Sound in Java

Multimedia Systems (Module 1 Lesson 4)

Summary:

- Sound API Basics
- Playing Audio
  - In Memory (bounded)
  - Streamed (unbounded)
- Recording Sound
- MIDI

Sources:

- Chapter 18 of this book:
  Beginning Java2 - JDK 1.3
  Version by Ivor Horton
- Link to examples is available on the class website

Java Sound Preliminaries

File Formats Supported

- `.au` or `.snd` : usually stores 8-bit μ-law encoded samples, but can also store 8 or 16 bit linear samples
- `.aif` : usually stores 8 or 16 bit linear encoded samples
- `.wav` :Can store 8 or 16 bit samples using linear or μ-law encoded samples
- `.midi` : follows the midi data format

Note: The file header indicates the actual format

Frames and Frame Rates

- Sample Frame
  - Stores all the samples taken at an instant of time
  - # of bytes in a frame = # of bytes in a sample X number of channels
- Frame Rate
  - The number of frames per second of sound
  - In *most* cases frame rate is same as sample rate
  - In compressed sound, the frame rate will be less than sample rate.

Simple Sound Output

PlayIt.java

```
import java.applet.*;
import javax.swing.*;
import java.awt.event.*;

public class PlayIt extends JApplet
{
  AudioClip clip;    // The sound clip
  JButton button;
  final String play = "PLAY";
  final String stop = "STOP";

  public void init()
  {
    // Get the sound clip
    String fileName = getParameter("clip");
            // Get the file name
    clip =
      getAudioClip(getDocumentBase(),fileName);
            // Create the clip

    // Rest of the applet initialization...
    button = new JButton(play);
```

```
button.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        if(e.getActionCommand().equals(play))
        {
          clip.loop();
          button.setText(stop);
        }
        else
        {
          clip.stop();
          button.setText(play);
        }
      }
    }
  );
  getContentPane().add(button);
}
```

Playit.html

```
<APPLET CODE ="PlayIt.class" CODEBASE = "." WIDTH = 300 Height = 50 clip =
    "myClip.wav" >
</APPLET>
```

Sound in Applications

Similar to an Applet

- The `Applet` class defines a *static* method, `newAudioClip()` that retrieves an audio clip from a URL and returns a reference type `AudioClip` that encapsulates the audio clip.
  - This method being `static`, you don't have to have an `applet` object to call it. It can be called from an application as well, like so:

    ```
    AudioClip clip = Applet.newAudioClip(URL
      location)
    ```
  - Take a look at example `PlaySounds.java` in the examples directory

Java Sound API

System Resources

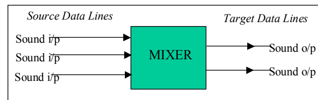
- *Audio Line:* Any resource that is a source or destination of sampled sound data
  - A line can encapsulate several channels
  - Example: input/output ports on the sound card
  - Lines have controls (*gain* and *pan* control)
- *Mixer:* Receives input from one or more *source data lines* and outputs the result of combining the input to an output line called a *target data line*

Other Sound Sources

- A file or more generally a URL

Terminology:

- A ***source data line*** is a source for a *mixer*, not a source for you; you ***write*** to it
- A **target data line** is the output from the *mixer*, your ***read*** from it



Java Sound API (…contd)

Packages:

- `javax.sound.sampled`
- `javax.sound.midi`
- `javax.sound.sampled.spi`
- `javax.sound.sampled.midi`

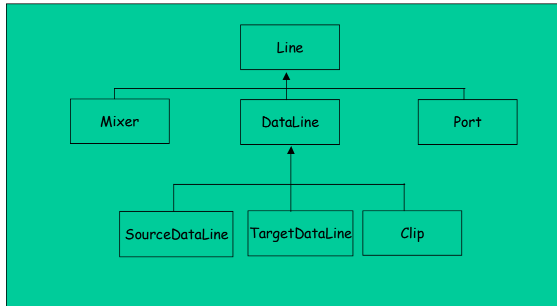
The `AudioSystem` class

- Establish whether a particular resource is available
- Get a ref. to the object that encapsulates the resource
- Call methods to operate the resource

`AudioInputStream` class

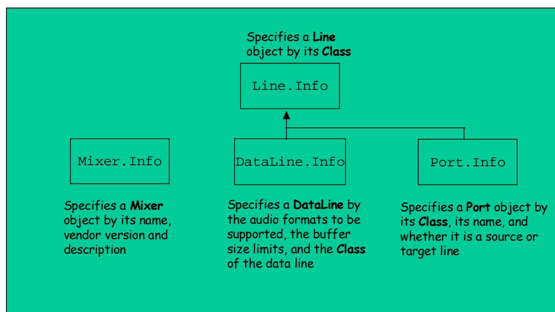
- Represents a stream that is a source of sampled sound data with a specific format
- You can create an `AudioInputStream` object from a local sound file, from another input stream or a URL
- You can
  - Read data from it
  - Write its contents to an output stream
  - Convert its format

Sampled Sound Interfaces



Resource Descriptor Classes

The `Line`, `DataLine`, `Mixer` and `Port` interface definitions each include an inner class with the name `Info`. Objects of these class types encapsulate data **specifying** an object of the corresponding type



Why Descriptor Class?

