Chapter 5. Sound, Audio Effects, and Music Synthesis

We start by discussing the simple Applet play() method and AudioClip class, available since the early days of Java. However, most of this chapter is about the Java Sound API, introduced in J2SE 1.3, which supports the recording, playback, and synthesis of sampled audio and MIDI sequences.

An overview of the Sound API is illustrated with small examples showing the playback of clips, streamed audio, and MIDI sequences. This leads into a larger application, LoadersTests, which demonstrates our ClipsLoader and MidisLoader classes for loading, playing, pausing, resuming, stopping, and looping clips and sequences. These loader classes will be used in later applications when audio is required.

One of the key advantages of the Sound API over AudioClips, is the ability for a programmer to delve into the low-level details of audio files, and affect (to some degree) audio devices (e.g. the mixer, sequencer, or synthesizer). This permits a variety of audio effects to be applied, and we consider clip and MIDI channel controllers, sample byte array manipulation, and the modification of MIDI messages.

The synthesis of new audio during a game's execution can be useful. We look at how to generate tone sequences for sampled audio, and create sequences at run time. We describe tools/APIs that can help.

The Sound API is compared to the Java Media Framework (JMF), and the recently introduced JOAL, a Java binding to OpenGL's music API.

One large topic missing from this chapter is audio *capture*, which seems less necessary for games. Good Web resources on this topic (and others related to Java Sound) are listed at the end.

1. Applet Playing

The Applet play() method loads the sound (perhaps from across the network) and plays it once. play() causes the applet's drawing and event handling to freeze while the audio data is retrieved, and does nothing if the audio cannot be found (i.e. no exception is raised). The sound is marked for garbage collection after being played, so may need to be downloaded again when play() is called again later. A typical 1990's example:

```
import java.applet.Applet;
import java.awt.*;

public class OldMcDonald extends Applet
{
  public void init()
  {  play( getCodeBase(), "McDonald.mid");  }

  public void paint(Graphics g)
  {  g.drawString("Older McDonald", 25, 25);  }
}
```

The MIDI file (containing the tune "Old McDonald") is loaded and played as the applet is loaded.

Early versions of Java only supported 8-bit mono Windows Wave files, but the variety of format was considerably extended in JDK 1.2 to include Sun Audio (.au files), Mac AIFF files, MIDI (Musical Instrument Digital Interface) files (type 0 and type 1), and RMF (Rich Media Format). Data can be 8- or 16- bit, mono or stereo, with sample rates between 8,000 and 48,000 Hz.

getCodeBase() indicates that the file can be found in the same place as the applet's .class file. An alternative is getDocumentBase(), which specifies a location relative to the enclosing Web page.

2. The AudioClip Class

Many of the shortcomings with Applet's play() are remedied by the AudioClip class. Crucially, AudioClip separates loading from playing, and allow looping and termination via the loop() and stop() methods.

An updated McDonald applet using AudioClip:

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


public class McDonald extends JApplet
{
  private AudioClip mcdClip;

  public void init()
  {  mcdClip = getAudioClip(getCodeBase(), "mcdonald.mid");  }

  public void paint(Graphics g)
  {  g.drawString("Old McDonald", 25, 25);  }

  public void stop()
  {  mcdClip.stop(); }

  public void start()
  /* A looping play (and a call to play()) always starts at
     the beginning of the clip. */
  { mcdClip.loop();  }

} // end of McDonald.java
```

The clip is loaded with getAudioClip() in init(), causing the applet to suspend until the download is completed. The sound is played repeatedly due to the loop() call in start (), continuing until the applet is removed from the browser (triggering a call to stop()). If the page is displayed again, start()’s call to loop() will play the music from the beginning.

An application employs AudioClips in just about the same way, except that the clip is loaded with newAudioClip() from the Applet class, as shown in the PlaySound application:

```
import java.applet.Applet;
import java.applet.AudioClip;

public class PlaySound
{
  public PlaySound(String fnm)
  { try {
      AudioClip clip = Applet.newAudioClip(
                              getClass().getResource(fnm) );
      clip.play();  // play the sound once
    }
    catch (Exception e) {
      System.out.println("Problem with " + fnm);
    }
  }

  public static void main(String[] args)
  { if (args.length != 1) {
      System.out.println("Usage: java PlaySound <sound file>");
      System.exit(0);
    }
    new PlaySound(args[0]);
  }
}
```

Despite AudioClip's simplicity, useful applications/applets can be written with it. One of its great strengths is the large number of file formats that it supports. Another is that multiple AudioClips can be played at the same time.

A drawback is the suspension caused by calls to getAudioClip() and newAudioClip(). The Java Sound tutorial suggests threads as a solution: the tutorial's SoundApplet and SoundApplication examples fire off a separate thread to load the audio, allowing the main program to continue. Another answer is to download the sound resources with the code, wrapped inside a JAR, making the subsequent loading a local, fast operation.

A stubborn problem with AudioClip is the lack of knowledge about when a piece of audio finishes. This can be quite useful in games, for example when the next game session should only start when the audio commentary or introductory music has ended. A hacky work-around is to call sleep() for a period based on the audio file's byte size (which can be obtained via a File object).

A third issue is the lack of access to the sound data internals or the audio device, to permit run-time effects like volume changing, panning between speakers, and echoing. Related to this is the inability to generate new sounds during execution (i.e. sound/music synthesis), although many early Java texts proudly included variants of the following:

```
public class Bells
{
  public static void main(String[] args)
  {
    // \u0007 is the ASCII bell
    System.out.println("BELL 1\u0007");

    try {
      Thread.sleep(1000);   // separate the bells
    }
    catch(InterruptedException e) {}

    // ring the bell again, using Toolkit this time
    java.awt.Toolkit.getDefaultToolkit().beep();
    System.out.println("BELL 2");
    System.out.flush();
  } // end of main()

} // end of Bells
```

The ASCII character bell works on many platforms. Only Java applications can ring the Toolkit beep().

3. The SoundPlayer Application

SoundPlayer.java (located in SoundPlayer/) is a more realistic application using AudioClips. Figure 1 shows its GUI.

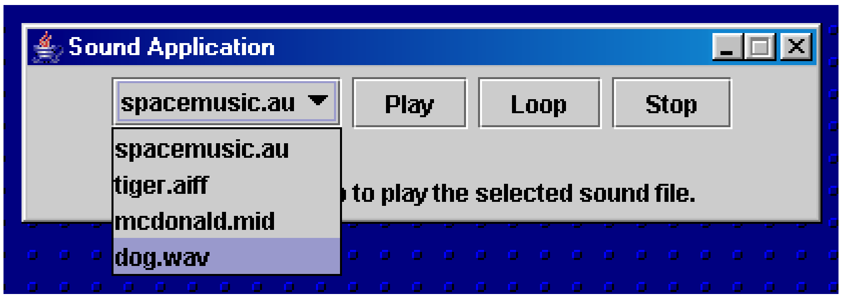


Figure 1. The SoundPlayer Application.

A selection of sound files in different formats (all located in the Sounds/ subdirectory) are offered up. They can be played once, looped, or stopped. It's possible to have multiple clips playing/looping at the same time, and the stop button terminates all the currently playing clips. This example is somewhat similar to the Java sound tutorial example, SoundApplication.

Figure 2 gives the class diagram for SoundPlayer, showing all the public and private methods and variables.

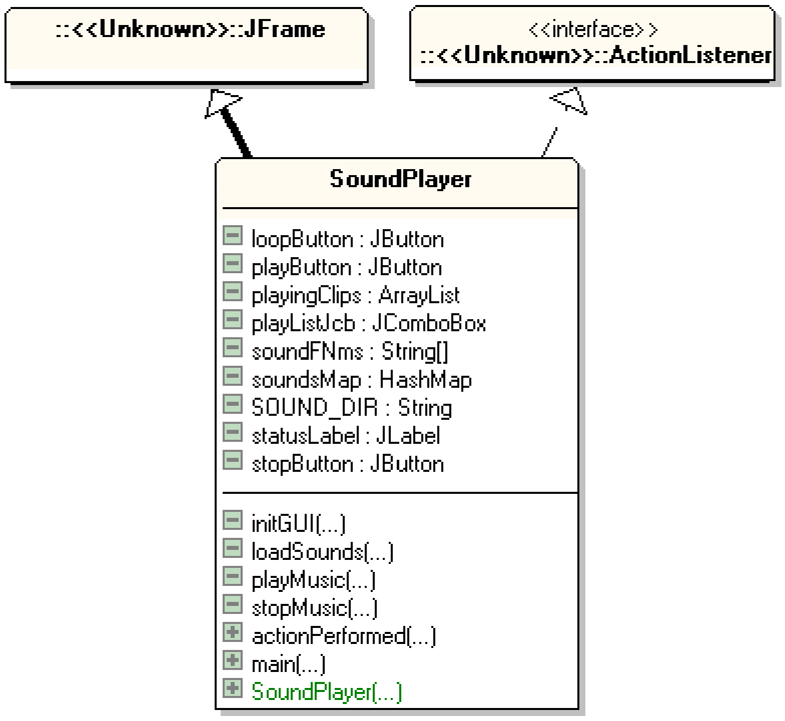


Figure 2. Class Diagram for SoundPlayer.

Two important data structures:

```
private HashMap soundsMap;
private ArrayList playingClips;
```

soundsMap holds the loaded AudioClips, indexed by their filenames. playingClips maintains a list of currently playing AudioClips (or, to be more precise, what we *think* is playing).

loadSounds() loads the AudioClips and stores them in soundsMap for later use.

```
private void loadSounds()
{ soundsMap = new HashMap();
  for (int i=0; i < soundFNms.length; i++) {
    AudioClip clip = Applet.newAudioClip(
                getClass().getResource(SOUND_DIR + soundFNms[i]) );
    if (clip == null)
     System.out.println("Problem loading "+SOUND_DIR+soundFNms[i]);
    else
      soundsMap.put(soundFNms[i], clip);
  }
}
```

newAudioClip() is employed since SoundPlayer is an application, and the URL is specified using the assumption that the files are locally stored in the SOUND_DIR subdirectory (Sounds/). The final version of SoundPlayer is a JAR file, created thus:

```
jar cvmf mainClass.txt SoundPlayer.jar SoundPlayer.class Sounds
```

All the class files and everything in the Sounds/ subdirectory are packed together. mainClass.txt contains a single line:

```
Main-Class: SoundPlayer
```

The JAR can be started by double-clicking on its icon, or from the command line:

```
> java –jar SoundPlayer.jar
```

playMusic() in SoundPlayer retrieves the relevant AudioClip, and starts it playing once or repeatedly. It also stores a reference to the clip in the playingClips ArrayList to register that the clip is playing.

```
private void playMusic(boolean toLoop)
{
  String chosenFile = (String) playListJcb.getSelectedItem();

  // try to get the AudioClip.
  AudioClip audioClip = (AudioClip) soundsMap.get(chosenFile);
  if (audioClip == null) {
    statusLabel.setText("Sound " + chosenFile + " not loaded");
    return;
  }

  if (toLoop)
    audioClip.loop();
  else
    audioClip.play();     // play it once

  playingClips.add(audioClip);   // store a ref to the playing clip
  String times = (toLoop) ? " repeatedly" : " once";
  statusLabel.setText("Playing sound " + chosenFile + times);
}  // end of playMusic()
```

playMusic() is called from actionPerformed() when the user presses the "Play" or "Loop" button, and is passed a toLoop argument to distinguish between the two.

stopMusic() stops all the playing music by calling AudioClip.stop() on all the references in playingClips. An issue is that some of the clips may already have finished, but there’s no way to detect it. This isn't really a problem since calling stop() on a stopped AudioClip has no effect.

```
private void stopMusic()
{
  if (playingClips.isEmpty())
    statusLabel.setText("Nothing to stop");
  else {
    AudioClip audioClip;
    for(int i=0; i < playingClips.size(); i++) {
      audioClip = (AudioClip) playingClips.get(i);
      audioClip.stop();   // may already have stopped, but calling
                          // stop() again does no harm
    }
```

