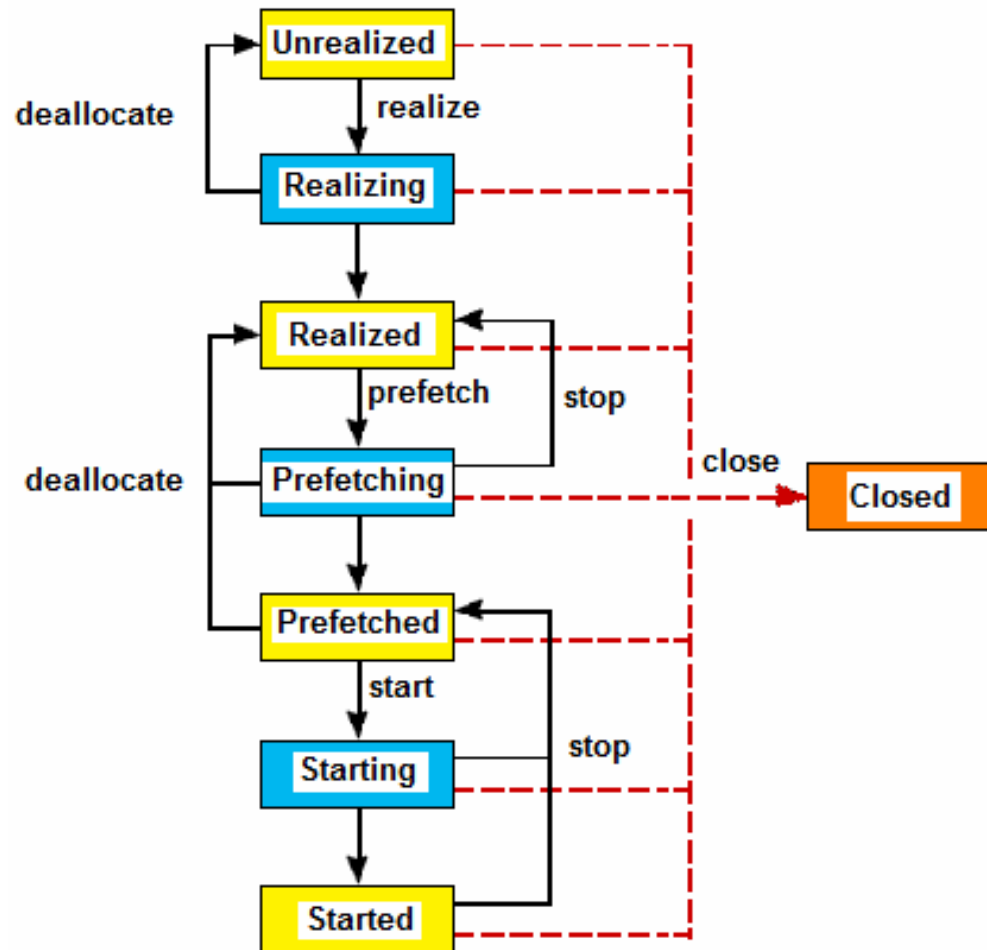
javax.media.Player API. Estados

- Los eventos que indican parada heredan en realidad de **javax.media.StopEvent** y pueden ser de los siguientes tipos:
 - **javax.media.StopByRequestEvent**: llamada a **stop()**
 - **javax.media.StopAtTimeEvent**: se ha llegado al **StopTime**
 - **javax.media.RestartingEvent**: Se le ha **cambiado** el **rate**, el **Media Time**, de manera que necesita re-arrancar para procesar los nuevos parámetros.
 - **javax.media.EndOfMediaEvent**: Se acabó el clip.
 - **javax.media.DataStarvedEvent**: Nos hemos parado porque el DataSource no ha proporcionado los datos lo suficientemente rápido.

javax.media.Player API. Estados

- Otros eventos importantes:
 - **org.davic.media.MediaPresentedEvent**: generado cuando realmente YA se está presentando al usuario el contenido. (evita equivocaciones en este sentido por temas de buffering por ejemplo...)
 - **org.davic.media.ResourceWithdrawnEvent** y **org.davic.media.ResourceReturnedEvent**: gestión de Recursos caros: Después lo vemos...
 - **org.dvb.media.PresentationChangedEvent**: generado cuando el contenido de lo que se está presentando ha cambiado...Única forma de enterarnos...Si queremos reaccionar podemos hacerlo.
 - **javax.media.DurationUpdateEvent**: Lanzado por un Controller cuando la duración varía.
- Existen más eventos referidos a aspectos más de detalle de JMF que no nos interesa ver aquí. Echad un ojo al API...(p.e. GainChangeEvent, ClockStartedError...)

javax.media.Player API. Estados



javax.media.Control

- Hemos hecho referencia a los javax.media.Control que puede ofrecer un Player, y que consisten, como decíamos, en posibles componentes gráficos que pueden servirnos para controlar algún elemento del contenido, p.e. el volumen de un clip de audio.
- **Los Control soportados por MHP se detallan en la tabla adjunta**, y son los que debemos usar, pues han sido creados para poder gestionar convenientemente el contenido Broadcast, cuyo comportamiento difiere mucho del estático.

org.davic.media.LanguageControl

org.davic.media.AudioLanguageControl

org.davic.media.SubtitlingLanguageControl

org.davic.media.FreezeControl

org.davic.media.MediaTimePositionControl

org.dvb.media.VideoPresentationControl

org.dvb.media.BackgroundVideoPresentationControl

org.dvb.media.VideoFormatControl

org.dvb.media.DVBMediaSelectControl

org.dvb.media.SubtitlingEventControl

org.dvb.media.ComponentBasedPlayerControl

javax.tv.media.AWTVideoSizeControl

javax.tv.media.MediaSelectControl

javax.media.Control

Recordemos.

