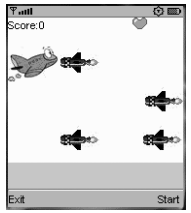
Wireless Online Game Development for Mobile Device

Lesson 6

What have we learnt last week?

- Introduction to Thread
- Game Key Detection
- Develop a sample action game / shooting game



Introduction

- Many multimedia types and formats exist in today's market, and new types and formats are being introduced all the time.
- There are also many, diverse methods to store and deliver these various media types. For example, there are traditional storage devices (such as CDs, and DVDs), wired protocols (HTTP, etc.) and wireless protocols (WAP, etc.).

Mobile Media API

- The Mobile Media API (MMAPI) is an optional package that supports multimedia applications on J2ME-enabled devices.
- It has been designed to run with any protocol and format; for example, it does not specify that the implementation must support particular transport protocols such as HTTP or Real-Time Transport Protocol (RTP), or media formats such as MP3, MIDI, or MPEG-4.

Features of MMAPI

- Support for Tone Generation, Playback and Recording of Time Based Media
  - Supports any time-based audio and video content by offering tools to control the flow of the media stream.
- Small Footprint
  - CLDC as the target configuration sets strict limits for memory consumption. Design of the API emphasizes that as much as possible.
- Protocol and Content Agnostic
  - The design of the API is not biased towards any specific protocol or content type.

Features of MMAPI

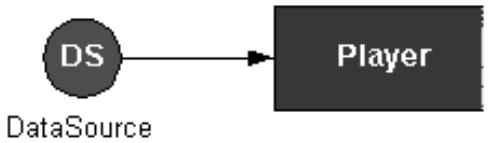
- Sub-settable: Audio-only vs. General Multimedia
  - It is possible to separate a subset of the API in order to provide support for only some type of content (e.g. for basic audio). It allows profiles that cannot support all the features of the full API to only take the parts that are needed.
- Extensible
  - The API is designed in a way that allows new features to be added later without breaking the old functionality.
- Optionally for Implementation
  - The API offers a wide range of features for different purposes.

Multimedia Processing

- Multimedia processing can be broken into two parts:
  - Protocol Handling
    - Reading data from a source (such as a file, capture device, or streaming server) into a media processing system.
  - Content Handling
    - Processing the media data (parsing or decoding, etc…) and rendering the media to output devices such as an audio speaker or video display.

High-level Object Types

- To facilitate multimedia processing , MMAPI provides two high-level object types:
  - **DataSource** for protocol handling
  - **Player** for content handling



DataSource

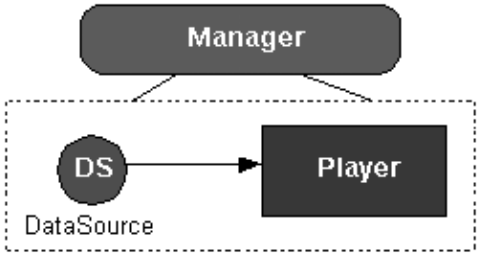
- A DataSource encapsulates protocol handling.
- It hides the details of how the data is read from its source, whether the data is coming from a file, streaming server, or proprietary delivery mechanism.
- DataSource provides a set of methods to allow a Player to read data from it for processing.

Player

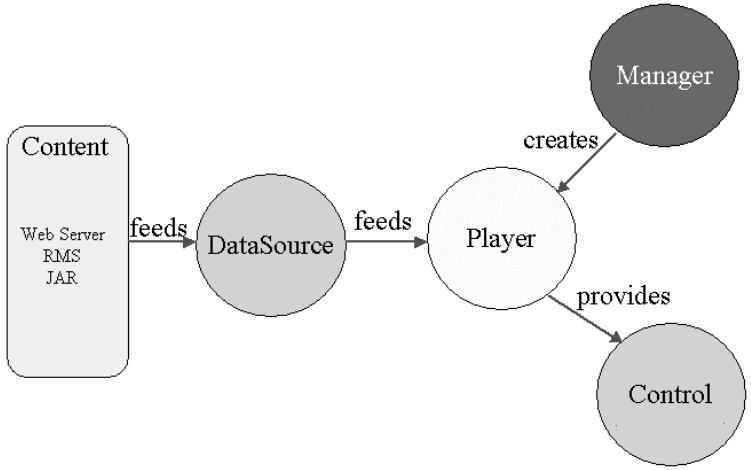
- A Player reads from the DataSource, processes the data, and renders the media to the output device.
- It provides a set of methods to control media playback and basic synchronization.
- Players also provide some type-specific controls to access features for specific media types.