```
      playingClips.clear();
      statusLabel.setText("Stopped all music");
    }
  }  // end of stopMusic()
```

4. The Java Sound API

The Java Sound API was introduced in J2SE 1.3, to support the capture, playback, and synthesis of sampled audio and MIDI sequences. The javax.sound.sampled package handles the former, javax.sound.midi the latter.

There are also two *service provider* packages, javax.sound.sampled.spi and javax.sound.midi.spi, to encourage extensibility. For instance, they can be utilized to add in new audio devices (i.e. new mixers, synthesizers), and formats (e.g. MP3).

In sections 5 and 6, sampled audio and MIDI is introduced, and code given for playing clips, buffered samples, and a MIDI sequence. Section 7 is a longer example, employing our ClipsLoader and MidisLoader classes for loading and playing clips and sequences.

Sections 8 and 9 are about applying audio effects to existing sampled audio and sequences.

Sections 10, 11, and 12 describe various approaches for the synthesis of audio samples and sequences.

5. Sampled Audio

Sampled audio is a series of digital samples extracted from analog signals, as illustrated by Figure 3. Each sample represents the amplitude (loudness) of the signal at a given moment.

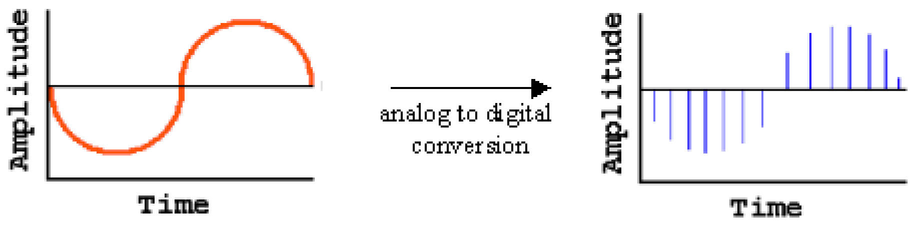


Figure 3. From Analog to Digital Audio.

The quality of the digital result depends on two factors: time resolution (the *sampling rate*), measured in Hertz (Hz), and amplitude resolution (*quantization*), the number of bits representing each sample. For example, a CD track is typically sampled at 44.1 KHz (44,100 samples/sec), and each sample uses 16 bits to encode a possible 65,536 amplitudes.

Descriptions of sampled audio often talk about *frames* (e.g. frame size, frame rate). For most audio formats, a frame is the number of bytes required to represent a single sample. For example, a sample in 8-bit mono PCM (pulse code modulation) format requires 1 frame (1 byte) per sample. 16-bit mono PCM samples require 2 frames, and 16-bit stereo PCM needs 4 frames: two bytes each for the left and right 16-bit samples in the stereo.

As the sample rate and quantization increases, so does the memory requirements. For instance, a 3 *second* stereo CD track, using 16-bit PCM, requires 44,100 * 4 * 3 bytes of space, or 517 KB. The '4' in the calculation reflects the need for 4 frames to store each stereo16-bit sample.

The higher the sample rate and quantization, the better the sound quality when the digital stream is converted back to an analog signal suitable for speakers or headphones. Figure 4 shows that the smoothness and detail of the signal depends on the number of samples and their amplitude accuracy.

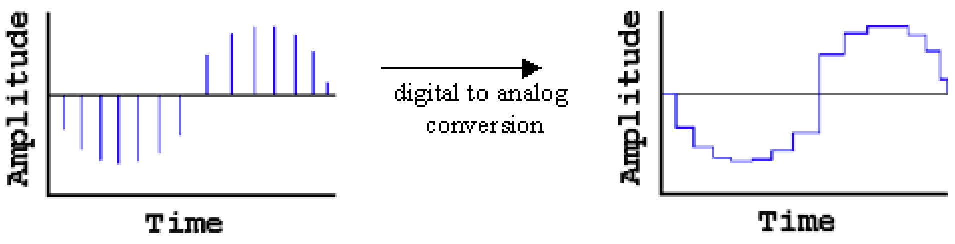


Figure 4. From Digital to Analog Audio.

Sampled audio can be encode with either the Clip or SourceDataLine classes, due to the sample size issue.

A Clip object holds sampled audio that is small enough to be loaded completely into memory during execution; as such, a Clip is quite similar to AudioClip. Small enough usually means less then 2-5 Mb.

A SourceDataLine is a buffered stream which permits 'chunks' of the audio to be delivered to the mixer in stages over time, without requiring the entire thing to be in memory at once.

The buffered streaming in SourceDataLine should not be confused with the video and audio streaming offered by JMF. The crucial difference is that JMF supports time-based protocols, such as RTP, which permits the audio software and hardware to manage the network latency and bandwidth issues when data chunks are transferred to it over a network. We say a little more about JMF in section 13.

Streaming in Java Sound does not have timing capabilities, making it difficult to maintain a constant flow of data through a SourceDataLine if the data is coming from the network; clicks and hisses will be heard as the system plays the sound. However, if SourceDataLine obtains its data from a local file, such problems are very unlikely to occur.

Section 5.2. describes how to play a clip, while section 5.3 considers how to play a buffered sample loaded from the local machine.

5.1. The Mixer

Both Clip and SourceDataLine are subclasses of the Line class; lines are the piping which allows digital audio to be moved around the audio system, for instance, from a microphone to the mixer, and from the mixer to the speakers (see Figure 5).

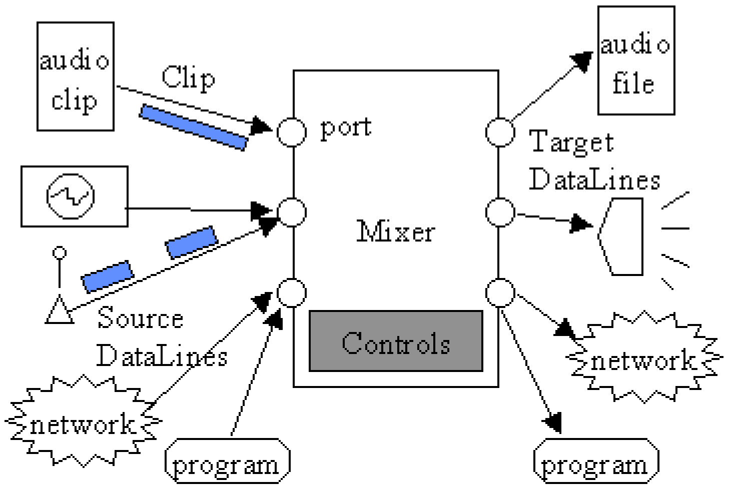


Figure 5. Audio IO to/from the Mixer.

Figure 5 is a stylized view of a mixer, intended to help explain the various classes and coding techniques for sampled audio.

Inputs to a mixer may include data read as a Clip object or streamed in from a device, the network, or generated by a program. Output can include audio written to a file, sent to a device, transmitted over the network, or sent as streamed output to a program.

The mixer, represented by the Mixer class, may be a hardware audio device (e.g. the sound card), or software interfaced to the sound card. A mixer can accept audio streams coming from several source lines, and pass them onto target lines, perhaps mixing the audio streams together in the process, and applying audio effects like volume adjustment or panning.

The capabilities of Java Sound's default mixer have changed in the transition from J2SE 1.4.2. to J2SE 1.5. In J2SE 1.4.2 or earlier, the default mixer was the "Java Sound Audio Engine", which had playback capabilities but could not capture sound; that was handled by another mixer. In J2SE 1.5, the "Direct Audio Device" is the default, supporting both playback and recording.

Clip, SourceDataLine and TargetDataLine are part of the Line class hierarchy shown in Figure 6.

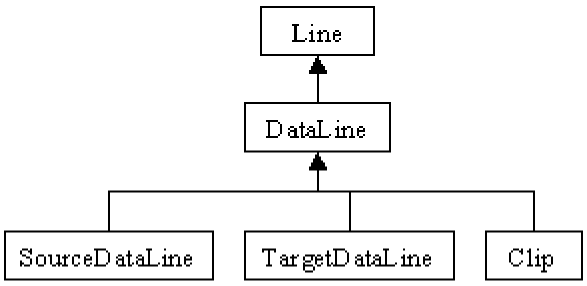


Figure 6. Part of the Line Hierarchy.

DataLine adds media features to Line, including the ability to determine the current read/write position, to start/stop/pause/resume the sound, and retrieve status details.

The SourceDataLine adds methods for buffering data for playback by the mixer. The name of the class is a little confusing: 'source' refers to a source of data *for the mixer*. From the programmer's point of view, data is written out to a SourceDataLine in order to send it a mixer.

The TargetDataLine is a streaming line in the same way as SourceDataLine. 'Target' refers to the destination of the data sent out by the mixer. For instance, an application might use a TargetDataLine to receive captured data gathered by the mixer from a microphone or CD drive. A TargetDataLine is a source of audio for the application.

A Clip is pre-loaded rather than streamed, so its duration is known before playback. This permits it to offer methods for adjusting the starting position and looping.

A LineListener can be attached to any line to monitor LineEvents, which are issued when the audio is opened, closed, started, or stopped. This last event can be utilized by application code to react to a sound's termination.

Figure 5 shows that lines are linked to the mixer through ports. A Port object typically allows access to the sound card features dealing with I/O. For example, an input port may be able to access the analog-to-digital converter. An output port may permit access to the digital-to-analog converter used by the speakers or headphones. A change to a port will affect all the lines connected to it. The Port class was not implemented prior to J2SE 1.5.

The box marked "Controls" inside the mixer allows audio effects to be applied to incoming clips or SourceDataLines. The effects may include volume control, panning between speakers, muting, and sample rate control, although the exact selection depends on the mixer. Section 8.3 describes an example using mixer controls on a clip.

Another form of audio manipulation is to modify the sample data before it is passed through a SourceDataLine to the mixer. For example, volume control is a matter of amplitude adjustment, coded by bit manipulation. Section 8.2 shows how this can be done.

5.2. Playing a Clip

PlayClip.java (in SoundPlayer/) loads an audio file specified on the command line as a clip, and plays it once.

The main() method creates a PlayClip object, and exits afterwards.

```
public static void main(String[] args)
{ if (args.length != 1) {
    System.out.println("Usage: java PlayClip <clip file>");
    System.exit(0);
  }
  new PlayClip(args[0]);
  System.exit(0);    // required in J2SE 1.4.2. or earlier
}
```

The call to exit() must be present in J2SE 1.4.2 or earlier (it’s unnecessary if you're using J2SE 1.5). The problem is that the sound engine does not terminate all of its threads when it finishes, which prevents the JVM from terminating without an exit() call.

The PlayClip class implements the LineListener interface in order to detect when the clip has finished. The LineListener update() method is described below.

```
public class PlayClip implements LineListener
{ ... }   // PlayClip must implement update()
```

The PlayClip() constructor loads and plays the clip.

```
public PlayClip(String fnm)
{
  df = new DecimalFormat("0.#");  // 1 dp
  loadClip(SOUND_DIR + fnm);
  play();

  // wait for the sound to finish playing; guess at 10 mins!
  System.out.println("Waiting");
  try {
    Thread.sleep(600000);   // 10 mins in ms
  }
  catch(InterruptedException e)
  { System.out.println("Sleep Interrupted"); }
}
```

The PlayClip constructor has a problem: it shouldn't return until the sound has finished playing. However, play() starts the sound playing and returns immediately, so the code must wait in some way. We choose to sleep for 10 minutes. Does this mean that PlayClip hangs around for 10 minutes even after it has finished played a 1 second clip? Fortunately, no. The LineListener update() method will allow PlayClip to exit as soon as the clip has ended.

loadClip() is the heart of PlayClip, and illustrates the low-level nature of Java Sound.

Its length is due to AudioSystem's lack of direct support for ULAW and ALAW formatted data. ULAW and ALAW are compression-based codings which affect the meaning of the bits in a sample. By default, only linear encodings (such as PCM) are understood.

The playing of a ULAW or ALAW file is dealt with by converting its data into PCM format as it is read into the Clip object. If we ignore this conversion code, and other error-handling, then loadClip() carries out six tasks:

```
// 1. access the audio file as a stream
AudioInputStream stream = AudioSystem.getAudioInputStream(
                                getClass().getResource(fnm) );

// 2. Get the audio format for the data in the stream
AudioFormat format = stream.getFormat();

// 3. Gather information for line creation
DataLine.Info info = new DataLine.Info(Clip.class, format);

// 4. create an empty clip using that line information
Clip clip = (Clip) AudioSystem.getLine(info);

// 5. Start monitoring the clip's line events
clip.addLineListener(this);

// 6. Open the audio stream as a clip; now it's ready to play
clip.open(stream);
```

The monitoring of the clip's line events, which include when it is opened, started, stopped, and closed, is usually necessary in order to react to the ending of a clip.

AudioInputStream can take its input from a file, input stream, or URL, so is a versatile way of obtaining audio input.

The full version of loadClip():

```
private void loadClip(String fnm)
{
  try {
    AudioInputStream stream = AudioSystem.getAudioInputStream(
                        getClass().getResource(fnm) );

    AudioFormat format = stream.getFormat();

    // convert ULAW/ALAW formats to PCM format
    if ( (format.getEncoding() == AudioFormat.Encoding.ULAW) ||
         (format.getEncoding() == AudioFormat.Encoding.ALAW) ) {
      AudioFormat newFormat =
         new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,
                      format.getSampleRate(),
                      format.getSampleSizeInBits()*2,
                      format.getChannels(),
                      format.getFrameSize()*2,
                      format.getFrameRate(), true);  // big endian
      // update stream and format details
      stream = AudioSystem.getAudioInputStream(newFormat, stream);
      System.out.println("Converted Audio format: " + newFormat);
      format = newFormat;
    }

    DataLine.Info info = new DataLine.Info(Clip.class, format);

    // make sure the sound system supports this data line
    if (!AudioSystem.isLineSupported(info)) {
      System.out.println("Unsupported Clip File: " + fnm);
      System.exit(0);
    }

    clip = (Clip) AudioSystem.getLine(info);
    clip.addLineListener(this);
    clip.open(stream);

    // duration (in secs) of the clip
    double duration = clip.getBufferSize() /
            (format.getFrameSize() * format.getFrameRate());
    System.out.println("Duration: " + df.format(duration)+" secs");
  } // end of try block

  catch (UnsupportedAudioFileException audioException) {
    System.out.println("Unsupported audio file: " + fnm);
    System.exit(0);
  }
  catch (LineUnavailableException noLineException) {
    System.out.println("No audio line available for : " + fnm);
    System.exit(0);
  }
  catch (IOException ioException) {
    System.out.println("Could not read: " + fnm);
    System.exit(0);
  }
  catch (Exception e) {
    System.out.println("Problem with " + fnm);
    System.exit(0);
  }
} // end of loadClip()
```

PCM creation uses the AudioFormat constructor:

```
  public AudioFormat(AudioFormat.Encoding encoding,
                  float sampleRate, int sampleSizeInBits,
                  int channels, int frameSize,
                  float frameRate, boolean bigEndian);
```

loadClip() uses the constructor like so:

```
 AudioFormat newFormat =
           new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,
                        format.getSampleRate(),
                        format.getSampleSizeInBits()*2,
                        format.getChannels(),
                        format.getFrameSize()*2,
                        format.getFrameRate(), true);  // big endian
```

ALAW and ULAW use an 8-bit byte to represent each sample, but after this has been decompressed the data requires 14-bits. Consequently, the PCM encoding must use 16-bits (2 bytes) per sample. This explains why the sampleSizeInBits and frameSize arguments are double the values obtained from the file's original audio format details.

Once the sample size goes beyond a single byte, the ordering of the multiple bytes must be considered. Big endian specifies a high-to-low byte ordering, while little endian is low-to-high. This is relevant if we later want to extract the sample's amplitude as a short or integer, since the multiple bytes must be combined together correctly.

The channels arguments refers to the use of mono (1 channel) or stereo (2 channels).

The audio encoding is PCM_SIGNED which allows a range of amplitudes that include negatives. For 16-bit data, the range will be –2^15 to 2^15-1 (-32768 to 32767). The alternative is PCM_UNSIGNED which only offers positive values, 0 to 2^16 (65536).

A snippet of code calculates the clip’s running time:

```
double duration = clip.getBufferSize() /
             (format.getFrameSize() * format.getFrameRate());
```

The clip contains the entire audio data, and so it’s possible to calculate its duration. getFrameRate() returns the number of frames played per second, while getFrameSize () specifies the number of bytes used per frame. Their multiplication gives the number of bytes played per second. getBufferSize() returns the total size of the clip in bytes, and the division gives the time to play all those bytes.

PlayClip's play() method is trivial:

```
private void play()
{ if (clip != null)
    clip.start();   // start playing
}
```

This starts the clip playing without waiting.

PlayClip sleeps, for as much as 10 minutes, while the clip plays. However, most clips will finish after a few seconds. Due to the LineListener interface, this will trigger a call to update():

```
public void update(LineEvent lineEvent)
// called when the clip's line detects open,close,start,stop events
{
  // has the clip reached its end?
  if (lineEvent.getType() == LineEvent.Type.STOP) {
    System.out.println("Exiting...");
    clip.stop();
    lineEvent.getLine().close();
    System.exit(0);
  }
}
```

The calls to stop() and close() aren’t really unnecessary, but ensure that the audio system resources are in the correct state before termination.

5.3. Playing a Buffered Sample

As Figure 5 suggests, a program can pass audio data to the mixer by sending discrete packets (stored in byte arrays) along the SourceDataLine. The main reason for using this approach is to handle very large audio files which cannot be loaded into a Clip.

BufferedPlayer.java does the same task as PlayClip.java – it plays an audio file supplied on the command line. The differences are only apparent inside the code. One cosmetic change is that the program is written as a series of static methods called from main(). This is just a matter of taste, the code could be 'objectified' to look similar to PlayClip.java.

```
// globals
private static AudioInputStream stream;
private static AudioFormat format = null;
private static SourceDataLine line = null;


public static void main(String[] args)
{ if (args.length != 1) {
    System.out.println("Usage: java BufferedPlayer <clip file>");
    System.exit(0);
  }

  createInput("Sounds/" + args[0]);
  createOutput();

  int numBytes = (int)(stream.getFrameLength() *
                              format.getFrameSize());
       // use getFrameLength() from the stream, since the format
       // version may return -1 (WAV file formats always return -1)
  System.out.println("Size in bytes: " + numBytes);

  play();
  System.exit(0);   // necessary in J2SE 1.4.2 and earlier
}
```

createInput() is quite similar to PlayClip's loadClip() method, but a little simpler. If we ignore the PCM conversion code for ULAW and ALAW formatted data, and other error handling, it does two tasks:

```
// access the audio file as a stream
stream = AudioSystem.getAudioInputStream( new File(fnm) );

// get the audio format for the data in the stream
format = stream.getFormat();
```

createOutput() creates the SourceDataLine going to the mixer.

```
private static void createOutput()
{
  try {
    // gather information for line creation
    DataLine.Info info =
          new DataLine.Info(SourceDataLine.class, format);
```

```
      if (!AudioSystem.isLineSupported(info)) {
        System.out.println("Line does not support: " + format);
        System.exit(0);
      }
      // get a line of the required format
      line = (SourceDataLine) AudioSystem.getLine(info);
      line.open(format);
    }
    catch (Exception e)
    {  System.out.println( e.getMessage());
       System.exit(0);
    }
  }  // end of createOutput()
```

createOutput() is doing two tasks as well: the collection of line information, and the creation of a SourceDataLine based on that information.

play() repeatedly reads a chunk of bytes from the AudioInputStream and writes them to the SourceDataLine, until the stream is empty. As a result, BufferedPlayer only requires memory large enough for the byte array buffer, not the entire audio file.