- ¿ Qué es un Control ?

```
public interface Control {  
    public java.awt.Component getControlComponent();  
}
```

- Nos ofrece **un posible componente gráfico** para gestionar los valores a los que se refiere, p.e. un slider para el volumen...
- **Y además de este método podrá ofrecer otras funcionalidades específicas a nivel de API no gráfico.**

Ejercicios Bloque JMF-2

javax.tv.media.MediaSelectControl

- Permite añadir/quitar/cambiar los Streams que se están presentando de un Servicio, para por ejemplo, cambiar el idioma o el vídeo por otro realizado desde otra cámara en otro ángulo.
- Sería posible cambiar TODOS los Streams si se quiere.
- Estas operaciones se hacen sin parar el Player! La única limitación es que

solo se pueden tomar Streams del mismo Service

- En la práctica es un componente complicado de usar. Es más sencillo, por ejemplo, transmitir varios Services cada uno con un ángulo distinto de cámara que intentar cambiar el stream de vídeo dentro de un mismo service
- EL API:
 - public void **add**(Locator component)
 - Añade un ServiceComponent, como por ejemplo Subtítulos.
 - **No se permiten componentes que requieran de una re-sincronización.**
 - La llamada es asíncrona. Se recibirá un javax.tv.media.MediaSelectEvent

javax.tv.media.MediaSelectControl

- EL API:

- public void **addMediaSelectListener**(MediaSelectListener listener)
- public void **removeMediaSelectListener**(MediaSelectListener listener)

- Listener del MediaSelectControl.

```
public interface MediaSelectListener extends java.util.EventListener {  
    public void selectionComplete(MediaSelectEvent event);  
}
```

- Los eventos que se pueden recibir son:

MediaSelectFailedEvent: operación de select ha fallado.

MediaSelectCARefusedEvent extends MediaSelectFailedEvent : fallo por problema de CA