Manager

- The Manager creates Players from DataSources.
- Manager also provides methods to create Players from locators and InputStreams



MMAPI Architecture



MMAPI Architecture (1)

- A player knows how to interpret media data.
- One type of player, for example, might know how to produce sound based on MP3 audio data. Another type of player might be capable of showing a QuickTime movie.
- Players are represented by implementations of the ***javax.microedition.media.Player*** interface.

MMAPI Architecture (2)

- You can use one or more controls to modify the behavior of a player.
- You can get the controls from a Player instance and use them while the player is rendering data from media.
  - For example, you can use a VolumeControl to modify the volume of a sampled audio Player.
- Controls are represented by implementations of the ***javax.microedition.media.Control*** interface; specific control subinterfaces are in the ***javax.microedition.media.Control*** package.

MMAPI Architecture (3)

- A data source knows how to get media data from its original location to a player.
- Media data can be stored in a variety of locations, from remote servers to resource files or RMS databases.
- Media data may be transported from its original location to the player using HTTP, a streaming protocol like RTP, or some other mechanism.
- ***javax.microedition.media.protocol.DataSource*** is the abstract parent class for all data sources in the MMAPI.

MMAPI Architecture (4)

- Finally, a manager ties everything together and serves as the entry point to the API.
- The ***javax.microedition.media.Manager*** class contains static methods for obtaining Players or DataSources.

The MMAPI Packages

- MMAPI comprises three packages:
  - **javax.microedition.media** provides some interfaces, an exception, and the Manager class, which is the access point for obtaining system-dependent resources such as Players for multimedia processing.
  - **javax.microedition.media.control** defines the specific control types that can be used with a Player: VolumeControl, VideoControl, and others.
  - **javax.microedition.media.protocol** defines the protocols for handling custom controls. For example, it includes the DataSource class, which is an abstraction for media-control handlers.

Five States for Player

- A Player has five states:
  - UNREALIZED
  - REALIZED
  - PREFETCHED
  - STARTED
  - CLOSED

Player State - UNREALIZED

- A Player starts in the UNREALIZED state. An unrealized Player does not have enough information to acquire all the resources it needs to function.
- The following methods must not be used when the Player is in the UNREALIZED state. Otherwise, an IllegalStateException will be thrown.
  - getContentType
  - setMediaTime
  - getControls
  - getControl

Player State - REALIZED

- A Player is in the REALIZED state when it has obtained the information required to acquire the media resources. Realizing a Player can be a resource and time consuming process. The Player may have to communicate with a server, read a file, or interact with a set of objects.
- Although a realized Player does not have to acquire any resources, it is likely to have acquired all of the resources it needs except those that imply exclusive use of a scarce system resource, such as an audio device.
- Normally, a Player moves from the UNREALIZED state to the REALIZED state. After realize has been invoked on a Player, the only way it can return to the UNREALIZED state is if deallocate is invoked before realize is completed. Once a Player reaches the REALIZED state, it never returns to the UNREALIZED state. It remains in one of four states: REALIZED, PREFETCHED, STARTED or CLOSED.

Player State - PREFETCHED

- Once realized, a Player may still need to perform a number of time-consuming tasks before it is ready to be started. For example, it may need to acquire scarce or exclusive resources, fill buffers with media data, or perform other start-up processing. Calling prefetch on the Player carries out these tasks.
- Once a Player is in the PREFETCHED state, it may be started. Pre-fetching reduces the startup latency of a Player to the minimum possible value.
- When a started Player stops, it returns to the PREFETCHED state.

Player State - STARTED

- Once prefetched, a Player can enter the STARTED state by calling the start method. A STARTED Player means the Player is running and processing data. A Player returns to the PREFETCHED state when it stops, because the stop method was invoked, or it has reached the end of the media.
- When the Player moves from the PREFETCHED to the STARTED state, it posts a STARTED event. When it moves from the STARTED state to the PREFETCHED state, it posts a STOPPED, END_OF_MEDIA event depending on the reason it stopped.
- The “setLoopCount” method must not be used when the Player is in the STARTED state, Otherwise, an IllegalStateException will be thrown.

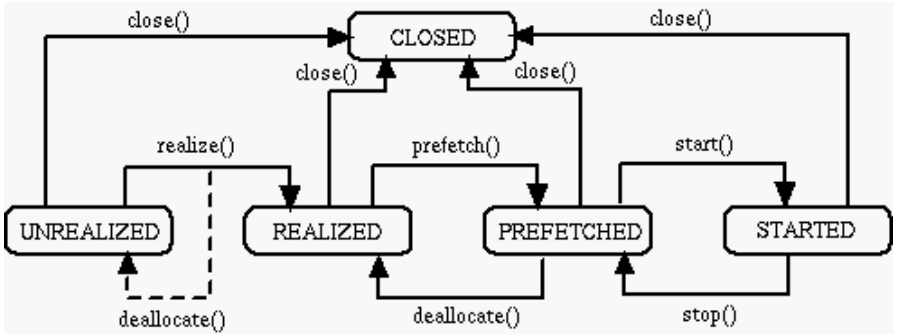
Player State - CLOSED

- Calling close on the Player puts it in the CLOSED state.
- In the CLOSED state, the Player has released most of its resources and must not be used again.