```
  private static void play()
  {
    int numRead = 0;
    byte[] buffer = new byte[line.getBufferSize()];

    line.start();
    // read and play chunks of the audio
    try {
      int offset;
      while ((numRead = stream.read(buffer,0,buffer.length)) >= 0) {
        offset = 0;
        while (offset < numRead)
          offset += line.write(buffer, offset, numRead-offset);
      }
    }
    catch (IOException e)
    {  System.out.println( e.getMessage()); }

    // wait until all data is played, then close the line
    line.drain();
    line.stop();
    line.close();
  }
```

The size of the buffer is determined by asking the SourceDataLine via getBufferSize (). Alternatively, we could calculate a size ourselves.

After the loop finishes, drain() causes the program to wait until all the data in the line has been passed to the mixer. Then it is safe for the line to be stopped, closed, and the program to terminate.

6. MIDI

A key benefit of the Musical Instrument Digital Interface (MIDI) is that it represents musical data in an extremely efficient way, leading to drastic reductions in file sizes compared to sampled audio. For instance, files containing high quality stereo sampled audio require about 10 Mb per minute of sound, while a typical MIDI sequence may need less than 10 Kb.

The secret to this phenomenal size reduction is that a MIDI sequence stores 'instructions' for playing the music rather than the music itself. A simple analogy is that a sequence is the written score for a piece of music rather than a recording of it.

The drawback is that the sequence must be converted to audio output at run-time. This is achieved using a sequencer and synthesizer. Their configuration is shown in greatly simplified form in Figure 7.

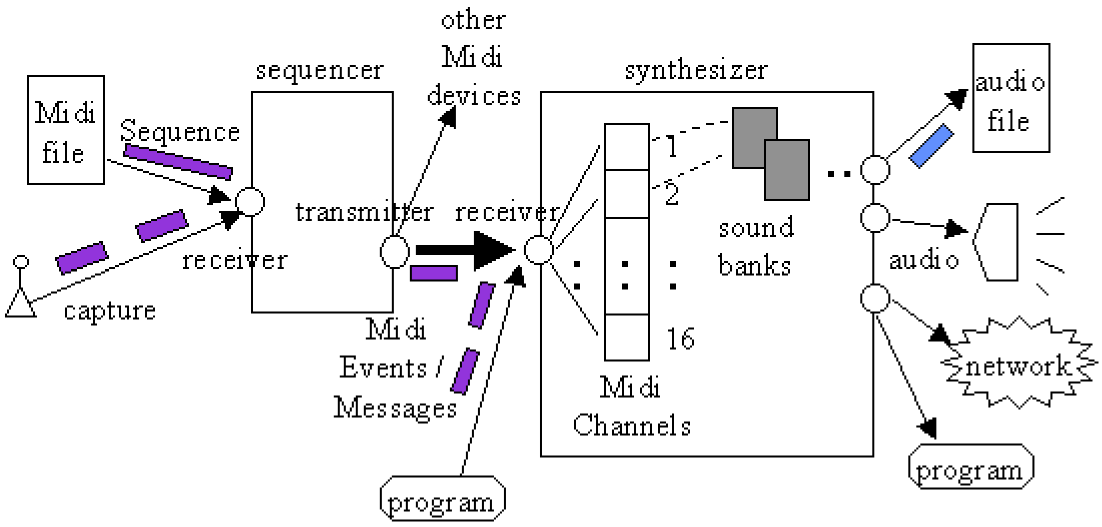


Figure 7. A MIDI Sequencer and Synthesizer.

A MIDI sequencer allows MIDI data sequences to be captured, stored, edited, combined, and performed, while the MIDI data's transformation into audio is carried out by the synthesizer.

Continuing our analogy, the sequencer is an orchestral conductor, who receives the score to play, perhaps making changes to it in the process. The synthesizer is the orchestra, made up of musicians playing different parts of the score. The musicians correspond to the MidiChannel objects in the synthesizer. They are allocated instruments from the sound banks, and play concurrently.

Usually a complete sequence (a complete score) is passed to the sequencer, but it's also possible to send it a stream of MIDI events. These are passed to the synthesizer (or other MIDI device) which can output audio in a variety of ways.

In J2SE 1.4.2 and earlier, the sequencer and synthesizer were represented by a single Sequencer object. This has changed in J2SE 1.5 – it's now necessary to obtain distinct Sequencer and Synthesizer objects and link them together using Receiver and Transmitter objects.

6.1. A MIDI Sequence

A Sequence object represents a multi-track data structure, each track containing time-ordered MIDIEvent objects. These events are time-ordered, based on an internal 'tick' value (a time-stamp). Each event contains musical data in a MidiMessage object. The sequence structure is illustrated in Figure 8.

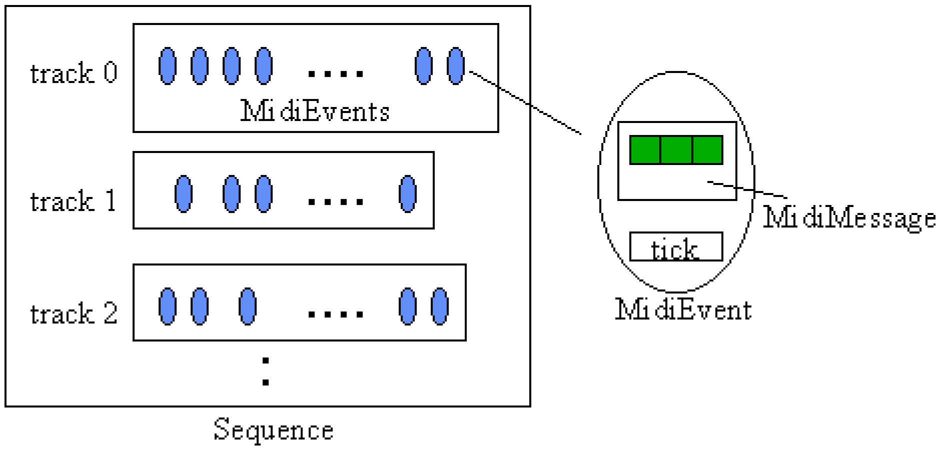


Figure 8. The Internals of a MIDI Sequence.

Tracks are employed as an optional organizational technique, to place 'related' MIDI data together; the synthesizer makes no use of the information. Java Sound supports Type 0 and Type 1 MIDI sequences, the main difference between them being that Type 0 files only have a single track.

MIDI messages are encoded using three subclasses of MidiMessage: ShortMessage, SysexMessage, and MetaMessage.

SysexMessage deals with system-exclusive messages, such as patch parameters or sample data sent between MIDI devices, which are usually specific to the MIDI device. MetaMessages are used to transmit 'meta' information about the sequence, such as tempo settings, and instrument information.

ShortMessage is the most important class, since it includes the NOTE_ON and NOTE_OFF messages for starting and terminating note playing on a given MidiChannel. Typically, one MidiEvent contains a NOTE_ON for beginning the playing, and a later MidiEvent holds a NOTE_OFF for switching it off. The duration of the note corresponds to the time difference between the tick values in the two events.

As shown in Figure 5, a program can directly communicate with the synthesizer, sending it a stream of MidiEvents or MidiMessages. The difference between the approaches is the timing mechanism: a stream of MidiEvents contain tick values, which the synthesizer can use to space out note playing and other activities. A stream of MidiMessages contains no timing data, and so it is up to the program to send the messages at the required time intervals. Examples of these techniques are given in section 11 on MIDI synthesis.

The internal format of a MidiMessage is very simple: there is a 8-bit status byte which identifies the message type, followed by two data bytes. Depending on the message, one or both of these bytes may be utilized. The byte size means that values usually range between 0 and 127.

One source of confusion for a programmer already familiar with MIDI is that the MidiMessage class and its subclasses do not correspond to the names used in the MIDI specification (see http://www.midi.org). ShortMessage includes the MIDI channel voice, channel mode, system common, and system real-time messages -- in other words, everything except system exclusive and meta events. In the rest of this chapter, we'll use the Java Sound MIDI class names.

6.2. Playing a MIDI Sequence

PlayMidi.java (stored in SoundPlayer/) loads a MIDI sequence and plays it once.

```
  public static void main(String[] args)
  { if (args.length != 1) {
      System.out.println("Usage: java PlayMidi <midi file>");
      System.exit(0);
    }
    new PlayMidi(args[0]);
    System.exit(0);    // required in J2SE 1.4.2. or earlier
  }
```

As with PlayClip, the call to exit() must be present in J2SE 1.4.2 or earlier (but is unnecessary in J2SE 1.5).

The PlayMidi class implements the MetaEventListener interface in order to detect when the sequence has reached the end of its tracks. This is done through the meta() method described below.

```
public class PlayMidi implements MetaEventListener
{
  // midi meta-event constant used to signal the end of a track
  private static final int END_OF_TRACK = 47;

  private final static String SOUND_DIR = "Sounds/";

  private Sequencer sequencer;
  private Synthesizer synthesizer;
  private Sequence seq = null;
  private String filename;

  private DecimalFormat df;
          :
}
```

The PlayMidi constructor initializes the sequencer and synthesizer, loads the sequence, and starts it playing.

```
  public PlayMidi(String fnm)
  {
    df = new DecimalFormat("0.#");  // 1 dp

    filename = SOUND_DIR + fnm;
    initSequencer();
    loadMidi(filename);
    play();

    // wait for the sound to finish playing; guess at 10 mins!
    System.out.println("Waiting");
    try {
      Thread.sleep(600000);   // 10 mins in ms
    }
    catch(InterruptedException e)
    { System.out.println("Sleep Interrupted"); }
  }
```

As with PlayClip, PlayMidi waits a while in order to give the sequence time to play. When the sequence finishes, the call to meta() allows PlayMidi to exit from its slumbers ahead of time.

initSequence() obtains a sequencer and synthesizer from the MIDI system, and links them together. It also sets up the meta event listener.

```
  private void initSequencer()
  {
    try {
      sequencer = MidiSystem.getSequencer();

      if (sequencer == null) {
        System.out.println("Cannot get a sequencer");
        System.exit(0);
      }

      sequencer.open();
      sequencer.addMetaEventListener(this);

      // maybe the sequencer is not the same as the synthesizer
      // so link sequencer --> synth (this is required in J2SE 1.5)
      if (!(sequencer instanceof Synthesizer)) {
        System.out.println("Linking the sequencer to a synthesizer");
        synthesizer = MidiSystem.getSynthesizer();
        Receiver synthReceiver = synthesizer.getReceiver();
        Transmitter seqTransmitter = sequencer.getTransmitter();
        seqTransmitter.setReceiver(synthReceiver);
      }
      else
        synthesizer = (Synthesizer) sequencer;
           // we don't use the synthesizer in this simple code,
           // so storing it as a global isn't really necessary
    }
    catch (MidiUnavailableException e){
      System.out.println("No sequencer available");
      System.exit(0);
    }
  } // end of initSequencer()
```

loadMidi() loads the sequence by calling MidiSystem.getSequence() inside a large try-catch block to catch the many possible kinds of errors.

```
private void loadMidi(String fnm)
{
  try {
    seq = MidiSystem.getSequence( getClass().getResource(fnm) );
    double duration =
            ((double) seq.getMicrosecondLength()) / 1000000;
    System.out.println("Duration: " + df.format(duration)+" secs");
  }
   // several catch blocks go here; see the code for details
}
```

play() loads the sequence into the sequencer and starts it playing.

```
private void play()
{ if ((sequencer != null) && (seq != null)) {
    try {
      sequencer.setSequence(seq);  // load MIDI into sequencer
      sequencer.start();   // start playing it
    }
    catch (InvalidMidiDataException e) {
      System.out.println("Corrupted/invalid midi file: " +
                                           filename);
      System.exit(0);
    }
  }
}
```

start() will return immediately, and PlayMidi will go to sleep back in the constructor.

meta() is called quite frequently as the sequence begins playing, but we're only interested in responding to the end-of-track event:

```
public void meta(MetaMessage event)
{ if (event.getType() == END_OF_TRACK) {
    System.out.println("Exiting...");
    close();
    System.exit(0);
  }
}
```

7. The LoadersTests Application

LoadersTests is primarily a test-bed for the ClipsLoader and MidisLoader classes, which offer a range of methods for loading, playing, pausing, resuming, stopping, and looping sounds. They will be used in later chapters for games requiring sounds or music. Figure 9 shows the LoadersTests GUI:

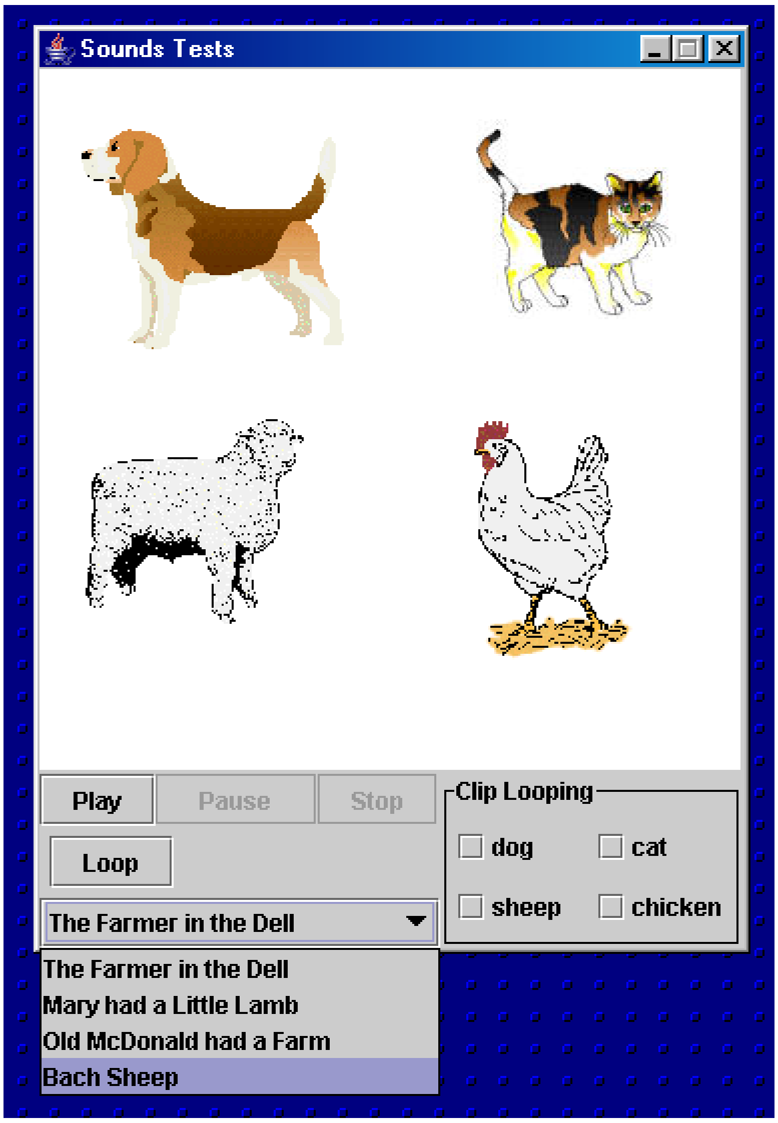


Figure 9. The LoadersTests Application.

The control panel on the left offers a choice between four MIDI sequences (all with a farming theme). The selection can be played once or repeatedly. Once playing, the "Pause" and "Stop" buttons are enabled. If the "Pause" button is pressed, the music pauses until resumed with the "Resume" button (which is the "Pause" button renamed). Only a single sequence can be played at a time.

The right-hand control panel is a series of check boxes for turning looping on and off for the 'dog', 'cat', 'sheep' and 'chicken' clips. A clip is started by the user clicking on the relevant image in the top half of the GUI. Multiple clips can be played at once, together with a MIDI sequence.

My personal favourite is a looping 'Old McDonald' with all the clips playing repeatedly as well. The joys of silence will soon become apparent.

The LoadersTests application is located in LoadersTests/.

Figure 10 shows the class diagrams for LoadersTests, with only the public methods visible.

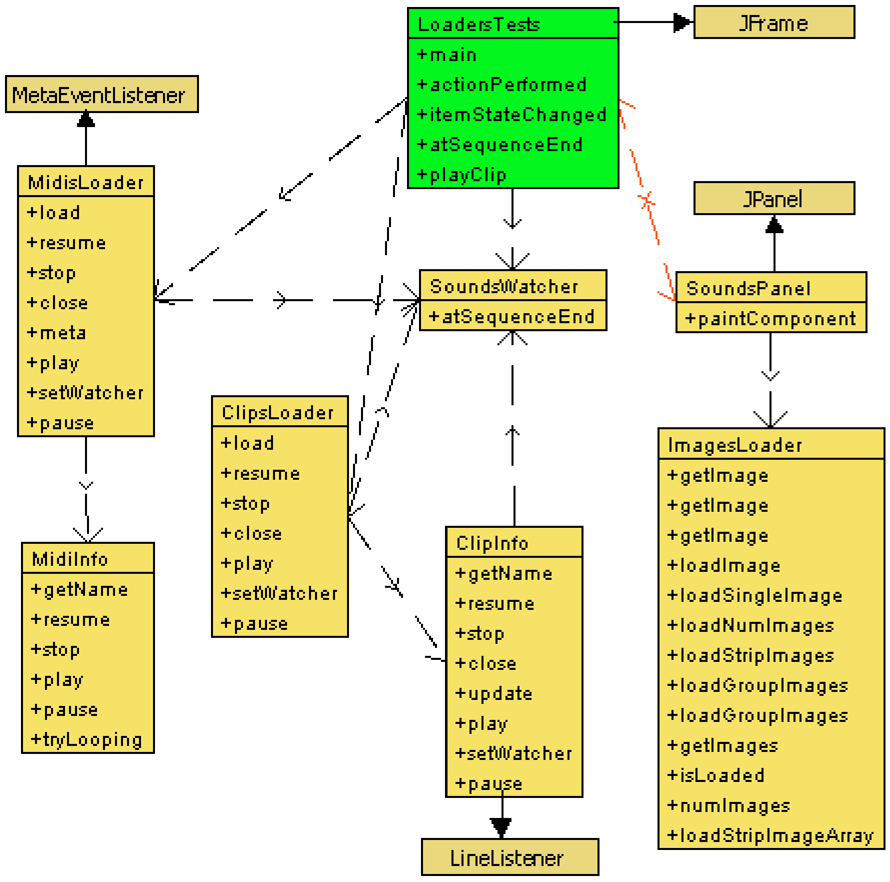


Figure 10. UML Class Diagrams for LoadersTests.

LoadersTests creates the GUI, initializes the loaders, and deals with user input. The images panel is coded by the SoundsPanel class.

The largest class is ImagesLoaders, previously described in chapter 4. It's used here to load the four animal GIFs – arguably an example of coding overkill for such simple tasks.

MidisLoader loads and manages multiple MIDI sequences, with each sequence stored in its own MidiInfo object. ClipsLoader does the same for clips, which are stored in ClipInfo objects.

The MIDI sequence and clips can be configured to call atSequenceEnd() in SoundsWatcher when they finish playing. In this example, LoadersTests implements the SoundsWatcher's interface.

7.1. The LoadersTests Class

The constructor for LoadersTests creates the images canvas (a SoundsPanel object), the rest of the GUI, and initializes the loaders:

```
// the clip and midi sound information files, located in Sounds/
private final static String SNDS_FILE = "clipsInfo.txt";
private final static String MIDIS_FILE = "midisInfo.txt";
        :

// global variables
private ClipsLoader clipsLoader;
private MidisLoader midisLoader;
        :

public LoadersTests()
{ super( "Sounds Tests" );
  Container c = getContentPane();
  c.setLayout( new BorderLayout() );

  SoundsPanel sp = new SoundsPanel(this);   // the images canvas
  c.add( sp, BorderLayout.CENTER);
  initGUI(c);                             // the rest of the controls

  // initialise the loaders
  clipsLoader = new ClipsLoader(SNDS_FILE);
  clipsLoader.setWatcher("dog", this);     // watch the dog clip

  midisLoader = new MidisLoader(MIDIS_FILE);
  midisLoader.setWatcher(this);         // watch the midi sequence

  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent ev) {
      midisLoader.close();   // shut down the sequencer
      System.exit(0);
    }
  });

  pack();
  setResizable(false);  // fixed size display
  centerFrame();        // placed in the center of the screen
  show();
}
```

As part of the loaders set-up, setWatcher() is called in the ClipsLoader and MidisLoader objects:

```
clipsLoader.setWatcher("dog", this);    // watch the dog clip
        :
midisLoader.setWatcher(this);           // watch midi playing
```

LoadersTests implements the SoundsWatcher interface, so the loaders will call its atSequenceEnd() method when the 'dog' clip reaches its end, and when the MIDI sequence ends. If the clip or sequence are looping, the method will be called each time the music reaches the end of the current cycle.

atSequenceEnd() is defined by LoadersTests as:

```
public void atSequenceEnd(String name, int status)
// can be called by the ClipsLoader or MidisLoader
{
  if (status == SoundsWatcher.STOPPED)
    System.out.println(name + " stopped");
  else if (status == SoundsWatcher.REPLAYED)
    System.out.println(name + " replayed");
  else
    System.out.println(name + " status code: " + status);
}
```

The two possible meanings of 'sequence end' are represented by the SoundsWatcher constants STOPPED and REPLAYED. The name argument is a string assigned to the clip or sequence by the loader.

When LoadersTests is terminated, windowClosing() calls close() in the MidisLoader to terminate its sequencer. This is preferable to relying on the audio system to release the resources. windowClosing() also calls exit() to force the JVM to terminate even though some audio threads are still running; this is not necessary in J2SE 1.5.

centerFrame() centers the JFrame in the middle of the screen. The code can be easily modified to place the window at any location.

7.2. The Listener Methods

In initGUI(), ActionListeners are attached to the buttons, and ItemListeners to the check boxes.

A simplified version of actionPerformed() is shown below, with the many calls to setEnable() edited out. We utilize setEnable() to manage the user's behaviour by restricting the available buttons: a useful GUI trick. When nothing is playing, "Play" and "Loop" are enabled. When a sequence is executing, only "Pause" and "Stop" are available. When a piece of music is paused, only the "Resume" button is active (this is the renamed "Pause" button).

```
public void actionPerformed(ActionEvent e)
/* Triggered by a "Play", "Loop", "Pause/Resume", "Stop" button
   press. The relevant method in MidisLoader is called.

   A lot of effort is spent on disabling/enabling buttons,
   which I've edited out from the code here.
*/
{ // which song is currently selected?
  String songName = shortSongNames[ namesJcb.getSelectedIndex() ];

  if (e.getSource() == playJbut)       // "Play" pressed
    midisLoader.play(songName, false); // play sequence, no looping
  else if (e.getSource() == loopJbut)  // "Loop" pressed
    midisLoader.play(songName, true);  // play with looping
  else if (e.getSource() == pauseJbut) {  // "Pause/Resume" pressed
    if (isPauseButton) {
      midisLoader.pause();    // pause the sequence
      pauseJbut.setText("Resume");      // Pause --> Resume
    }
```

```
      else {
        midisLoader.resume();  // resume the sequence
        pauseJbut.setText("Pause");      // Resume --> Pause
      }
      isPauseButton = !isPauseButton;
    }
    else if (e.getSource() == stopJbut)   // "Stop" pressed
      midisLoader.stop();   // stop the sequence
    else
      System.out.println("Action unknown");
  }  // end of actionPerformed()
```

The correspondence between button presses and calls to the MidisLoader is quite clear. A once-only play and repeated playing of a clip are both handled by play() with a boolean argument to distinguish the mode.

itemStateChanged() handles the four checkboxes on the right side of the GUI which specify if clips should be looped when played. However, a clip only starts to play when the user clicks on its image in the SoundsPanel.

The approach is to maintain the looping settings for all the clips in an array of booleans called clipLoops[]. The relevant boolean is passed to ClipsLoader's play() method when the clip is played.

```
  // global clip image names (used to label the checkboxes)
  private final static String[] names =
                        {"dog", "cat", "sheep", "chicken"};
            :
  // global clip loop flags, stored in names[] order
  private boolean[] clipLoops = {false, false, false, false};
            :

  public void itemStateChanged(ItemEvent e)
  // Triggered by selecting/deselecting a clip looping checkbox
  {
    // get the name of the selected checkbox
    String name = ((JCheckBox)e.getItem()).getText();
    boolean isSelected =
            (e.getStateChange() == e.SELECTED) ? true : false ;

    boolean switched = false;
    for (int i=0; i < names.length; i++)
      if (names[i].equals(name)) {
        clipLoops[i] = !clipLoops[i];   // update the clip loop flags
        switched = true;
        break;
      }
    if (!switched)
      System.out.println("Item unknown");
    else {
      if (!isSelected)   // user just switched off looping for name
        clipsLoader.stop(name);    // so stop playing name's clip
    }
  }
```

The checkbox's name is found in the names[] array, and the corresponding index is used to choose the boolean in clipsLoops[] that is modified.

A quirk of LoadersTests' GUI is the lack of a button to stop a repeating clip. Instead, the deselection of its looping checkbox causes it to stop. This is perhaps rather counter-intuitive. Design decisions like this one should be tested on users who are uninvolved in the application's design or implementation.

LoadersTests also has no interface for allowing a clip to be paused and resumed, although this functionality is present in ClipsLoader.

7.3. The SoundsPanel Class

SoundsPanel implements a JPanel which draws a white background and four images. The interesting part of the code is the setting up and use of the images' hot-spots.

When a user clicks inside the panel, its coordinates are checked against a series of a rectangular areas with the same coordinates as the images. If the mouse press is inside one of the areas (hot-spots) then LoadersTests' playClip() plays the associated clip.

The SoundsPanel constructor stores a reference to LoadersTests, calls initImages(), and sets up the MouseListener to call selectImage().

```
// globals
private static final int PWIDTH = 350;     // size of this panel
private static final int PHEIGHT = 350;
     :
private LoadersTests topLevel;
     :

public SoundsPanel(LoadersTests sts)
{ topLevel = sts;
  setPreferredSize( new Dimension(PWIDTH, PHEIGHT) );
  initImages();
  addMouseListener( new MouseAdapter() {
    public void mousePressed( MouseEvent e)
    { selectImage( e.getX(), e.getY());  }
  } );
}
```

initImages() uses ImagesLoader to load the four GIFs, whose names are hard-wired into the code in the names[] array. The width and height of each image is used to build the array of Rectangle objects that represent the hot-spots.

```
// globals
// clip image names
private final static String[] names =
                    {"dog", "cat", "sheep", "chicken"};

// on-screen top-left coords for the images
private final static int[] xCoords = {20, 210, 20, 210};
private final static int[] yCoords = {25, 25, 170, 170};

// location of image and sound info
private final static String IMS_FILE = "imagesInfo.txt";
private int numImages;
private BufferedImage[] images;
private Rectangle[] hotSpots;
    // a click inside these triggers the playing of a clip
     :

private void initImages()
// load and initialise the images, and build their 'hot-spots'
{
  numImages = names.length;
  hotSpots = new Rectangle[numImages];
  images = new BufferedImage[numImages];

  ImagesLoader imsLoader = new ImagesLoader(IMS_FILE);

  for (int i=0; i < numImages; i++) {
    images[i] = imsLoader.getImage(names[i]);
    hotSpots[i] = new Rectangle( xCoords[i], yCoords[i],
                     mages[i].getWidth(), images[i].getHeight());
      // use images' dimensions for the size of the rectangles
  }
}
```

Each hot-spot rectangle is defined by a top-left coordinate, taken from the xCoords[] and yCoords[] arrays, and a width and height obtained from the loaded image.

paintComponent() draws the images in the panel using the same xCoords[] and yCoords[] data as the hot-spot rectangles, thereby ensuring that they occupy the same spaces.

selectImage() tries to find the hot-spot containing the mouse press coordinates. A matching hot-spot's index position in hotSpots[] is used to retrieve a clip name from names[]. playClip() is passed the name and the index.

```
private void selectImage(int x, int y)
/* Work out which image was clicked on (perhaps none),
   and request that its corresponding clip be played. */
{
  for (int i=0; i < numImages; i++)
    if (hotSpots[i].contains(x,y)) {    // (x,y) inside hot-spot?
      topLevel.playClip(names[i], i);   // play that name's clip
      break;
    }
}
```

Back in LoadersTests, playClip() is defined as:

```
public void playClip(String name, int i)
// called from SoundsPanel to play a given clip (looping or not)
{  clipsLoader.play(name, clipLoops[i]);  }
```

The index parameter is employed to look inside clipLoops[] to get the playing mode. Crucially, we must ensure that the clipLoops[] array refers to the clips in the same order as the arrays in SoundsPanel.

7.4. The ClipsLoader Class

ClipsLoader stores a collection of ClipInfo objects in a HashMap, keyed by their names.

The name and filename for a clip are obtained from a sounds information file, which is loaded when ClipsLoader is created. The information file is assumed to be in Sounds/.

ClipsLoader allows a specified clip to be played, paused, resumed, looped, and stopped. A SoundsWatcher can be attached to a clip. All of this functionality is handled in the ClipInfo object for the clip.

It is possible for many clips to play at the same time, since each ClipInfo object is responsible for playing its own clip.

There are two ClipsLoader constructor: one of them loads a sounds information file.

```
// globals
private HashMap clipsMap;
  /* The key is the clip 'name', the object (value)
     is a ClipInfo object */


public ClipsLoader(String soundsFnm)
{ clipsMap = new HashMap();
  loadSoundsFile(soundsFnm);
}

public ClipsLoader()
{  clipsMap = new HashMap();  }
```

loadSoundsFile() parses the information file, assuming each line contains a name and filename. For example, "clipsInfo.txt" used by LoadersTests is:

```
// sounds
cat cat.wav
chicken chicken.wav
dog dog.wav
sheep sheep.wav
```

The name can be any string. The file may also contain blank lines and comment lines beginning with '//'.

After a line's name and filename have been extracted, load() is called:

```
public void load(String name, String fnm)
// create a ClipInfo object for name and store it
{
  if (clipsMap.containsKey(name))
    System.out.println( "Error: " + name + "already stored");
  else {
    clipsMap.put(name, new ClipInfo(name, fnm) );
    System.out.println("-- " + name + "/" + fnm);
  }
}  // end of load()
```

A ClipInfo object is created, and added to the HashMap.

load() is public so a user can also directly add clips to the loader.

play() illustrates the coding style used in the other public methods in ClipsLoader (i.e. in close(), stop(), pause(), resume(), and setWatcher()).