MediaSelectSucceededEvent: operación de select OK

- public Locator[] **getCurrentSelection**();
 - Selección actual

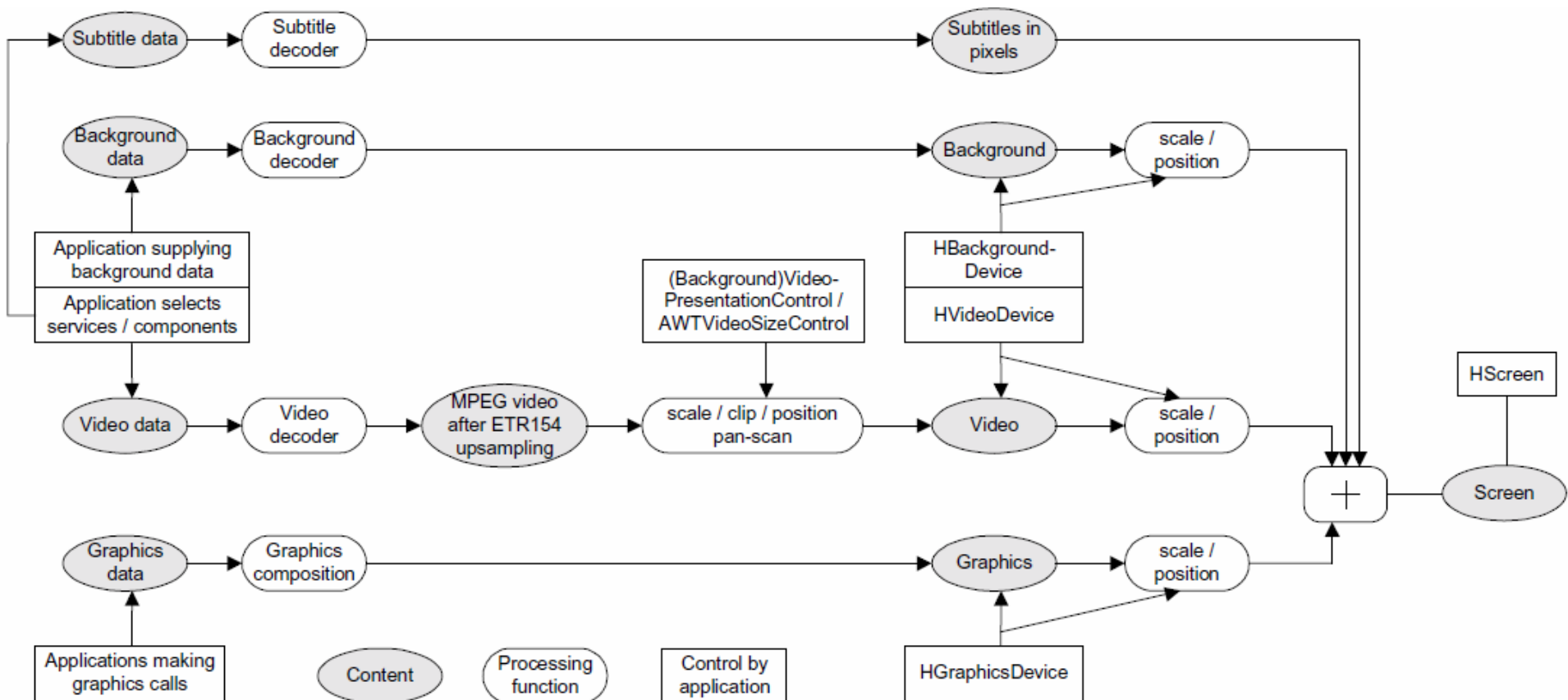
javax.tv.media.MediaSelectControl

- public void **remove**(Locator component)
 - Elimina un ServiceComponent, como por ejemplo Subtítulos.
 - No se permite eliminar componentes que requieran de una re-sincronización.
 - La llamada es asíncrona. Se recibirá un javax.tv.media.MediaSelectEvent
- public void **replace**(Locator fromComponent, Locator toComponent)
 - Cambia un ServiceComponent por otro.
 - No se permite eliminar componentes que requieran de una re-sincronización.
 - La llamada es asíncrona. Se recibirá un javax.tv.media.MediaSelectEvent
- public void **select**(Locator component)
 - Establece el Locator del Service Component a reproducir.
 - **OJO: sólo reproducirá este.**
- public void **select**(Locator[] components)
 - Establece una nueva selección en su totalidad de elementos a presentar en el Player.
 - La llamada es asíncrona. Se recibirá un javax.tv.media.MediaSelectEvent

Ejercicios Bloque JMF-3

javax.tv.media.AWTVideoSizeControl / org.dvb.media.VideoPresentationControl

- Un detalle respecto a **donde se van a aplicar estos “Control”**. El full-screen es la resolución requerida de up-sampling (MHP 1.1.2 A0068r1)



javax.tv.media.AWTVideoSizeControl

- Permite definir la posición, el tamaño y el área de clipping **del video que se presenta en el VideoDevice**. NO se refiere al que se presenta en un componente AWT. (No tiene sentido que se llame AWT...pero en fin).
- El escalado y posicionamiento tiene lugar usando coordenadas de pixel de Screen (no Normalizadas), esto es, la resolución en pixels de full-screen. Para saber la resolución actual sabemos que (A068r1):

The java.awt.Toolkit.getScreenSize method shall be equivalent to the pixel resolution of the current configuration of the default screen device returned by HScreen.getDefaultHGraphicsDevice.
- Este Control lo ofrece un Player que está presentando el Video en el VideoDevice, ya sabemos como acceder a este Player:
 - > **Service Context > ServiceMediaHandler > getControls** (o getControl(interface))
- Una vez que tenemos el Control la forma de trabajar consiste en indicarle lo que queremos mediante la clase: **javax.tv.media.AWTVideoSize**

javax.tv.media.AWTVideoSizeControl. javax.tv.media.AWTVideoSize

- Un AWTVideoSize representa una transformación de video: **un rectángulo (en coordenadas Screen) del vídeo fuente** será trasladado y escalado **para ajustarse a un rectángulo destino** definido en el mismo esquema de coordenadas pixel de Screen.

```
public class AWTVideoSize {  
    public AWTVideoSize(Rectangle source, Rectangle dest)  
    public Rectangle getSource()  
    public Rectangle getDestination()  
    public float getXScale() (destino.w/origen.w)  
    public float getYScale() (destino.h/origen.h)  
}
```

- Una vez que hemos definido nuestro AWTVideoSize hemos de comprobar que es posible aplicarlo, lo cual se hace mediante el método de AWTVideoSizeControl:
 - public AWTVideoSize **checkSize**(AWTVideoSize sz);
 - Nos ofrece el AWTVideoSize más próximo al que queremos.

javax.tv.media.AWTVideoSizeControl. javax.tv.media.AWTVideoSize

- Interesante: si posicionamos el video en el **Component.getLocationOnScreen()** de nuestra HScene haremos que la esquina superior izquierda del video coincida con la de HScene.
- Como decíamos, la resolución de la full-screen se obtiene de `java.awt.Toolkit.getScreenSize()`, y las coordenadas provistas por `Component.getLocationOnScreen()` están en ese esquema de coordenadas.

javax.tv.media.AWTVideoSizeControl

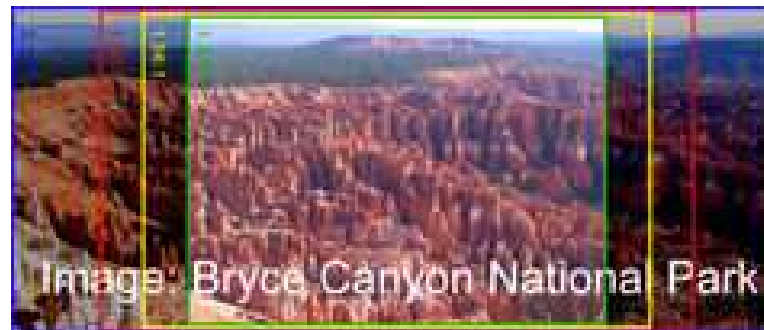
- El API
 - public AWTVideoSize **getSize()**;
 - Dimensiones en las que el Player **está trabajando**. Ideal para conocer las coordenadas originales de la representación de Video.
 - public AWTVideoSize **getDefaultSize()**;
 - Valor por defecto en el que el Video se representaría de no ser manipulado. **Ideal igualmente para volver a la configuración original.**
 - public Dimension **getSourceVideoSize()**;
 - Tamaño del Video fuente en coordenadas Screen(pixel).
 - public boolean **setSize**(AWTVideoSize sz);
 - Establecemos las nuevas coordenadas: true OK, false no pudo establecerlas.
 - public AWTVideoSize **checkSize**(AWTVideoSize sz);
 - Nos ofrece el AWTVideoSize más próximo al que queremos.