Player Life Cycle Diagram

- A Player implementation must allow successful state transition to each of these states using the six state-transition methods:
  - realize( )
  - prefetch( )
  - start( )
  - stop( )
  - deallocate( )
  - close( )

Player Life Cycle Diagram



State Transaction Flow

- When a player is created, it is in the UNREALIZED state.
- Calling realize() moves it to the REALIZED state and initializes the information the player needs to acquire media resources.
- Calling prefetch() moves it to PREFETCHED, establishes network connections for streaming data, and performs other initialization tasks.
- Calling start() causes a transition to the STARTED state, where the player can process data.
- When it finishes processing (reaches the end of a media stream), it returns to the PREFETCHED state.
- Calling close() moves the player to the CLOSED state.

State Transaction Flow

- A Player provides controls specific to the particular types of media it processes.
- The application uses getControl() to obtain a single control, or getControls() to get an array of them. As an example, if a player for MIDI media invokes getControl() it gets back a MIDIControl.

Limitations on Using MMAPI in the J2ME Wireless Toolkit

- An application can create multiple players, but can realize only **one MIDI player** or **one tone-sequence** player. Any attempt to realize a second player of either type will throw a ***MediaException***.
- The toolkit allows any number of simultaneous playTone() calls for a single note, but guarantees only that at least four will actually work. The playTone() method can be called while a MIDI or tone-sequence player is playing.
- The number of video and WAV audio players are limited by the size of the heap, as specified from "J2ME Wireless Toolkit" by choosing Edit → Preferences → Storage.
- Only one audio capture can be in use at one time.

Content Types

- Wave audio files: audio/x-wav
- AU audio files: audio/basic
- MP3 audio files: audio/mpeg
- MIDI files: audio/midi
- Tone sequences: audio/x-tone-seq
- MPEG video files: video/mpeg

Feature sets for five different types of media

Feature Set	Implementation Requirements
Sampled Audio	• *Should* implement `VolumeControl`, `StopTimeControl`.
MIDI	• *Should* implement `VolumeControl`, `MIDIControl`, `TempoControl`, `PitchControl`, `StopTimeControl`.
Tone Sequence (`Player` for `TONE_DEVICE_LOCATOR`)	• *Must* implement `ToneControl`. • *Should* implement `VolumeControl`, `StopTimeControl`.
Interactive MIDI (`Player` for `MIDI_DEVICE_LOCATOR`)	• *Must* implement `MIDIControl`.
Video	• *Must* implement `VideoControl`. • *Should* implement `FramePositioningControl`, `StopTimeControl`, `VolumeControl` (if audio is also available).

Coding Examples

- Single-Tone Generation
- Simple Media Playback with Looping
- Fine-Grained Playback Control
- MIDI Playback with Some Fine-Grained Control
- Video Playback
- Playing Back from Media Stored in RMS
- Playing Back from Media Stored in JAR
- Synchronization of Different Players
- Tone Sequence Generation
- Capture and Recording
- Camera

Single-Tone Generation

```
try {
  // Play a C4 tone for 1 seconds at max volume
  Manager.playTone (ToneControl.C4, 5000, 100);
} catch (MediaException e) {
}
```

Simple Media Playback with Looping

```
try {
    Player p = Manager.createPlayer
    ("http://webserver/music.mp3");
    p.setLoopCount(5);
    p.start();
} catch (IOException ioe) {
} catch (MediaException me) {
}
```


Fine-Grained Playback Control

```
static final long SECS_TO_MICROSECS = 1000000L;
Player p;
VolumeControl vc;

try {
  p = Manager.createPlayer("http://webserver/music.mp3");
  p.realize();
  p.addPlayerListener(new Listener());                    // Set a listener.
  vc = (VolumeControl)p.getControl("VolumeControl"); // Grab volume control for player
  if (vc != null)
    vc.setLevel(100);                                     // Set Volume to max.
  p.setMediaTime(5 * SECS_TO_MICROSECS);          // Set a start time
  p.prefetch();            // Guarantee that the player can start with the smallest latency
  p.start();               // Non-blocking start
} catch (IOException ioe) {
} catch (MediaException me) { }
```