```
  public void play(String name, boolean toLoop)
  // play (perhaps loop) the specified clip
  {  ClipInfo ci = (ClipInfo) clipsMap.get(name);
     if (ci == null)
       System.out.println( "Error: " + name + "not stored");
     else
      ci.play(toLoop);   // delegate operation to ClipInfo obj
  }
```

Audio manipulation is delegated to the ClipInfo object associated with the specified clip name.

7.5. The ClipInfo Class

A ClipInfo object is responsible for loading a clip, and plays, pauses, resumes, stops, and loops it when requested by ClipsLoader. Also, an object implementing the SoundsWatcher interface can be notified when the clip loops or stops.

Much of the manipulation carried out by ClipInfo, such as clip loading, is virtually identical to that found in PlayClip.java in section 5.2. Perhaps the largest difference is that PlayClip tends to exit when it encounters a problem while ClipInfo prints an error message, and attempts to soldier on.

loadClip() is very similar to PlayClip's loadClip(), so certain parts have been commented away in the code below to simplify matters:

```
  // global
  private Clip clip = null;
     :

  private void loadClip(String fnm)
  {
    try {
      // 1. access the audio file as a stream
      AudioInputStream stream = AudioSystem.getAudioInputStream(
                          getClass().getResource(fnm) );

      // 2. Get the audio format for the data in the stream
      AudioFormat format = stream.getFormat();

      // convert ULAW/ALAW formats to PCM format...
      // several lines, which update stream and format

      // 3. Gather information for line creation
      DataLine.Info info = new DataLine.Info(Clip.class, format);

      // make sure the sound system supports the data line
      if (!AudioSystem.isLineSupported(info)) {
        System.out.println("Unsupported Clip File: " + fnm);
        return;
      }

      // 4. create an empty clip using the line information
      clip = (Clip) AudioSystem.getLine(info);

      // 5. Start monitoring the clip's line events
      clip.addLineListener(this);

      // 6. Open the audio stream as a clip; now it's ready to play
      clip.open(stream);
    } // end of try block

    // several catch blocks go here ...
  } // end of loadClip()
```

play() starts the loop playing:

```
  public void play(boolean toLoop)
  { if (clip != null) {
      isLooping = toLoop;    // store playing mode
      clip.start(); // start playing from where stopped
    }
  }
```

The Clip class has a loop() method, which is *not* used by our play() when toLoop is true. Instead the looping mode is stored in the isLooping global, and utilized later in update().

Clip's start() is asynchronous, so our play() method will not suspend. This makes it possible for a user to start multiple clips playing at the same time. If play() is called again for an already playing clip, start() has no effect.

Our stop() method stops the clip, and resets it to the beginning, ready for future playing:

```
  public void stop()
  { if (clip != null) {
      isLooping = false;
      clip.stop();
      clip.setFramePosition(0);
    }
  }
```

setFramePosition() can set the playing position anywhere inside the clip.

Our pause() and resume() are similar to stop() and play():

```
public void pause()
// stop the clip at its current playing position
{ if (clip != null)
    clip.stop();
}

public void resume()
{ if (clip != null)
    clip.start();
}
```

pause() does not reset the clip's playing position. Consequently, resume() will start playing the clip from the point where the sound was suspended.

ClipInfo implements the LineListener interface, and so is notified when the clip generates line events. Our update() only deals with STOP events.

```
public void update(LineEvent lineEvent)
{
  // when clip is stopped / reaches its end
  if (lineEvent.getType() == LineEvent.Type.STOP) {
    clip.stop();
    if (!isLooping) {  // it isn't looping
      if (watcher != null)
        watcher.atSequenceEnd(name, SoundsWatcher.STOPPED);
    }
    else {      // else play it again
      clip.start();
      if (watcher != null)
        watcher.atSequenceEnd(name, SoundsWatcher.REPLAYED);
    }
  }
} // end of update()
```

A STOP event is triggered in two slightly different situation: when the clip reaches its end and when the clip is stopped with Clip.stop().

When the clip reaches its end, it may have been set to loop. This is *not* implemented by using Clip's loop() method, but instead by examining the value of the global isLooping boolean.

If isLooping is false, then the watcher (if one exists) is told that the clip has stopped. If isLooping is true then the clip is started again, and the watcher is told that the clip is playing again.

This explicit restarting of a 'looping' clip allows us to insert additional processing (e.g. watcher notification) between the clip's finishing and restarting.

7.6. The MidisLoader Class

MidisLoader stores sequences as a collection of MidiInfo objects in a HashMap, keyed by their names. The name and filename for a sequence are obtained from an information file loaded when MidisLoader is created. The file is assumed to be in Sounds/.

MidisLoader allows a specified sequence to be played, stopped, resumed, looped. A SoundsWatcher can be attached to the *sequencer* (not to a sequence).

MidisLoader deliberately offers almost the same interface as ClipsLoader (see Figure 10), although internally there are some significant differences.

MidisLoader contains a single Sequencer object for playing all the sequences. Consequently, only one sequence can be played at a time; this contrasts with ClipsLoader where multiple clips can be playing.

A reference to the sequencer is passed to each MidiInfo object, thereby giving them the responsible for playing, stopping, resuming and looping their sequences.

The MidisLoader initializes the sequencer using initSequencer(), and loads the information file.

```
// globals
private Sequencer sequencer;
private HashMap midisMap;
private MidiInfo currentMidi = null;
      // reference to currently playing MidiInfo object
   :

public MidisLoader()
{ midisMap = new HashMap();
  initSequencer();
}

public MidisLoader(String soundsFnm)
{ midisMap = new HashMap();
  initSequencer();
  loadSoundsFile(soundsFnm);
}
```

The simpler versions of the constructor allows the loader to be created without an information file.

initSequencer() is essentially the same as the version in PlayMidi.java in section 6.2.

loadSoundsFile() is very similar to the same named method in ClipsLoader: it parses the information file, assuming each line contains a name and filename. For example, "midisInfo.txt" used by LoadersTests is:

```
// midis
baa bsheep.mid
farmer farmerinthedell.mid
mary maryhadalittlelamb.mid
mcdonald mcdonald.mid
```

The name can be any string. The file may also contain blank lines and comment lines beginning with '//'.

After a line's name and filename have been extracted, load() is called:

```
public void load(String name, String fnm)
// create a MidiInfo object, and store it under name
{
  if (midisMap.containsKey(name))
    System.out.println( "Error: " + name + "already stored");
  else if (sequencer == null)
    System.out.println( "No sequencer for: " + name);
  else {
    midisMap.put(name, new MidiInfo(name, fnm, sequencer) );
    System.out.println("-- " + name + "/" + fnm);
  }
}
```

This creates a MidiInfo object for the sequence and stores it in the midisMap HashMap. The last MidiInfo constructor argument is the sequencer.

Playing a sequence is a matter of looking up the specified name in midisMap, and calling its play() method. A slight complication is that only a single sequence can be played at a time. This is dealt with by storing a reference to the currently playing MidisInfo object in the currentMidi global.

```
public void play(String name, boolean toLoop)
// play (perhaps loop) the sequence
{
   MidiInfo mi = (MidiInfo) midisMap.get(name);
   if (mi == null)
     System.out.println( "Error: " + name + "not stored");
   else {
     if (currentMidi != null)
       System.out.println("Sorry, " + currentMidi.getName() +
                              " already playing");
     else {
      currentMidi = mi;  // store a reference to playing midi
      mi.play(toLoop);   // pass play request to MidiInfo object
     }
   }
} // end of play()
```

Playing is prohibited if currentMidi is not null, which means that a sequence is already playing.

Pausing and resuming is handled by passing the tasks to the currently playing MidiInfo object:

```
public void pause()
{ if (currentMidi != null)
    currentMidi.pause();
  else
    System.out.println( "No music to pause");
}

public void resume()
{ if (currentMidi != null)
    currentMidi.resume();
  else
    System.out.println("No music to resume");
}
```

Stopping a sequence uses the same delegation strategy. The stop() method in MidisInfo will trigger an end-of-track meta event in the sequencer, which is handled by MidisLoader' meta() method.

```
public void stop()
{ if (currentMidi != null)
    currentMidi.stop();  // this will cause an end-of-track event
    System.out.println("No music playing");
}


public void meta(MetaMessage meta)
{
  if (meta.getType() == END_OF_TRACK) {
    String name = currentMidi.getName();
    boolean hasLooped = currentMidi.tryLooping();
                              // music still looping?
    if (!hasLooped)   // no it's finished
      currentMidi = null;

    if (watcher != null) {   // tell the watcher
      if (hasLooped)        // the music is playing again
        watcher.atSequenceEnd(name, SoundsWatcher.REPLAYED);
      else                  // the music has finished
        watcher.atSequenceEnd(name, SoundsWatcher.STOPPED);
    }
  }
} // end of meta()
```

The code in meta() only deals with an end-of-track meta event. This kind of event is triggered by a MidiInfo object when its sequence reaches its end *or* is stopped. However, a sequence at its end may be looping, which is checked by calling tryLooping() in MidiInfo. If there is a watcher, it is notified of the status.

As LoadersTests terminates, it calls close() in MidisLoader to release the sequencer.

```
public void close()
{
  stop();    // stop the playing sequence
  if (sequencer != null) {
    if (sequencer.isRunning())
      sequencer.stop();

    sequencer.removeMetaEventListener(this);
    sequencer.close();
    sequencer = null;
  }
}  // end of close()
```

7.7. The MidiInfo Class

A MidiInfo object holds a single MIDI sequence, and a reference to the sequencer created in MidisLoader. This allows it to play, stop, pause, and resume a clip, and make it loop.

The constructor is passed the sequence's name, filename, and the sequencer reference, and then loads the sequence using MidiSystem.getSequence().

A sequence is played by loading it into the sequencer, and starting it.

```
public void play(boolean toLoop)
{ if ((sequencer != null) && (seq != null)) {
    try {
      sequencer.setSequence(seq);   // load sequence into sequencer
      sequencer.setTickPosition(0); // reset to the start
      isLooping = toLoop;
      sequencer.start();            // play it
    }
    catch (InvalidMidiDataException e) {
      System.out.println("Invalid midi file: " + filename);
    }
  }
}
```

The Sequencer class has several loop() methods, but they are *not* used here. A similar coding technique is employed as in ClipInfo: a global isLooping boolean is set to true, and employed later by tryLooping().

Stopping a sequence with Sequencer.stop() causes it to stop at its current position. More importantly, no meta event is generated unless the stopping also coincides with the end of the track. In order to generate an event, our stop() method 'winds' the sequence to its end.

```
public void stop()
{
  if ((sequencer != null) && (seq != null)) {
    isLooping = false;
    if (!sequencer.isRunning())   // the sequence may be paused
      sequencer.start();
    sequencer.setTickPosition( sequencer.getTickLength() );
       // move to end of sequence to trigger end-of-track event
  }
}
```

This behaviour means that meta() in MidisLoader is called in two situations: when the sequence reaches its end *and* when the sequence is stopped. This corresponds to the ways that a LineListener STOP event can be generated for clips.

MidisLoader's meta() calls tryLooping() in MidiInfo to determine if the sequence is looping or not. tryLooping() is responsible for restarting the sequence if its isLooping boolean is true.

```
public boolean tryLooping()
{
  if ((sequencer != null) && (seq != null)) {
    if (sequencer.isRunning())
      sequencer.stop();
    sequencer.setTickPosition(0);
    if (isLooping) {    // play it again
      sequencer.start();
      return true;
    }
  }
  return false;
}
```

Admittedly, this is rather convoluted coding: stop() triggers meta() which calls tryLooping(). tryLooping() restarts a looping sequence.

Part of the problem is that looping is not implemented with Sequencer.loop(). Instead a sequence comes to its end, and is started again by tryLooping() calling start(). The allows additional processing in meta() (e.g. watcher communication) between the end of the sequence and its restart.

Another aspect is that the sequence control code is located in MidiInfo (stop() and tryLooping()), but the meta-event processing is inside meta() in MidisLoader.

MidiInfo's pause() and resume() are implemented using the Sequencer class's start() and stop(). These Sequencer methods do not adjust the sequence's playing position.

```
public void pause()
{ if ((sequencer != null) && (seq != null)) {
    if (sequencer.isRunning())
      sequencer.stop();
  }
}

public void resume()
{ if ((sequencer != null) && (seq != null))
    sequencer.start();
}
```

7.8. LoadersTests as a JAR

It is straightforward to package the LoadersTests code, its images, and sounds into a JAR:

```
> jar cvmf mainClass.txt LoadersTests.jar *.class Sounds Images
> jar i LoadersTests.jar
```

All the class files and everything in the Sounds/ and Images/ subdirectories are packed together. The "jar i" calls adds indexing information to the JAR, which will speed up the execution of JARS containing many files.

mainClass.txt contains a single line:

```
Main-Class: LoadersTests
```

The JAR can be started by double-clicking on its icon, or from the command line:

```
> java –jar LoadersTests.jar
```

One serious issue to consider before releasing applications is copyright. In MS Windows, the copyright for sounds can be obtained as a side-effect of playing them in the Windows Media Player: the details appear in the GUI. For example, the "Bach Sheep" and "Farmer in the Dell" sequences used in LoadersTests, and by several other of my examples, are copyrighted by David E Lovell and Diversified Software respectively.

8. Audio Effects on Sampled Audio

Audio effects include modifying a sound so that is gets louder, quieter, plays faster or slower, echoes, or is sent to one speaker. We discuss three approaches for creating these kinds of effects:

1. Pre-calculation;
2. Byte Array Manipulation;
3. Utilizing Mixer Controls.

8.1. Pre-calculation

Manipulating audio inside Java can be time-consuming and complicated. If the sound effect is going to be used regularly (e.g. a fading scream, an echoing explosion), then it’s probably better to create a new sound file with an audio editor.

I've found WavePad useful for a variety of editing, format conversion, and effects tasks (http://nch.com.au/wavepad/). Its supported effects include: amplification, reverberation, echoing, noise reduction, fading, and sample rate conversion. It also offers recording and CD track ripping. It's small (320 kb), free, and has a decent manual.

Needless to say, there are many tools out there: do a search for "audio editor" at Google, or visit a software site such as tucows.

8.2. Byte Array Manipulation

The most versatile manipulation approach in Java (but potentially very tricky to get right) is to load the audio file as a byte array. Audio effects then become a matter of changing byte values, rearranging blocks of data, or perhaps adding new data. Once completed, the resulting array can be passed through a SourceDataLine into the mixer. The EchoSamplesPlayer.java application below shows how this can be done.

A variant of this approach is to employ streaming. Instead of reading in the entire file as a (very large) byte array, it can be incrementally read, changed, and sent on to the mixer. This coding style is possible for effects which only make localized changes, such as amplification or fading.

8.2.1. EchoSamplesPlayer.java

EchoSamplesPlayer plays the specified clip, adding a series of echoes to its end.

The entire sound file is read in as a byte array via an AudioInputStream. The echo is applied by creating a new byte array, and adding five copies of the original sound to it; each copy is less loud than the one before it. The resulting array is passed in small chunks to the SourceDataLine, and so to the mixer.

EchoSamplesPlayer is an extended version of the BufferedPlayer application described in section 5.3. The main change is to play(), the main addition is a getSamples() method: it applies the effect implemented in echoSamples(). There is also an isRequiredFormat() method for checking that the input is suitable for modification. The program is stored in SoundPlayer/.

To simplify the implementation, the effect is only applied to 8-bit PCM signed or unsigned audio. The choice of PCM means that the amplitude information is stored unchanged in the byte, not compressed as in the ULAW or ALAW formats. The 8-bit requirement means that a single byte is used per sample, so we don't have to deal with big- or little-endian issues. PCM unsigned data stores values between 0 and 2^8-1 (255), while the signed range is $–2^7$ to 2^7-1 (-128 to 127). This isn't really an issue except when we cast a byte into a short prior to changing it.

The main() method in EchoSamplesPlayer is similar to the one in BufferedPlayer.