Antes de seguir. Entendiendo la Llegada del Video

- Por un lado al STB llega una señal de video con una determinada resolución, la cual, mediante el procedimiento de up-sampling (ETR 154), la transforma para adaptarse a una resolución (y Aspect Ratio), lo que se llama full-screen, en la mayor parte de los casos: 720x576 (SVDT).
- Este contenido de Video puede verse afectado por transformaciones que impliquen que lo que finalmente se verá en el Screen no se corresponda con la totalidad del video que llega, por ejemplo: imaginemos que recibimos una señal con un formato 16:9 y 1024x576, la cual se adapta en el proceso de up-sampling a un Screen 720x576 y 14:9. En este momento TODO el video que llegó se está viendo en la Screen.
- Ahora bien, imaginemos que para poder visualizar el contenido correctamente, se decide realizar una transformación pan-scan al video después del proceso de up-sampling, de manera que en lugar de presentar **todo** el origen en el espacio definido por 14:9 y 720x576, vamos a estirarlo para que en el Screen se vea la zona central de la “peli”.

Inciso: Pan & Scan

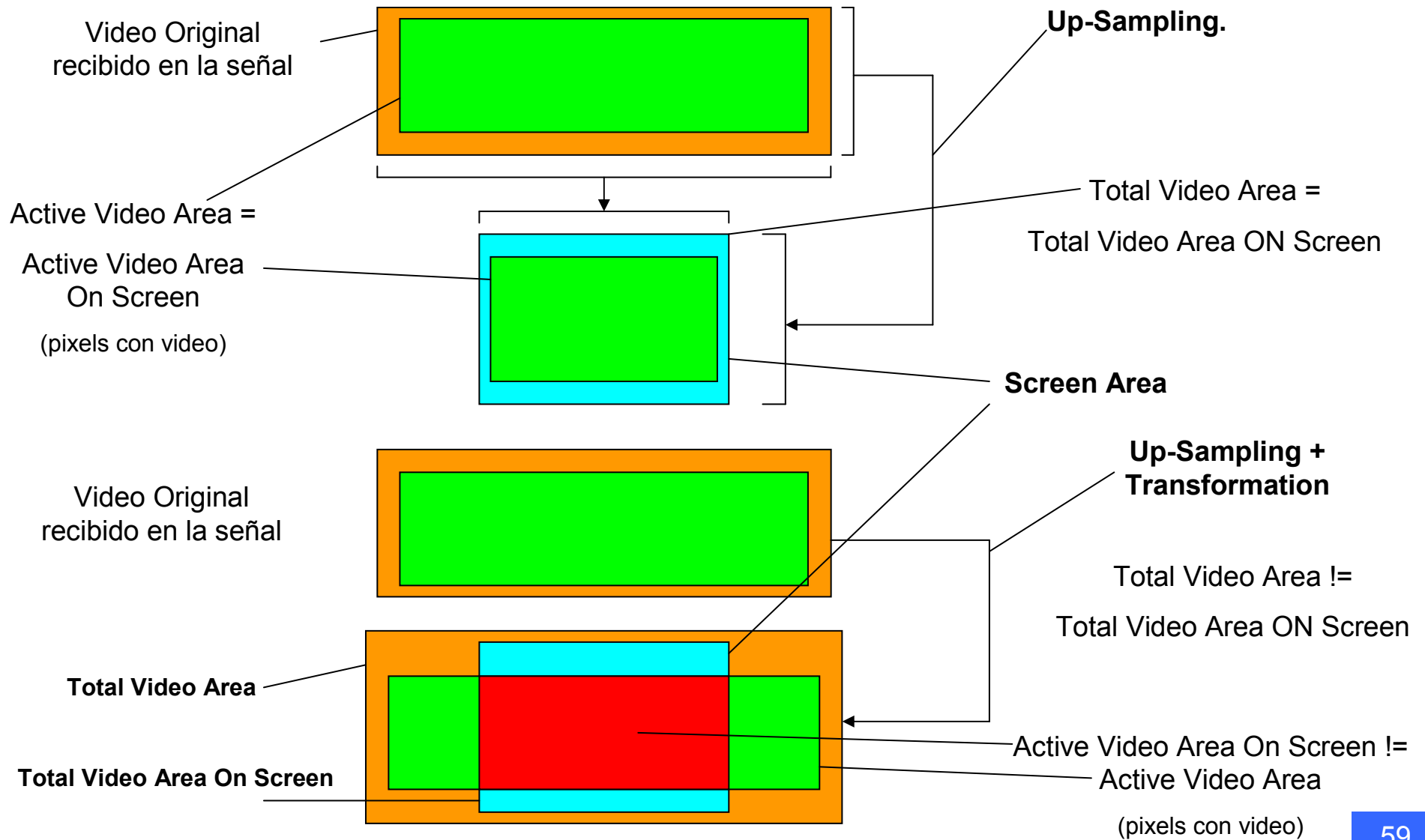
- Técnica para hacer que películas en WideScreen se vean en formatos 4:3
- El problema es que se pueden cortar zonas amplias de la visión.
- Como veremos en el Control VideoFormatControl, es posible a través de este obtener un VideoTransformation que represente pan & scan:
 - public VideoTransformation **getVideoTransformation**(int dfc);
- Cuando el componente obtenido ya está aplicando pan&scan VideoTransformation.isPanScan() será true, pero si se usa a posteriori setClipRegion(...) el método ofrecerá false.