In order to answer that question, we have to look at the steps involved in playing audio:

1. Create an instance of a Line that specifies the **format** of the sound data you wish to play (*use descriptor class*).
2. Get a reference to a line (a DataLine, Port, Mixer or Clip) that supports this format
    1. May check if supported before requesting reference
3. Create an Audio Stream that encapsulates the sound data (file, URL, another stream)
    1. Extract the format from the Audio Stream
4. Tie the source of the sound data to the reference (line) that will play it. I.e., open the source
5. Play it; loop; goto; quit.

Playing a Clip vs. Stream

```
AudioInputStream source =
    AudioSystem.getAudioInputStream(file);
// Step 3.

DataLine.Info clipInfo = new
    DataLine.Info(Clip.class,
    source.getFormat());
// Step 1.

if(AudioSystem.isLineSupported(clipInfo))
{
   Clip newClip =
    (Clip)AudioSystem.getLine(clipInfo);
    // Step 2.

   newClip.open(source);// Step 4.
}

clip.loop(clip.LOOP_CONTINUOUSLY); // loop
clip.stop(); // stop
clip.setFramePosition(0);
Clip.close();
```

```
AudioInputStream newSource =
    AudioSystem.getAudioInputStream(file);
// Step 3.
AudioFormat format = newSource.getFormat();
DataLine.Info sourceInfo = new
    DataLine.Info(SourceDataLine.class,
    format); // Step 1.
if(AudioSystem.isLineSupported(sourceInfo))
{
  srcLine =
    (SourceDataLine)AudioSystem.getLine(sou
    rceInfo); // Step 2.
  bufferSize =
    (int)(format.getFrameSize()*format.getF
    rameRate()/2.0f);
  soundData = new byte[bufferSize];
  srcLine.open(format, bufferSize); //4.
}
while(playing)
{
  byteCount = source.read(soundData, 0,
    soundData.length); // Read the stream
  if(byteCount == -1)
  {
   sourceLine.drain(); // rest of buffer
   playing = false; break;
  }
  sourceLine.write(soundData,0, byteCount);
     // Write the array to the line
}
```

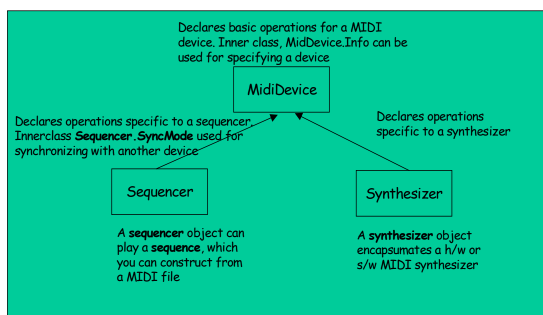
MIDI in JavaSound

- Data is a MIDI file is a series of commands that defines a piece of music
- Up to 16 MIDI channels are available (each instrument uses one channel)
- A MIDI Synthesizer reproduces(synthesizes) sounds in response to MIDI commands
  - H/W part of the sound card
  - S/W as in JavaSound
- A **sequencer** is a device that processes a MIDI sequence in order to play it on a synthesizer, or possible to edit it.
  - H/W or S/W

A device conforming to the General MIDI spec. must provide:

- A min. of 128 preset instruments + 47 percussive sounds
- A min. of 16 simultaneous **timbres** (instruments)
- A min. of 24 simultaneous **voices**, where each voice is a note of given velocity (loudness) for any of the available instruments and percussive sounds
- 16 midi channels, where each channel is polyphonic(can play multiple simultaneous voices). Percussive sounds are always on channel 10

MIDI Resources in JavaSound



Playing a MIDI file

To play a MIDI file, you don't need to access a synthesizer directly. All you need is a `Sequencer` reference and an object encapsulating the sequence that you want to play.

Steps:

1. sequencer = MidiSystem.getSequencer();
   // Get a sequencer
2. sequencer.open();
3. sequence = MidiSystem.getSequence(midiFile)
   // Encapsulate the midi src (file here; URL possible) in a sequence obj.
4. sequencer.setSequence(sequence);
   // Hand the sequence over to the sequencer
5. sequencer.start();
   // Play it. Stop it: sequencer.stop()