```
public static void main(String[] args)
{ if (args.length != 1) {
    System.out.println("Usage: java EchoSamplesPlayer <clip>");
    System.exit(0);
  }

  createInput("Sounds/" + args[0]);

  if (!isRequiredFormat()) {    // not in SamplesPlayer
    System.out.println("Format unsuitable for echoing");
    System.exit(0);
  }

  createOutput();

 int numBytes=(int)(stream.getFrameLength()*format.getFrameSize());
  System.out.println("Size in bytes: " + numBytes);

  byte[] samples = getSamples(numBytes);
  play(samples);

  System.exit(0);   // necessary in J2SE 1.4.2 and earlier
}
```

The createInput() and createOutput() methods are unchanged from BufferedPlayer.

isRequiredFormat() tests the AudioFormat object that was created in createInput():

```
private static boolean isRequiredFormat()
// Only 8-bit PCM signed or unsigned audio can be echoed
{
  if (((format.getEncoding()==AudioFormat.Encoding.PCM_UNSIGNED) ||
       (format.getEncoding() == AudioFormat.Encoding.PCM_SIGNED))&&
       (format.getSampleSizeInBits() == 8))
    return true;
  else
    return false;
}
```

AudioFormat has a wide selection of get() methods for examining different aspects of the audio data. For example, getChannels() returns the number of channels used (1 for mono, 2 for stereo). The echoing effect doesn't need this information: all the frames, independent of the number of channels, will be amplified.

getSamples() adds the echoes after it has extracted the complete samples[] array from the AudioInputStream:

```
private static byte[] getSamples(int numBytes)
{
   // read the entire stream into samples[]
   byte[] samples = new byte[numBytes];
   DataInputStream dis = new DataInputStream(stream);
   try {
     dis.readFully(samples);
   }
   catch (IOException e)
   { System.out.println( e.getMessage());
     System.exit(0);
   }
   return echoSamples(samples, numBytes);
}
```

echoSamples() returns a modified byte array, which becomes the result of getSamples (). Different audio effects would replace echoSamples() at this point in the code.

echoSamples() creates a new byte array, newSamples(), big enough to hold the original sound and ECHO_NUMBER (4) copies. The volume of each one is reduced (decayed) by DECAY (0.5) over its predecessor.

```
private static byte[] echoSamples(byte[] samples, int numBytes)
{
  int numTimes = ECHO_NUMBER + 1;
  double currDecay = 1.0;
  short sample, newSample;
  byte[] newSamples = new byte[numBytes*numTimes];

  for (int j=0; j < numTimes; j++) {
    for (int i=0; i < numBytes; i++)  // copy the sound's bytes
      newSamples[i + (numBytes*j)] =
                    echoSample(samples[i], currDecay);
    currDecay *= DECAY;
  }
  return newSamples;
  }
```

The nested for-loop makes the required copies a byte at a time. echoSample() utilizes a byte in the original data to create an 'echoed' byte for newSamples[]. The amount of echoing is determined by the currDecay double, which gets smaller for each successive copy of the original sound.

echoSample() does slightly different tasks depending on whether the input data is unsigned or signed PCM. In both cases, the supplied byte is translated into a short so it can be manipulated easily, then the result is converted back to a byte.

```
  private static byte echoSample(byte sampleByte, double currDecay)
  {
    short sample, newSample;
    if (format.getEncoding() == AudioFormat.Encoding.PCM_UNSIGNED) {
      sample = (short)(sampleByte & 0xff);  //unsigned 8 bit -> short
      newSample = (short)(sample * currDecay);
      return (byte) newSample;
    }
    else if (format.getEncoding()==AudioFormat.Encoding.PCM_SIGNED){
      sample = (short)sampleByte;   // signed 8 bit --> short
      newSample = (short)(sample * currDecay);
      return (byte) newSample;
    }
    else
      return sampleByte;    //no change; this branch should be unused
  }
```

The byte-to-short conversion must be done carefully. An unsigned byte needs masking as it's converted since Java stores shorts in signed form. A short is two bytes long, and so the masking ensures that the bits in the high-order byte are all set to 0's. Without the mask, the conversion would add in 1's when it saw a byte value above 127.

No masking is required for the signed byte to signed short conversion, since the translation is correct by default.

play() is quite similar to the one in BufferedPlayer.java. The difference is that the byte array must be passed through an input stream before it can be sent to the SourceDataLine.

```
  private static void play(byte[] samples)
  {
    // byte array --> stream
    InputStream source = new ByteArrayInputStream(samples);

    int numRead = 0;
    byte[] buf = new byte[line.getBufferSize()];

    line.start();
    // read and play chunks of the audio
    try {
      while ((numRead = source.read(buf, 0, buf.length)) >= 0) {
        int offset = 0;
        while (offset < numRead)
```

```
          offset += line.write(buf, offset, numRead-offset);
      }
    }
    catch (IOException e)
    {  System.out.println( e.getMessage()); }

    // wait until all data is played, then close the line
    line.drain();
    line.stop();
    line.close();
  }  // end of play()
```

8.3. Utilizing Mixer Controls

The mixer diagram (Figure 5) includes a grayish box labeled "Controls". These can be accessed through Clip or SourceDataLine to apply a *limited* set of effects to the input sound. The effects may include volume control, panning between speakers, reverberation, and sample rate control.

The bad news is that the default mixer in J2SE 1.5 offers even less controls than were present in J2SE 1.4.2, since controls tend to have an adverse effect on speed, even when they're not being used. However, if a control is present, then it's much easier to apply than the byte array technique.

8.3.1. PlaceClip

PlaceClip plays a clip, allowing its volume and pan settings to be adjusted via command line parameters. It is called with the following format:

```
java PlaceClip <clip file> [ <volume value> [<pan value>] ]
```

The volume and pan values are optional; if they are both left out then the clip plays normally.

The volume setting should be between 0.0f and 1.0f (the loudest); -1.0f means that the volume is left unchanged.

The pan value should be between -1.0f and 1.0f; -1.0f causes all the sound to be set to the left speaker, 1.0f focuses only on the right speaker, and values in between will send the sound to both speakers with varying weights.

For example:

```
> java PlaceClip dog.wav 0.8f -1.0f
```

will make the left speaker bark loudly.

This mixing of volume and speaker placement is a rudimentary way of placing sounds at different 'locations' in a game.

PlaceClip is an extended version of PlayClip described in section 5.2. The changes are in the extra methods for reading the volume and pan settings from the command line, and the setVolume() and setPan() methods for adjusting the clip controls.

PlaceClip's main() method is similar to the one in PlayClip.java.

```
// globals
private float volume, pan;   // settings from the command line
    :

public PlaceClip(String[] args)
{
  df = new DecimalFormat("0.#");  // 1 dp

  getSettings(args);   // get the volume and pan settings
                       // from the command line
  loadClip(SOUND_DIR + args[0]);

  // clip control methods
  showControls();
  setVolume(volume);
  setPan(pan);

  play();
  try {
    Thread.sleep(600000);   // 10 mins in ms
  }
  catch(InterruptedException e)
  { System.out.println("Sleep Interrupted"); }
}
```

loadClip() and play() are virtually unchanged from PlayClip (loadClip() uses a globally defined AudioFormat variable, and has some extra println()'s).

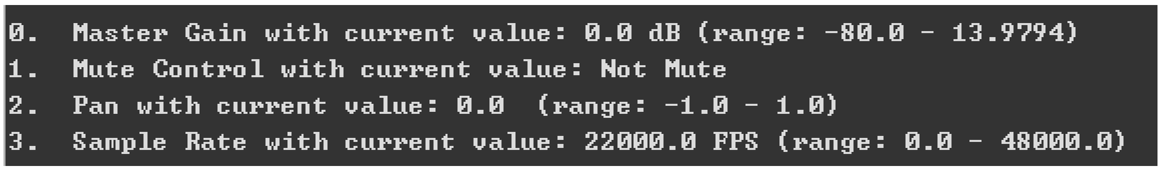
8.3.2. What Controls are Available?

showControls() displays all the controls available for the clip, which will vary depending on the clip's audio format and the mixer.

```
private void showControls()
{ if (clip != null) {
    Control cntls[] = clip.getControls();
    for(int i=0; i<cntls.length; i++)
      System.out.println( i + ".  " + cntls[i].toString() );
  }
}
```

getControls() only returns information once its clip has been opened.

For the dog.wav example, executed using the J2SE 1.4.2 default mixer, showControls ()'s output is given in Figure 12.



```
0.  Master Gain with current value: 0.0 dB (range: -80.0 - 13.9794)
1.  Mute Control with current value: Not Mute
2.  Pan with current value: 0.0  (range: -1.0 - 1.0)
3.  Sample Rate with current value: 22000.0 FPS (range: 0.0 - 48000.0)
```

Figure 12. showControls()'s Output.

Four controls are available: gain (volume), mute, panning, and sample rate.

Reverberation and balance controls, may also be shown for some types of clips and mixers. In J2SE 1.5, panning, sample rate, and reverberation are no longer supported, and the balance control is only available for audio files using stereo.

Out in the real world, a pan control distributes *mono input* between stereo output lines (e.g. the lines going to the speakers). A balance control does a similar job, but for *stereo input*.

In J2SE 1.4.2 and before, the pan and balance controls could be used with mono or stereo input – there was no distinction made between them. Also, output lines were always opened in stereo mode. The default J2SE 1.4.2 mixer is the Java Sound Audio Engine.

The default mixer in J2SE 1.5 is the Direct Audio Device, with resulting changes to the controls. If the mixer receives mono input it will open a *mono* output line, not a stereo one. This means there's no pan control, since there's no way to map mono to stereo. There is a balance control, but that's for mapping stereo input to stereo output.

In J2SE 1.5, our example will report that panning is unavailable, since dog.wav was recorded in mono. The simplest solution is to convert it to stereo using WavePad (http://nch.com.au/wavepad/), or similar software. The balance controls will then be available, and setPan() can carry out 'panning' by adjusting the balance.

8.3.3. Java Audio Controls

The various controls are represented by subclasses of the Control class: BooleanControl, FloatControl, EnumControl, and CompoundControl.

BooleanControl is used to adjust binary settings, such as mute on/off. FloatControl is employed for controls that range over floating point values, such as volume, panning, and balance. EnumControl permits a choice between several settings, as in reverberation. CompoundControl groups controls. These controls will only function if the clip is open.

A code fragment that turns mute on and off with a BooleanControl:

```
BooleanControl muteControl =
     (BooleanControl) clip.getControl( BooleanControl.Type.MUTE );
muteControl.setValue(true);     // mute on; sound is switched off
     :
muteControl.setValue(false);    // mute off; sound is audible again
```

Play a clip at 1.5 times its normal speed via a FloatControl:

```
FloatControl rateControl =
   (FloatControl) clip.getControl( FloatControl.Type.SAMPLE_RATE );
rateControl.setValue( 1.5f * format.getSampleRate() );
        // format is the AudioFormat object for the audio file
```

8.3.4. Setting the Volume in PlaceClip

In PlaceClip, the user's volume setting is between 0.0f and 1.0f (loudest), or is the value NO_VOL_CHANGE (-1.0f), and needs to be mapped to a gain setting in decibels.

The decibel scale is logarithmic, and related to the square of the distance from the sound source, which makes the conversion somewhat tricky.

We take a simpler approach: mapping the user's volume setting to a linear scale between the control's minimum and maximum values. The drawback is that the sound becomes inaudible when the setting drops below about 0.5f.

```
  private void setVolume(float volume)
  {
    if ((clip != null) && (volume != NO_VOL_CHANGE)) {
      if (clip.isControlSupported(FloatControl.Type.MASTER_GAIN)) {
        FloatControl gainControl = (FloatControl)
              clip.getControl(FloatControl.Type.MASTER_GAIN);

        float range = gainControl.getMaximum() -
                                     gainControl.getMinimum();
        float gain = (range * volume) + gainControl.getMinimum();
        System.out.println("Volume: "+volume+"; New gain: " + gain);
        gainControl.setValue(gain);
      }
      else
        System.out.println("No Volume controls available");
    }
  }
```

The code is more robust than the fragments given above, since it uses isControlSupported() before attempting to access/change the control setting.

FloatControl have several useful methods, including shift() which changes the control value gradually, over a specified time period. The method returns without waiting for the shift to finish. Unfortunately, it has never been fully implemented, and currently modifies the control value in one step.

8.3.5. Panning between the Speakers in PlaceClip

setPan() is supplied with a pan value between -1.0f and 1.0f (which positions the output somewhere between the left and right speakers), or with NO_PAN_CHANGE (0.0f). The method tries panning first, then looks for the balance control if panning is unavailable, then gives up if balancing is unsupported.

```
  private void setPan(float pan)
  {
    if ((clip == null) || (pan == NO_PAN_CHANGE))
      return;   // do nothing

    if (clip.isControlSupported(FloatControl.Type.PAN)) {
      FloatControl panControl =
         (FloatControl) clip.getControl(FloatControl.Type.PAN);
      panControl.setValue(pan);
    }
    else if (clip.isControlSupported(FloatControl.Type.BALANCE)) {
      FloatControl balControl =
         (FloatControl) clip.getControl(FloatControl.Type.BALANCE);
      balControl.setValue(pan);
    }
    else {
      System.out.println("No Pan or Balance controls available");
      if (format.getChannels() == 1)   // mono input
        System.out.println("Your audio file is mono;
                                 try converting it to stereo");
    }
  }
```

9. Audio Effects on MIDI Sequences

There are several ways of applying audio effects to MIDI sequences:

1. Pre-calculation;
2. Sequence Manipulation;
3. MIDI Channel Controllers;
4. Sequencer Methods.

9.1. Pre-calculation

As with sampled audio, using Java at execution time to modify a sequence can be time-consuming, and difficult to implement. There are several tools which create or edit MIDI sequences, although you do need an understanding of music and MIDI to use them. Two packages that I've tinkered with:

- the free version of Anvil Studio (http://www.anvilstudio.com/) which supports the capture, editing, and direct composing of MIDI. It also handles WAV files.
- BRELS MIDI Editor, a free, small MIDI editor. It is easiest to obtain it from a software site, such as tucows.
- Midi Maker (http://www.necrocosm.com/midimaker/). The program emulates a standard keyboard synthesizer. Available for 14 days trial.

9.2. Sequence Manipulation

Figure 12 (which is a repeat of Figure 8) shows the internals of a sequence.

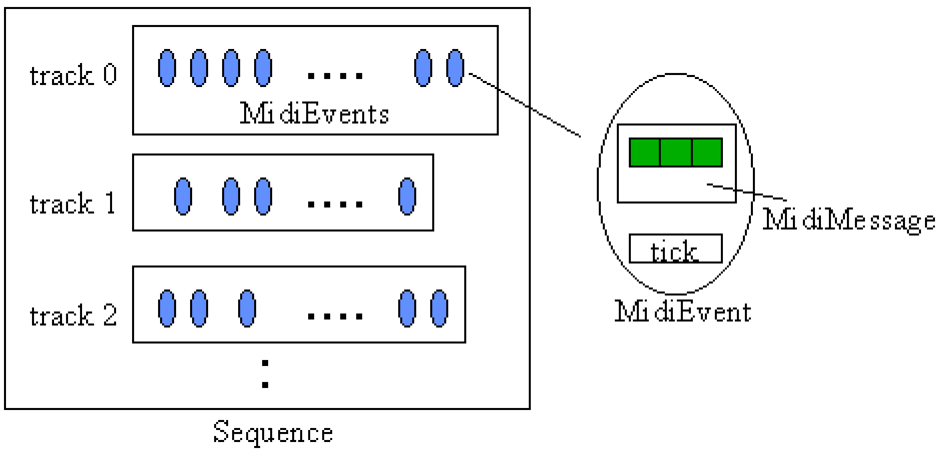


Figure 12. The Internals of a MIDI Sequence.

The regularity of the data structure means that it's not too difficult to modify at run time. The downside is the need to understand the MIDI specification.

9.2.1. Doubling the Sequence Volume

The basic code for playing a sequence is shown below, without the necessary try-catch blocks. Look at PlayMidi.java (section 6.2) for a complete version.