<http://en.wikipedia.org/wiki/Image:PanScan7BridesPan.gif>

http://en.wikipedia.org/wiki/Image:Image_cropping_133x1.jpg

Antes de seguir. Entendiendo la Llegada del Video



org.dvb.media.VideoPresentationControl

- Se obtiene de un Player que está representando el Video y se usa para poder indicar **qué área del Video original** queremos representar en el **AWT Component**, si es que deseamos una zona concreta. **Para el posicionamiento y el escalado ya están las propiedades AWT de dicho componente.**
- Si NO establecemos un área de clip entonces el vídeo completo se mostrará en el componente.

org.dvb.media.VideoPresentationControl

- **Atención:** MHP 1.1.2 A068r1 **NO** Asumamos que el Component devuelto por el Player en el método `getVisualComponent()` es un **org.havi.ui.HVideoComponent !!!**

A.7.4.32 HVideoComponent

A.7.4.32.1 HAVi Specification

- In section 8.3.3.6 "Integrating HAVi Video Support into Platforms" the following two sentences shall be removed:
 - "The class HVideoComponent is intended to be returned by a platform specific controller for video. In platforms based on the Java Media Framework, the `Player.getVisualComponent` method shall return objects of this class."
- Note that the HVideoComponent class must be present, and MHP terminals may choose to use it or not. **Interoperable applications should not use HVideoComponent.**

org.dvb.media.VideoPresentationControl

- El API

- public java.awt.Dimension **getInputVideoSize()**;

- Tamaño del Video **justo después de up-sampling** del Video que llega en la señal **pero antes de cualquier transformación**.
- Ved el API. Veremos los resultados en una transformación

This method returns the dimensions of the video **before** any scaling has taken place (but **after** ETR154 up-sampling). **On 50Hz standard definition systems this method always returns 720x576.**

- public java.awt.Dimension **getVideoSize()**;

- Tamaño del Video que se presenta al Usuario. **Tiene en cuenta las operaciones de escalado y clipping**.
- Ved el API. Veremos los resultados en una transformación.

This method returns the size of the decoded video as it is being presented to the user. **It takes scaling and clipping into account.**

org.dvb.media.VideoPresentationControl

– public HScreenRectangle **getTotalVideoArea()**;

- Tamaño de Video total tras el up-sampling y las posibles transformaciones que se hayan aplicado después.
- Se ofrece en coordenadas normalizadas respecto a la Screen (pueden ser negativas si por ejemplo es mayor)
- Ved el API. Veremos los resultados en una transformación

This method returns a relative size and location of the total video area, including any "bars" used for letterboxing or pillarboxing that are included in the broadcast stream, but excluding any "bars" introduced as a result of video filtering. This may be larger or smaller than the size of the physical display device.

This method only describes the relationship between the total video and the screen.

It does not describe which portion of the screen is displaying the video.

Note: **This method includes any video scaling.**

– public HScreenRectangle **getTotalVideoAreaOnScreen()**;

- Zona del **TotalVideoArea** que se ve en el Screen.
- Se ofrece en coordenadas normalizadas respecto a la Screen.
- Ved el API. Veremos los resultados en una transformación

This method returns a relative size and location of the total video area on-screen, including any "bars" used for letterboxing or pillarboxing that are included in the broadcast stream, but excluding any "bars" introduced as a result of video filtering.

This method only describes the area on-screen where total video is being presented. This does not really describe which part of the video is being shown on-screen. This is especially true for pan&scan. Note: **This method includes any video scaling.**

org.dvb.media.VideoPresentationControl

– public HScreenRectangle **getActiveVideoArea()**;

- Area del TOTAL VIDEO en coordenadas HScreen en las que realmente se presentan pixels de Video.
- Ved el API. Veremos los resultados en una transformación

This method returns the size and location of the active video area. The active video area excludes any "bars" used for letterboxing or pillarboxing that the receiver knows about. Bars that are included in the broadcast stream and not signalled by active format descriptors are included in the active video area. The active video area may be larger/smaller than the screen, and may possibly be offset. The offsets will be negative if the origin of the active video area is above/left of the top, left corner of the screen. **In case of pan&scan, the value returned may vary over time. This method only describes the relationship between the active video and the screen. It does not describe which portion of the screen is displaying the video.** **Note:** This method **includes any video scaling.**

– public HScreenRectangle **getActiveVideoAreaOnScreen()**;

- Zona del ActiveVideoArea que se ve en el Screen.
- Se ofrece en coordenadas normalizadas respecto a la Screen.
- Ved el API. Veremos los resultados en una transformación

This method returns the size and location of the active video area on-screen. The active video area excludes any "bars" used for letterboxing or pillarboxing that the receiver knows about. Bars that are included in the broadcast stream and not signalled by active format descriptors are included in the active video area. The active video area on-screen may be smaller than the area of the screen, and may possibly be offset a positive amount. **This method only describes the area on-screen where active video is being presented.** It does not really describe which part of the video is being shown on-screen. This is especially true for pan&scan. **Note:** This method **includes any video scaling.**

org.dvb.media.VideoPresentationControl

- public boolean **supportsClipping()**;
 - Si el STB soporta clipping
- public java.awt.Rectangle **setClipRegion(java.awt.Rectangle clipRect)**;
 - Establecemos el área del vídeo que será presentada en el AWT Video Component: en el espacio de coordenadas pixel después de ETR154 up-sampling.
 - Si no se soporta clipping no se hace nada.
 - Devuelve lo que ha podido obtener más aproximado a lo solicitado.
- public java.awt.Rectangle **getClipRegion()**;
 - Devuelve el área clipped o, si no se soporta clipping serán las dimensiones del Video.

org.dvb.media.VideoPresentationControl

- public float[] **supportsArbitraryHorizontalScaling()**;
 - Devuelve dos float: valores mínimo y máximo de escala horizontal, p.e. [0.2, 3]
 - Null si no se soporta escalado horizontal ARBITRARIO.
- public float[] **supportsArbitraryVerticalScaling()**;
 - Idem al anterior respecto al vertical
- public float[] **getHorizontalScalingFactors()**;
 - En el caso de que **ArbitraryHorizontalScaling NO se soporte, devolverá los** valores de escalado soportados de forma ordenada. NULL si no hay soporte.
- public float[] **getVerticalScalingFactors()**;
 - Idem al anterior para Vertical

org.dvb.media.VideoPresentationControl

– public byte **getPositioningCapability()**;