Fine-Grained Playback Control

```
class Listener implements PlayerListener {
   public void playerUpdate(Player p, String event, Object eventData) {
      if (event == END_OF_MEDIA || event == STOP_AT_TIME) {
         System.out.println("Done processing");
         try {
            p.setMediaTime(5 * SECS_TO_MICROSECS);
            p.start();
         } catch (MediaException me) { }
         break;
      }
   }
}
```


MIDI Playback with Some Fine-Grained Control

```
Player p;
TempoControl tc;

try {
   p = Manager.createPlayer("http://webserver/tune.mid");
   p.realize();

  // Grab the tempo control
   tc = (TempoControl)p.getControl("TempoControl");
   tc.setTempo(120000);                                    // 120 beats/min
   p.start();

} catch (IOException ioe) {
} catch (MediaException me) { }
```

Video Playback

```
Player p;
VideoControl vc;

try {
   p = Manager.createPlayer("http://webserver/movie.mpg");
   p.realize();
   // Grab the video control and set it to the current display.
   vc = (VideoControl)p.getControl("VideoControl");
   if (vc != null) {
      Form form = new Form("video");
      form.append((Item)vc.initDisplayMode(vc.USE_GUI_PRIMITIVE, null));
      Display.getDisplay(midlet).setCurrent(form);
   }
   p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```


Playing Back from Media Stored in RMS

```
RecordStore rs;
int recordID;
  :  // code to set up the record store.

try {
   InputStream is = new
   ByteArrayInputStream(rs.getRecord(recordID));
   Player p = Manager.createPlayer(is, "audio/X-wav");
   p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```


Playing Back from Media Stored in JAR

```
// Notice that in MIDP 2.0, the wav format is mandatory only
// in the case that the device supports sampled audio.
try {
   InputStream is = getClass().getResourceAsStream
   ("audio.wav");
   Player p = Manager.createPlayer (is, "audio/X-wav");
   p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```


Synchronization of Different Players

```
Player p1, p2;
try {
   p1 = Manager.createPlayer("http://webserver/tune.mid");
   p1.realize();
   p2 = Manager.createPlayer("http://webserver/movie.mpg");
   p2.realize();
   p2.setTimeBase(p1.getTimeBase());
   p1.prefetch();
   p2.prefetch();
   p1.start();
   p2.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```

Tone Sequence Generation

```
byte tempo = 30;                    // set tempo to 120 bpm
byte d = 8;                         // eighth-note

byte C4 = ToneControl.C4;
byte D4 = (byte)(C4 + 2);           // a whole step
…
byte rest = ToneControl.SILENCE;    // rest

byte[] mySequence = {
   ToneControl.VERSION, 1,          // version 1
   ToneControl.TEMPO, tempo,        // set tempo
   ToneControl.BLOCK_START, 0,      // start define "A" section
   E4,d, D4,d, C4,d, E4,d,          // content of "A" section
   ToneControl.BLOCK_END, 0,        // end define "A" section
   ToneControl.PLAY_BLOCK, 0,       // play "A" section
};
```


Tone Sequence Generation

```
try{
   Player p = Manager.createPlayer
    (Manager.TONE_DEVICE_LOCATOR);
   p.realize();
   ToneControl c = (ToneControl)p.getControl("ToneControl");
   c.setSequence(mySequence);
   p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```


Capture and Recording

```
try {
   // Create a DataSource that captures live audio.
   Player p = Manager.createPlayer("capture://audio");
   p.realize();
   // Get the RecordControl, set the record location, and start the Player and record for 5s
   RecordControl rc = (RecordControl)p.getControl("RecordControl");
   rc.setRecordLocation("file:/tmp/audio.wav");
   rc.startRecord();
   p.start();
   Thread.currentThread().sleep(5000);
   p.stop();
   rc.stopRecord();
   rc.commit();
} catch (IOException ioe) {
} catch (MediaException me) {
} catch (InterruptedException e) { }
```


Camera

```
Player p;
VideoControl vc;
// initialize camera
try {
   p = Manager.createPlayer("capture://video");
   p.realize();
   // Grab the video control and set it to the current display.
   vc = (VideoControl)p.getControl("VideoControl");
   if (vc != null) {
      Form form = new Form("video");
      form.append((Item)vc.initDisplayMode(vc.USE_GUI_PRIMITIVE, null));
      Display.getDisplay(midlet).setCurrent(form);
   }
   p.start();
} catch (IOException ioe) {
} catch (MediaException me) { }
```

Camera

```
// now take a picture
try {
    byte[] pngImage = vc.getSnapshot(null);

    // do something with the image ...
} catch (MediaException me) { }
```