```
Sequence seq = MidiSystem.getSequence(getClass().getResource(fnm));

// change the sequence: double its volume in this case
doubleVolumeSeq(seq);

sequencer.setSequence(seq);  // load changed sequence
sequencer.start();           // start playing it
```

The sequence is modified after being loaded with getSequence(), before being assigned to the sequencer with setSequence().

Volume doubling is applied to every track in the sequence:

```
private void doubleVolumeSeq(Sequence seq)
{ Track tracks[] = seq.getTracks();     // get all the tracks
  for(int i=0; i < tracks.length; i++)  // iterate through them
    doubleVolume(tracks[i], tracks[i].size());
}
```

doubleVolume() examines every MidiEvent in the supplied track, extracting its component tick and MIDI message. If the message is a NOTE_ON, then its volume is doubled (up to a maximum of 127).

```
private void doubleVolume(Track track, int size)
{
  MidiEvent event;
  MidiMessage message;
  ShortMessage sMessage, newShort;

  for (int i=0; i < size; i++) {
    event = track.get(i);          // get the event
    message = event.getMessage();  // get its MIDI message
    long tick = event.getTick();   // get its tick
    if (message instanceof ShortMessage) {
      sMessage = (ShortMessage) message;

      // check if the message is a NOTE_ON
      if (sMessage.getCommand() == ShortMessage.NOTE_ON) {
         int doubleVol = sMessage.getData2() * 2;
         int newVol = (doubleVol > 127) ? 127 : doubleVol;
         newShort = new ShortMessage();
         try {
           newShort.setMessage(ShortMessage.NOTE_ON,
                               sMessage.getChannel(),
                               sMessage.getData1(), newVol);
           track.remove(event);
```

```
            track.add( new MidiEvent(newShort,tick) );
          }
          catch ( InvalidMidiDataException e)
          {  System.out.println("Invalid data");  }
        }
      }
    }
  }  // end of doubleVolume()
```

Each MIDI message is composed from three bytes: a command name and two data bytes. getCommand() is employed to check the name. If it's NOTE_ON, then the first byte is the note number, and the second its velocity (similar to a volume level).

The volume is obtained with a call to getData2(), then doubled, but with a ceiling of 127 since the number must fit back into a single byte.

A new ShortMessage object is constructed, and filled with relevant details (command name, destination channel ID, note number, new volume):

```
  newShort.setMessage(ShortMessage.NOTE_ON,
               sMessage.getChannel(), sMessage.getData1(), newVol);
```

The old MIDI event (containing the original message) must now be replaced by an event holding the new message: a two-step process involving remove() and add(). The new event is built from the new message and the old tick value:

```
  track.add( new MidiEvent(newShort,tick) );
```

The tick specifies where the event will be placed in the track.

9.3. MIDI Channel Controllers

Figure 7 shows the presence of 16 MIDI channels inside the synthesizer; each one a 'musician' playing a particular instrument. As the stream of MIDI messages arrive (either individually or as part of a sequence), each message is routed to a channel based on its channel setting.

Each channel has a set of controllers associated with it. The set depends on the particular synthesizer, although controllers defined in the General MIDI specification should be present, but there may be others. For example, controllers offering the Roland GS enhancements are found on many devices.

General MIDI controllers include controls for volume level, stereo balancing, and panning. Popular Roland GS enhancements include reverberation and chorus effects.

A list of channel controllers, complete with a short description of each one, can be found at http://improv.sapp.org/doc/class/MidiOutput/controllers/. Another site with similar information is http://improv.sapp.org/doc/class/MidiOutput/controllers/.

Our FadeMidi and PanMidi examples illustrate how to use channel controllers to affect the playback of an existing sequence. They reuse many methods from PlayMidi.java in section 6.2.

9.3.1. Making a Sequence Fade Away

FadeMidi.java (located in SoundPlayer/) plays a sequence, gradually reducing its volume level to 0 by the end. The volume settings for all 16 channels are manipulated by accessing each channel's main volume controller (number 7). There is also a fine grain volume controller (number 39) which should allow smaller change graduations, but many synthesizers don't support it.

The incremental volume reduction is managed by a VolChanger thread, which repeatedly lowers the volume reduction until the sequence has been played to its end.

Figure 13 gives the UML class diagrams for FadeMidi and VolChanger, showing only the public methods.

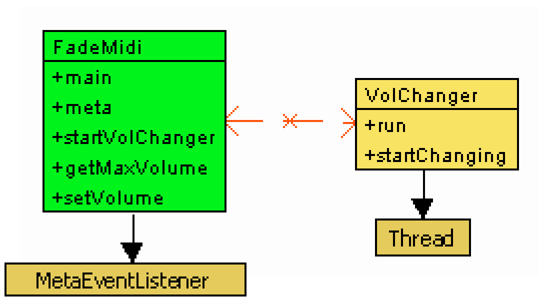


Figure 13. Class Diagrams for FadeMidi and VolChanger.

The main() method initializes FadeMidi and starts VolChanger:

```
public static void main(String[] args)
{ if (args.length != 1) {
    System.out.println("Usage: java FadeMidi <midi file>");
    System.exit(0);
  }

  // set up the player and the volume changer
  FadeMidi player = new FadeMidi(args[0]);
  VolChanger vc = new VolChanger(player);

  player.startVolChanger(vc);  // start volume manipulation
}
```

VolChanger is passed a reference to FadeMidi so that it can affect the synthesizer's volume settings.

startVolChanger() start the VolChanger thread running, and supplies the sequence duration in milliseconds. The thread needs it to calculate how often to change the volume.

```
public void startVolChanger(VolChanger vc)
{  vc.startChanging( (int)(seq.getMicrosecondLength()/1000) );  }
```

The FadeMidi constructor looks similar to the one in PlayMidi:

```
public FadeMidi(String fnm)
{
 df = new DecimalFormat("0.#");  // 1 dp
  filename = SOUND_DIR + fnm;
  initSequencer();
  loadMidi(filename);
  play();

  /* No need for sleeping to keep the object alive, since
     the VolChanger thread refers to it. */

}  // end of FadeMidi()
```

initSequencer() and loadMidi() are identical to the same named methods in PlayClip, and play() only slightly different. The most significant change is the absence of a call to sleep(), which keeps PlayMidi alive until its sequence has finished.

Sleeping is unnecessary in FadeMidi since the object is referred to by the VolChanger thread, which keeps calling its setVolume() method.

play() initializes a global array of MIDI channels.

```
private static final int VOLUME_CONTROLLER = 7;
     :
// global holding the synthesizer's channels
private MidiChannel[] channels;
     :

private void play()
{ if ((sequencer != null) && (seq != null)) {
    try {
      sequencer.setSequence(seq);  // load MIDI into sequencer
      sequencer.start();   // play it
      channels = synthesizer.getChannels();
      // showChannelVolumes();
    }
    catch (InvalidMidiDataException e) {
      System.out.println("Invalid midi file: " + filename);
      System.exit(0);
    }
  }
}

private void showChannelVolumes()
// show the volume levels for all the synthesizer channels
{
  System.out.println("Syntheziser Channels: " + channels.length);
  System.out.print("Volumes: {");
  for (int i=0; i < channels.length; i++)
    System.out.print( channels[i].getController(VOLUME_CONTROLLER)
                                                + " ");
  System.out.println("}");
}
```

The references to the channels should not be obtained until the sequence is playing (i.e. after calling sequencer.start()) or their controllers will not respond to changes. This seems to be a bug in the Java Sound implementation.

Channels in the array are accessed using the indices 0 to 15, although the MIDI specification numbers them 1 to 16. For instance, the special percussion channel is MIDI number 10, but is channels[9] in Java.

In showChannelVolumes(), getController() returns the current volume setting after being supplied with the controller number 7. A controller stores it data in a single byte, so its value will be in the range 0 to 127.

FadeMidi contains two public methods for getting and setting the volume, both used by VolChanger.

```
  public int getMaxVolume()
  // return the max level for all the volume controllers
  { int maxVol = 0;
    int channelVol;
    for (int i=0; i < channels.length; i++) {
      channelVol = channels[i].getController(VOLUME_CONTROLLER);
      if (maxVol < channelVol)
        maxVol = channelVol;
    }
    return maxVol;
 }

  public void setVolume(int vol)
  // set all the controller's volume levels to vol
  { for (int i=0; i < channels.length; i++)
      channels[i].controlChange(VOLUME_CONTROLLER, vol);
  }
```

getMaxVolume() returns a single volume, rather than all 16; this keeps the code simple.

setVolume() shows how controlChange() is used to change a specified controller's value. The data should be an integer between 0 and 127.

VolChanger

VolChanger gets started when its startChanging() method is called. At this point, the sequence will be playing, and the MIDI channel controllers available for manipulation.

```
  // globals
  // the amount of time between changes to the volume, in ms
  private static int PERIOD = 500;

  private FadeMidi player;
  private int numChanges = 0;
          :

  public void startChanging(int duration)
  /* FadeMidi calls this method, supplying the duration of
```

```
     its sequence in ms. */
  {
    // calculate how many times the volume should be adjusted
    numChanges = (int) duration/PERIOD;
    start();
  } // end of startChanging()
```

VolChanger adjusts the volume every PERIOD (500) ms, but how many times? The duration of the sequence is passed in as an argument to startChanging(), and used to calculate the number of volume changes.

run() implements a volume reduction / sleep cycle.

```
  public void run()
  {
     /* calculate stepVolume, the amount to decrease the volume
        each time that the volume is changed. */
     int volume = player.getMaxVolume();
     int stepVolume = (int) volume / numChanges;
     if (stepVolume == 0)
       stepVolume = 1;
     System.out.println("Max Volume: " + volume +
                                     ", step: " + stepVolume);
     int counter = 0;
     System.out.print("Fading");
     while(counter < numChanges){
       try {
         volume -= stepVolume;    // reduce the required volume level
         if ((volume >= 0) && (player != null))
           player.setVolume(volume);    // change the volume
         Thread.sleep(PERIOD);          // delay a while
       }
       catch(InterruptedException e) {}
       System.out.print(".");
       counter++;
     }
     System.out.println();
  }
```

9.3.2. PanMidi

PanMidi repeatedly switching its sequence from the left speaker to the right, and back again. A PanChanger thread switches the pan settings in all the channel controllers at periodic intervals during the playing of the sequence. PanMidi and PanChanger can be found in SoundPlayer/.

The class diagrams for PanMidi and PanChanger are given in Figure 14.

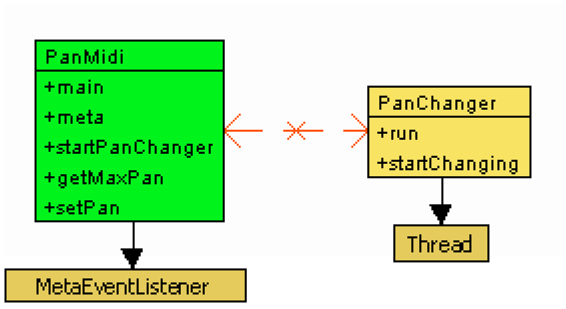


Figure 14. Class Diagrams for PanMidi and PanChanger.

The main() method initializes the player and the thread, and then calls PanMidi's startPanChanger() to start the thread running. startPanChanger() passes the duration of the sequence to the thread, so it can calculate the number of changes it will make.

The PanMidi pan methods used by PanChanger are getMaxPan() and setPan():

```
// global constants
// private static final int BALANCE_CONTROLLER = 8; //not working?
   private static final int PAN_CONTROLLER = 10;
            :

  public int getMaxPan()
  // return the max value for all the pan controllers
  { int maxPan = 0;
    int channelPan;
    for (int i=0; i < channels.length; i++) {
      channelPan = channels[i].getController(PAN_CONTROLLER);
      if (maxPan < channelPan)
        maxPan = channelPan;
    }
    return maxPan;
  }

  public void setPan(int panVal)
  // set all the controller's pan levels to panVal
  { for (int i=0; i < channels.length; i++)
      channels[i].controlChange(PAN_CONTROLLER, panVal);
  }
```

The only real difference from FadeMidi is the use of the PAN_CONTROLLER controller number. The balance controller should also work in this situation, but didn't on my test machines.

PanChanger

Unlike VolChanger, PanChanger carries out a cyclic series of changes to the pan value. However, the core of run() is still a loop repeatedly calling setPan() and sleeping for an interval.

The series of pan values that make up a single cycle are defined in a panVals[] array.

```
// time to move left to right and back again
private static int CYCLE_PERIOD = 4000;  // in ms

// pan values used in a single cycle
// (make the array's length integer divisible into CYCLE_PERIOD)
private int[] panVals = {0, 127};

// or try
// private int[] panVals = {0, 16, 32, 48, 64, 80, 96, 112, 127,
//                          112, 96, 80, 64, 48, 32, 16};
```

The run() method cycles through the panVals[] array until it has executed for a time equal to the sequence's duration.

```
public void run()
{ /* Get the original pan setting, just for information. It
      is not used any further. */
   int pan = player.getMaxPan();
   System.out.println("Max Pan: " + pan);

   int panValsIdx = 0;
   int timeCount = 0;
   int delayPeriod = (int) (CYCLE_PERIOD / panVals.length);

   System.out.print("Panning");
   while(timeCount < duration){
     try {
       if (player != null)
         player.setPan( panVals[panValsIdx] );
       Thread.sleep(delayPeriod);    // delay
     }
     catch(InterruptedException e) {}
     System.out.print(".");
     panValsIdx = (panValsIdx+1) % panVals.length;
                                  // cycle through the array
     timeCount += delayPeriod;
   }
   System.out.println();
}
```

9.4. Sequencer Methods

The Sequencer has methods which can change the tempo (speed) of playback. The easiest to use is probably setTempoFactor() which scales the existing tempo by the supplied float:

```
sequencer.setTempoFactor(2.0f);   // double the tempo
```

Tempo adjustments only work if the sequence's event ticks are defined in the PPQ (ticks per beat) format, since tempo affects the number of beats per minute. Use getTempoFactor() to check whether the requested change has occurred.

There are also methods which act upon the sequence's tracks: setTrackMute(), setTrackSolo(). A fragment of code, which sets and tests the mute value.