Nos ayuda indicándonos que posibilidades de posicionamiento de Video se ofrecen:

- POS_CAP_OTHER: No definido.
- POS_CAP_FULL: Donde queramos. Aunque se salga de la zona visible.
- POS_CAP_FULL_IF_ENTIRE_VIDEO_ON_SCREEN: Donde queramos en la pantalla siempre que se vea todo en la pantalla
- POS_CAP_FULL_EVEN_LINES: Donde queramos. Aunque se salga de la zona visible, pero siempre que, en caso de video entrelazado este se posicione en líneas impares (empezando en 0).
- POS_CAP_FULL_EVEN_LINES_IF_ENTIRE_VIDEO_ON_SCREEN: Donde queramos en la pantalla siempre que se vea todo en la pantalla, pero siempre que, en caso de video entrelazado este se posicione en líneas impares (empezando en 0).

org.dvb.media.VideoPresentationControl

- Pero ¿ cómo se hace para presentar el Video en el Componente AWT ?

Video y AWT

- Todos los Players se construyen como **Background Players**, que representan su contenido en el Video Layer.
- Para obtener un AWT Component de manera que podamos “jugar” con el Video como si de un java.awt.Component normal se tratase hemos de llamar a **Player.getVisualComponent()**, y es en este momento cuando cambia el modo de visionado.
- Si el Player se puede convertir de **Background a AWT Component** entonces ofrecerá el componente, de otro modo este retornará null.
- **Component Players es opcional! No tienen porqué ofrecerse.**

Video y AWT

- Si un Player dispone de un Control del tipo: **org.dvb.media.ComponentBasedPlayerControl**, entonces el método `getVisualComponent()` habrá de ofrecernos un AWT Component en el que se represente el Player.
- Este Control sólo ofrece un método, que nos permite **volver a modo Background**:
 - public boolean **releaseVisualComponent()**: True si el Player ha cambiado de Component a Background mode

org.dvb.media.BackgroundVideoPresentationControl

- Hereda de **VideoPresentationControl**
- Se obtiene de un Player que está representando el Video en el Video Device.
- Al igual que AWTVideoSizeControl, sirve para establecer la posición, tamaño y escalado de una zona de video.
- En este caso se usarán **Coordenadas Normalizadas para establecer el punto origen donde presentar el video: HScreenPoint**. Para definir la zona que queremos cortar se siguen utilizando pixel respecto a full-screen. En cuanto al width/height destino se manejarán factores de escala.
- El esquema es similar al de AWTVideoSizeControl con respecto a AWTVideoSize: aquí usaremos un **VideoTransformation** para establecer los cambios que queremos efectuar. De igual forma disponemos de un método para obtener el VideoTransformation que más se acerque a lo que necesitamos.

org.dvb.media.BackgroundVideoPresentationControl

- El API
 - public boolean **setVideoTransformation**(VideoTransformation t);
 - True = OK, false = no pudo establecerlo
 - public **VideoTransformation** **getVideoTransformation**();
 - Devuelve el que se está aplicando
 - public VideoTransformation **getClosestMatch**(VideoTransformation t);
 - Devuelve el aplicable más parecido al solicitado
- Veamos a continuación org.dvb.media.VideoTransformation

org.dvb.media.BackgroundVideoPresentationControl

- org.dvb.media.VideoTransformation API
 - public **VideoTransformation**(java.awt.Rectangle clipRect, float horizontalScalingFactor, float verticalScalingFactor, HScreenPoint location)
 - Se establece el área en esquema de coordenadas de origen después de ETR154 up-sampling
 - public void **setClipRegion**(java.awt.Rectangle clipRect)
 - Zona a cortar en coordenadas Screen
 - public java.awt.Rectangle **getClipRegion**()
 - Zona a cortar establecida
 - public void **setScalingFactors**(float horizontalScalingFactor, float verticalScalingFactor)
 - public float[] **getScalingFactors**() [horz,ver]
 - public void **setVideoPosition**(HScreenPoint location)

org.dvb.media.BackgroundVideoPresentationControl

- org.dvb.media.VideoTransformation API
 - public HScreenPoint **getVideoPosition()**
 - public boolean **isPanAndScan()**
 - Si el Video representa pan and scan

org.dvb.media.VideoFormatControl

- Control que se obtiene de un Player que está representando MPEG-2 video streams.
- En realidad es un Control que ofrece información y sirve de apoyo para obtener objetos **VideoTransformation** que podemos usar con **BackgroundVideoPresentationControl**
- **Nos permite saber qué Aspect Ratio tiene aplicada la señal**, o qué formato es el original. Así mismo sirve para notificarnos mediante un listener de cambios en este sentido.
- Las posibilidades de transformación que se pueden aplicar son muy variadas: 4:3, 16:9, 14:9, o también letterboxed, pan and scan, pillar-boxed. Además la señal puede llegar en un formato, la TV soportar otro e incluso el STB tener sus propias preferencias.
- Para poder generar todas las posibilidades el STB usa una técnica denominada **Decoder Format Conversion: DFC**. El STB ajusta el tamaño del vídeo que llega en la señal, imaginemos tipo 4:3, para ser mostrado en formato 16:9, tal y como el usuario prefiere, siempre con las limitaciones del STB.

org.dvb.media.VideoFormatControl

- Los cambios de DFC sobre la señal pueden aplicarse de forma automática debido a preferencias establecidas por el usuario en el STB (**TV Behavior Control**) y también porque una aplicación ha efectuado cambios a la señal (**Application Behavior Control**), y cuando esto sucede estos cambios tienen prioridad sobre los anteriores, de forma que sólo los establecidos por la aplicación tendrán lugar sobre el vídeo original.
- Esto es importante saberlo pues conocemos el punto de partida del vídeo.

org.dvb.media.VideoFormatControl

- Usando el método siguiente obtenemos un objeto **VideoTransformation** que aplicado al **BackgroundVideoPresentationControl** genera una salida de video que cumple con el DFC pasado.
 - VideoTransformation **getVideoTransformation**(int dfc);
- API
 - public int **getAspectRatio**(); (Ver ASPECT_RATIO_*)
 - Aspect Ratio del Video tal y como se transmite.
 - public int **getDisplayAspectRatio**(); (Ver DAR_*)
 - Aspect Ratio del Display Device conectado al STB.
 - En realidad da el Aspect Ratio del STB.
 - public int **getActiveFormatDefinition**(); (Ver AFD_*)
 - Nos permite saber con qué formato llega la señal de origen si esta info está en el signalling
 - public int **getDecoderFormatConversion**(); (Ver DFC_*)
 - Nos permite saber qué DFC se está aplicando a la señal.

org.dvb.media.VideoFormatControl

- public VideoTransformation **getVideoTransformation**(int dfc); (Ver DFC_*)
 - Ofrece un VideoTransformation con los parámetros ajustados al tipo de transformación indicado
- public boolean **isPlatform**();
 - Indica si el control sobre los DFC aplicados está siendo gestionado por la plataforma: tipo DFC_PLATFORM.
- public void **addVideoFormatListener**(VideoFormatListener l);
- public void **removeVideoFormatListener**(VideoFormatListener l);
 - Mediante un Listener nos permite conocer cambios en el formato.

```
public interface VideoFormatListener extends java.util.EventListener {  
    public void receiveVideoFormatEvent(VideoFormatEvent anEvent);  
}
```

org.dvb.media.VideoFormatControl (ved API)

| | |
|----------------------|----------------------------------|
| ASPECT_RATIO_UNKNOWN | AFD_16_9_SP_4_3 |
| ASPECT_RATIO_4_3 | DFC_PROCESSING_UNKNOWN |
| ASPECT_RATIO_16_9 | DFC_PROCESSING_NONE |
| ASPECT_RATIO_2_21_1 | DFC_PROCESSING_FULL |
| AFD_NOT_PRESENT | DFC_PROCESSING_LB_16_9 |
| AFD_16_9_TOP | DFC_PROCESSING_LB_14_9 |
| AFD_14_9_TOP | DFC_PROCESSING_CCO = 4 |
| AFD_GT_16_9 | DFC_PROCESSING_PAN_SCAN |
| AFD_SAME | DFC_PROCESSING_LB_2_21_1_ON_4_3 |
| AFD_4_3 | DFC_PROCESSING_LB_2_21_1_ON_16_9 |
| AFD_16_9 | DFC_PLATFORM |
| AFD_14_9 | DFC_PROCESSING_16_9_ZOOM |
| AFD_4_3_SP_14_9 | DAR_4_3 |
| AFD_16_9_SP_14_9 | DAR_16_9 |

org.davic.media.LanguageControl

- Obtenido de un Player que reproduce Audio (Películas) o subtítulos.
- Permite saber los idiomas disponibles y establecerlos.
 - `public String[] listAvailableLanguages();`
 - 3 letras. Language ISO 639
 - `public void selectLanguage(String lang)`
 - `public String getCurrentLanguage();`
 - `public String selectDefaultLanguage();`
 - Establece el idioma por defecto del deco

org.davic.media.AudioLanguageControl es idéntico a **LanguageControl**:

```
public interface AudioLanguageControl extends LanguageControl {  
}
```

org.davic.media.SubtitlingLanguageControl

- Hereda de **LanguageControl** y añade lo siguiente para activar/desactivar los subtítulos
 - public boolean **isSubtitlingOn()**;
 - public boolean **setSubtitling**(boolean new_value);

org.davic.media.FreezeControl

- Permite “congelar” la reproducción de un contenido. En el caso del vídeo deja visible el último frame

```
public interface FreezeControl extends javax.media.Control {  
    public void freeze() throws MediaFreezeException;  
    public void resume() throws MediaFreezeException;  
}
```

Otros datos. Control

- **org.davic.media.MediaTimePositionControl** sólo estará en contenido **NO Broadcast**

La intención es cambiar el punto de reproducción: saltar.

```
public interface MediaTimePositionControl extends javax.media.Control{  
    public javax.media.Time setMediaTimePosition(javax.media.Time mediaTime);  
    public javax.media.Time getMediaTimePosition();  
}
```

- VideoDrips Players sólo soportarán los siguientes. Nos permiten reproducir un VideoDrip en una zona del Video Device
 - BackgroundVideoPresentationControl
 - AWTVideoSizeControl

Players para controlar a Players

- Como ya mencionamos un Player puede usarse para controlar otros Players pues Player hereda de Controller.
- Este hecho permite que un Player pueda sincronizar la presentación de contenido de varios Players, recordemos que un Player es un Clock. Esta situación se puede dar si por ejemplo tenemos un Player para presentar video y otro para hacerlo con audio.
- **OJO:** sólo funciona para sincronizar Players no para obtener Control comunes a todos
- No es que sea frecuente su uso.

Video Drips

- ¿ Recordamos el Locator de un VideoDrip ? **dripfeed://**
- Un DripFeed se carga en un Player con un DataSource especial: **org.dvb.media.DripFeedDataSource**, el cual cuando se construye con el Locator **dripfeed://** simplemente sabe que el tipo de contenido es un VideoDrip, esto es, un I-Frame seguido de varios P-Frames.
- La ventaja de usar VideoDrips es que los frames sólo necesitan almacenar las diferencias respecto al anterior, lo cual ahorra mucho espacio.
- ¿ y cómo se le pasa el contenido ? Mediante el método del DataSource: **feed(byte[] data)**. **Es decir, que debes de disponer de un DripFeed en memoria para reproducirlo...el cual ha llegado a memoria como un file, o un acceso con una URL...**
- Referencia interesante

<http://forum.java.sun.com/thread.jspa?threadID=415548&messageID=1903429>

Player del Service actual

- Ya vimos cuando estudiamos el Service Context que aparecía un tipo de ServiceContentHandler denominado **ServiceMediaHandler**, pues bien, este es el Player encargado de presentar un DVBSservice, presentándose el stream de vídeo en la capa de Video por defecto.

Cambiando de Canal. ¿ Cómo afectan los cambios de Service a los Players ?

- Para empezar los cambios realizados mediante los Control se resetean.
- MHP indica que cuando esto sucede se debe de **re-acceder a los Players** (y por tanto a los Control) en uso. Cuando se cambia de canal lo suyo es liberar todos los recursos.
- **La forma de saber cuando se ha cambiado de canal** es mediante el listener al **ServiceContext** de forma que controlemos la llegada de eventos y podamos re-acceder a los Players:
 - **PresentationChangedEvent**: Indica que o bien hay un nuevo Service presentándose o que el que había ha dejado de hacerlo. El estado del ServiceContext es **Presenting**

Cambiando de Canal

- Flickering puede ser un problema....Pero no hay forma de reducirlo salvo desde el middle, retrasando por ejemplo el momento de hacer reset, de manera que las aplicaciones puedan establecer sus parámetros (no confiemos demasiado)...

Gestión de Recursos Caros en JMF

- Normalmente los Players usarán decoders MPEG Hardware, lo cual es un recurso muy caro. Incluso aun cuando estos estén implementados en SW, es tal la carga de CPU que sigue siendo igual de costoso.
- El mecanismo adoptado es sencillo: **cuando alguien solicita un Player se le da.**
 - Last-in/First-served
 - Cualquier otro Player que estuviera usando el recurso notificará mediante un evento (`javax.media.ControllerListener`) `org.davic.media.ResourceWithdrawnEvent` a aquel listener del Player, pero el Player se queda en situación “medio estable”
- Cuando el recurso se recupera se recibe un evento del tipo: **`org.davic.media.ResourceReturnedEvent`** (los Players que estuvieran en Started proseguirían con la reproducción)
- Cuando el middle destruye el Player, entonces se recibirá un **`org.davic.media.ResourceUnavailableEvent`** lo cual indica al usuario que deberá crear un nuevo Player. No cambia el estado, has de hacerlo tú.

Tuning en JMF

- A JMF no le está permitido usar el Tuner para sintonizar otro TS.
- JMF no puede decodificar nada que no esté en el TS actual
- Para cambiar se debe usar o bien JavaTV ServiceContext API (y lo hace de forma transparente al seleccionar un Service que está en otro TS) o Tuning API (explícito)

Playing Audio a partir de fuentes

- JMF sólo puede reproducir Audio a partir de algo que se pueda referir como un File:/ (sí que funciona el reproducir un Service que sea una cadena de radio digital, es un DVB Service)
- Para ayudar disponemos de la clase org.havi.ui.HSound, que permite reproducir MPEG-1 sound (o quizá algo mas...) pero de fuentes como memoria, ficheros, URLs...
- **OJO:** Los métodos son síncronos.

```
public class HSound {  
    public void load(String location)  
    public void load(java.net.URL contents)  
    public void set(byte data[])  
    public void play()  
    public void stop()  
    public void loop()  
    public void dispose()  
}
```

Ejercicios Bloque JMF-4

| | |
|-------------------------|---|
| ISO/IEC 13818-1 | Part 1. Elementary Streams transport definition |
| ISO/IEC 13818-6 | Part 6. Extensions for DSM-CC. Digital Storage Media Command and Control |
| ETSI EN 300 468 | Digital Video Broadcasting (DVB);Specification for Service Information (SI) in DVB systems |
| ETSI EN 301 192 | DVB specification for data broadcasting |
| ETSI TR 101 202 | Implementation Guidelines for Data broadcasting |
| ETSI TR 101 162 | Digital broadcasting systems for television, sound and data services; Allocation of Service Information (SI) codes for Digital Video Broadcasting (DVB) systems |
| ETSI TR 102 154 | Implementation guidelines for the use of MPEG-2 Systems, Video and Audio in Contribution and Primary Dist |
| ETSI TR 101 211 | Guidelines on implementation and usage of Service Information (SI) |
| ETSI TR 101 200 | Digital Video Broadcasting (DVB); A guideline for the use of DVB specifications and standards |
| DAVIC | Digital Audio Visual Council. davic 1.4.1 |
| HAVI | Specification of the Home Audio/Video Interoperability (HAVi) Architecture |
| Interactivetvweb | http://www.interactivetvweb.org/ |
| Wikipedia DSMCC | http://en.wikipedia.org/wiki/DSM-CC |
| MHP 1.1.2 | Multimedia Home Platform, A068r1 & tam668r23_11xdraft_20061115 |
| MHP 1.1.3 | Multimedia Home Platform, A068r3 |
| CDC 1.1 | Connected Device Configuration (CDC) 1.1 (JSR=218). |
| PBP 1.1 | Personal Basis Profile 1.1 (JSR 217) |
| MHP.org | www.mhp.org |
| INTRO MHP 1.1.3 | tam1032r1-mhp-iptv-presentation |