```
sequencer.setTrackMute(4, true);
boolean muted = sequencer.getTrackMute(4);
if (!muted)
  // muting failed
```

10. Sampled Audio Synthesis

Sampled audio is encoded as a series of samples in a byte array, which is sent through a SourceDataLine to the mixer. In previous examples, the contents of the byte array came from an audio file, although we saw that audio effects can manipulate, and even add, to the array.

In sampled audio synthesis, the application generates the byte array data itself, without requiring any audio input. Potentially any sound can be generated at run-time.

Audio is a mix of sine waves, each one representing a tone or note. A pure note is a single sine wave with a fixed amplitude and frequency (or pitch). Frequency can be defined as the number of sine waves which pass a given point in a second. The higher the frequency, the higher the note's pitch. The higher the amplitude, the louder the note.

Before we go further, it helps to introduce the usual naming scheme for notes; it's easier to talk about note names than note frequencies.

10.1. Note Names

Notes names are derived from the piano keyboard, which has a mix of black and white keys, as in Figure 15.

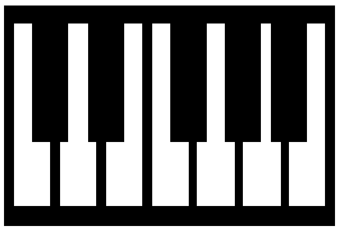


Figure 15. Part of the Piano KeyBoard.

Keys are grouped into octaves, each octave consisting of twelve consecutive white and black keys. The white keys are labeled with the letters 'A' to 'G' and an octave number. For example, the note named C4 is the white key closest to the center of the keyboard, often referred to as "middle C". The '4' means that the key is in the fourth octave, counting from the left of the keyboard.

A black key is labeled with the letter of the preceding white key and a sharp ('#'). For instance, the black key following C4 is known as C#4. (We'll ignore flats in this discussion.)

Figure 16 shows the keyboard fragment of Figure 15 again, but labeled with note names. We've assumed that the first white key is C4.

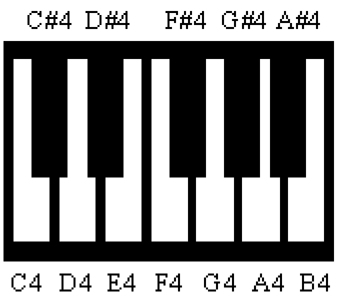


Figure 16. Piano Keyboard with Note Names.

Figure 16 utilizes the C Major scale, where the letters appear in the order C, D, E, F, G, A, and B. There is a Minor scale which starts at A, but we'll not be using it.

After B4, the fifth octave begins, starting with C5 and repeating the same sequence as in the fourth octave. Before C4, is the third octave which ends with B3.

Having introduced note names, we can start talking about their associated frequencies or pitches. Table 1 gives the frequencies for the C4 Major scale (the notes from C4 to B4).

Note Name	Frequency (in Hz)
C4	261.63
C#4	277.18
D4	293.66
D#4	311.13
E4	329.63
F4	349.23
F#4	369.99
G4	392.00
G#4	415.30
A4	440.00
A#4	466.16
B4	493.88

Table 1. Frequencies for the C4 Major Scale.

When we move to the next octave, the frequencies double for all the notes, for instance, C5 will be 523.26 Hz. The preceding octave contains frequencies that are halved, so C3 will be 130.82 Hz.

A table showing *all* piano note names and their frequencies can be found at http://www.phys.unsw.edu.au/~jw/notes.html. It also includes the corresponding MIDI numbers, which we consider in the section 11.

10.2. Playing a Note

A note can be played by generating its associated frequency, with an amplitude for loudness. But how can this approach be implemented in terms of a byte array suitable for a SourceDataLine?

A pure note is a single sine wave, with a specified amplitude and frequency, and this sine wave can be represented by a series of samples stored in a byte array. The idea is shown in Figure 17.

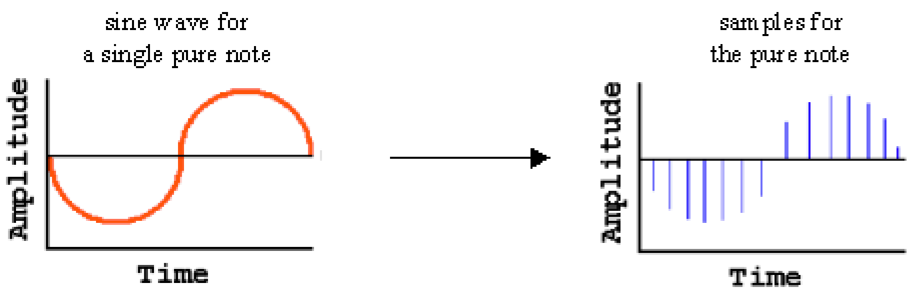


Figure 17. From Single Note to Samples.

This is just a simple form of analog-to-digital conversion. The question remains: how is the frequency converted into a given number of samples (i.e. how many blue lines should the sample contain)?

The SourceDataLine is set up to accept a specified audio format, which includes a sample rate. For example, a sample rate of 21,000 causes 21,000 samples to reach the mixer every second. The frequency of a note, e.g. 300 Hz, means that 300 copies of that note will reach the mixer per second.

The number of samples required to represent a single note is:

samples/note = (samples/second) / (notes/sec)

or samples/note = sample rate / frequency

For the example above, a single note would need 21,000/300 = 70 samples. In other words, the sine wave must consist of 70 samples. This approach is implemented in sendNote() in the NotesSynth.java application explained next.

10.3. NotesSynth

NotesSynth generates simple sounds at run time without playing a clip. The current version outputs an increasing pitch sequence, repeated 9 times, each time increasing a bit faster, and with decreasing volume. NotesSynth.java is stored in SynthSound/.

The main() method:

```
public static void main(String[] args)
{ createOutput();
  play();
  System.exit(0);    // necessary for J2SE 1.4.2 or earlier
}
```

createOutput() opens a SourceDataLine that accepts stereo, signed PCM audio, utilizing 16 bits per sample in little endian format. Consequently, four bytes must be used for each sample.

```
// globals
private static int SAMPLE_RATE = 22050;     // no. of samples/sec

private static AudioFormat format = null;
private static SourceDataLine line = null;
         :


private static void createOutput()
{
  format =  new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,
                      SAMPLE_RATE, 16, 2, 4, SAMPLE_RATE, false);
 /*  SAMPLE_RATE   // samples/sec
     16        // sample size in bits, values can be -2^15 - 2^15-1
     2         // no. of channels, stereo here
     4         // frame size in bytes (2 bytes/sample * 2 channels)
     SAMPLE_RATE   // same as frames/sec
     false         // little endian     */

  System.out.println("Audio format: " + format);

  try {
    DataLine.Info info =
          new DataLine.Info(SourceDataLine.class, format);
    if (!AudioSystem.isLineSupported(info)) {
      System.out.println("Line does not support: " + format);
      System.exit(0);
    }
    line = (SourceDataLine) AudioSystem.getLine(info);
    line.open(format);
  }
  catch (Exception e)
  {  System.out.println( e.getMessage());
     System.exit(0);
  }
}  // end of createOutput()
```

play() makes a buffer large enough for the samples, plays the pitch sequence using sendNote(), then closes the line.

```
private static void play()
{
  // calculate a size for the byte buffer holding a note
  int maxSize = (int) Math.round(
                (SAMPLE_RATE * format.getFrameSize())/MIN_FREQ);
                     // the frame size is 4 bytes
  byte[] samples = new byte[maxSize];

  line.start();

  /* Generate an increasing pitch sequence, repeated 9 times, each
     time increasing a bit faster, and the volume decreasing */
  double volume;
  for (int step = 1; step < 10; step++)
```

```
      for (int freq = MIN_FREQ; freq < MAX_FREQ; freq += step) {
         volume = 1.0 - (step/10.0);
         sendNote(freq, volume, samples);
      }

    // wait until all data is played, then close the line
    line.drain();
    line.stop();
    line.close();
  } // end of play()
```

maxSize must be big enough to store the largest number of samples for a generated note, which occurs when the note frequency is the smallest. Therefore, the MIN_FREQ value (250 Hz) is divided into SAMPLE_RATE.

sendNote() translates a frequency and amplitude into a series of samples representing that note's sine wave. The samples are stored in a byte array, then sent along the SourceDataLine to the mixer.

```
  // globals
  private static double MAX_AMPLITUDE = 32760;     // max loudness
              // actual max is 2^15-1, 32767, since we are using
              // PCM signed 16 bit

  // frequence (pitch) range for the notes
  private static int MIN_FREQ = 250;
  private static int MAX_FREQ = 2000;
    // Middle C (C4) has a frequency of 261.63 Hz; see table 1
          :


  private static void sendNote(int freq, double volLevel,
                                          byte[] samples)
  { if ((volLevel < 0.0) || (volLevel > 1.0)) {
      System.out.println("Volume level should be between
                                     0 and 1, using 0.9");
      volLevel = 0.9;
    }
    double amplitude = volLevel * MAX_AMPLITUDE;

    int numSamplesInWave =
              (int) Math.round( ((double) SAMPLE_RATE)/freq );
    int idx = 0;
    for (int i = 0; i < numSamplesInWave; i++) {
      double sine = Math.sin(((double) i/numSamplesInWave) *
                                            2.0 * Math.PI);
      int sample = (int) (sine * amplitude);
      // left sample of stereo
      samples[idx + 0] = (byte) (sample & 0xFF);          // low byte
      samples[idx + 1] = (byte) ((sample >> 8) & 0xFF);  // high byte
      // right sample of stereo (identical to left)
      samples[idx + 2] = (byte) (sample & 0xFF);
      samples[idx + 3] = (byte) ((sample >> 8) & 0xFF);
      idx += 4;
    }

    // send out the samples (the single note)
    int offset = 0;
    while (offset < idx)
      offset += line.write(samples, offset, idx-offset);
  }
```

numSamplesInWave is obtained by using the calculation described above: dividing the note frequency into the sample rate.

A sine wave value is obtained with Math.sin(), then split into two bytes since 16-bit samples are being used. The little endian format determines that the low-order byte is stored first, then the high-order one. Stereo means that we must supply two bytes for the left speaker, and two for the right. In our case, the data is the same for both.

NotesSynth.java could be extended in some interesting ways. We could allow the user to specify notes with note names (e.g. C4, F#6), and translate them into frequencies before calling sendNote().

play() is hardwired to output the same tones every time it's executed. It would be quite easy to have it read a 'notes files', perhaps written using note names, to play different tunes.

An important missing element is timing. Notes are played immediately after each other. It would be better to permit periods of silence as well.

11. MIDI Synthesis

We'll consider three approaches to synthesizing MIDI sound at run time:

1) Sending note-playing messages to a MIDI channel;
2) Sending MIDI messages to the synthesizer's receiver port;
3) Creating a sequence, which is passed to the sequencer.

These approaches are labeled in the MIDI devices diagram in Figure 18.

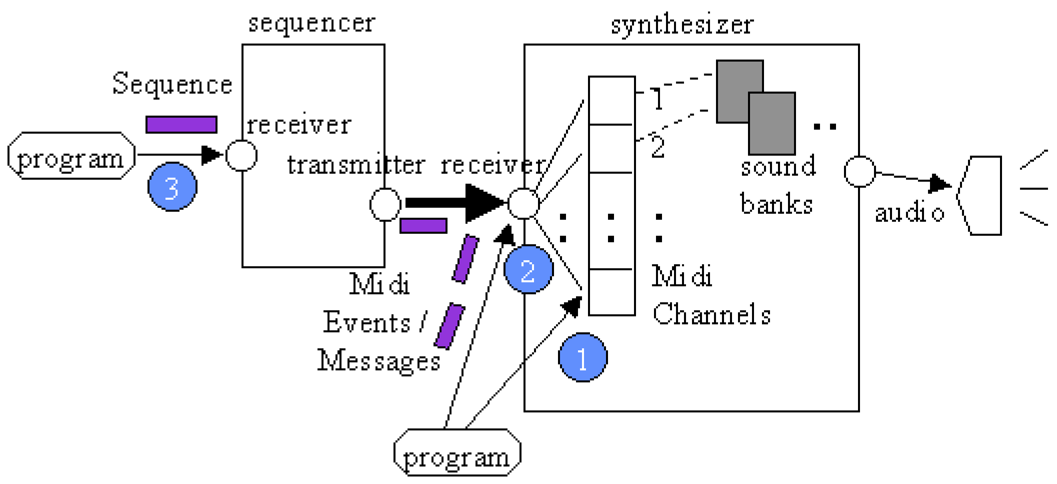


Figure 18. Different MIDI Synthesis Approaches.

There is a good *Java Tech Tip* on these topics, at http://java.sun.com/jdc/JDCTechTips/2003/tt0805.html.

11.1. Sending Note-Playing Message to a MIDI Channel

The MidiChannel class offers noteOn() and noteOff() methods which correspond to the NOTE_ON and NOTE_OFF MIDI messages:

```
      void noteOn(int noteNumber, int velocity);
      void noteOff(int noteNumber, int velocity);
or    void noteOff(int noteNumber);
```

The note number is the MIDI number assigned to a musical note, while velocity is roughly equivalent to the loudness.

A note will keep playing after a noteOn() call, until it is terminated with noteOff().

The two-argument form of noteOff() can affect how quickly the note fades away, but is often not implemented.

A table showing the correspondence between MIDI note numbers and note names can be found at http://www.phys.unsw.edu.au/~jw/notes.html.Table 2 shows the mapping for the 4th octave.

MIDI Number	Note Name
60	C4
61	C#4
62	D4
63	D#4
64	E4
65	F4
66	F#4
67	G4
68	G#4
69	A4
70	A#4
71	B4

Table 2. MIDI numbers and Note Names.

MIDI notes can range between 0 and 127, extending well beyond the piano's scope which only includes 88 standard keys. This means that the note naming scheme gets a little strange below note 12 (C0), since we have to start talking about octave -1 (e.g. see the table at http://www.harmony-central.com/MIDI/Doc/table2.html). Also a maximum value of 127 means that note names only go up to G9, there is no G#9.

A channel is obtained in the following way:

```
Synthesizer synthesizer = MidiSystem.getSynthesizer();
synthesizer.open();
MidiChannel drumChannel = synthesizer.getChannels()[9];
```

Channel 9 plays different percussion and audio effect sounds depending on the note numbers sent to it.

Playing a note corresponds to sending a NOTE_ON message, letting it play for a while, and then killing it with a NOTE_OFF message. This can be wrapped up in a playNote() method:

```
public void playNote(int note, int duration)
{
  drumChannel.noteOn(note, 70);  // 70 is the volume level
  try {
    Thread.sleep(duration*1000);   // secs --> ms
  }
  catch (InterruptedException e) {}
  drumChannel.noteOff(note);
}
```

The following will trigger applause:

```
for (int i=0; i < 10; i++)
  playNote(39, 1);  // 1 sec duration
```

A list of the mappings from MIDI numbers to drum sounds can be found at http://www.midi.org/about-midi/gm/gm1sound.shtml. Note 39 corresponds to a "hand clap" sound.

MidiChannel supports a range of useful methods aside from noteOn() and noteOff(), including setMute(), setSolo(), setOmni(), and setPitchBend(). The programChange() methods allow the channel's instrument to be changed, based on its bank and program numbers:

```
synthesizer.getChannels()[0].programChange(0, 15);
// change the instrument used by channel 0 to
// a dulcimer - located at bank 0, program 15
```

Instruments and soundbanks are explained in more detail in section 11.3.1.

11.2. Sending MIDI Messages to the Synthesizer's Receiver Port

This approach is functionally quite similar to the channel technique in the last section, except that we use MIDI messages directly. The advantages include the ability to direct messages to different channels, and send more kinds of messages than just NOTE_ON and NOTE_OFF.

Lists of available MIDI messages can be found at http://www.borg.com/~jglatt/tech/midispec.htm and http://users.chariot.net.au/~gmarts/midi.htm.

The receiver port for the synthesizer is obtained first:

```
Synthesizer synthesizer = MidiSystem.getSynthesizer();
synthesizer.open();
Receiver receiver = synthesizer.getReceiver();
```

As before, the sending of a note is actually two messages, separated by a delay to give the note time to play. We can wrap this up in another playNote() method.

```
public void playNote(int note, int duration, int channel)
{
  ShortMessage msg = new ShortMessage();
  try {
    msg.setMessage(ShortMessage.NOTE_ON, channel, note, 70);
                              // 70 is the volume level
    receiver.send(msg, -1);  // -1 means play immediately

    try {
      Thread.sleep(duration*1000);
    } catch (InterruptedException e) {}

    // reuse the ShortMessage object
    msg.setMessage(ShortMessage.NOTE_OFF, channel, note, 70);
    receiver.send(msg, -1);
  }
  catch (InvalidMidiDataException e)
  {  System.out.println(e.getMessage());  }
}
```

The receiver expects MIDI events, and so the MIDI message must be sent with a time-stamp. -1 means that the message should be processed immediately.

The following will be applauded once again:

```
for (int i=0; i < 10; i++)
  playNote(39, 1, 9); // sent to the drum channel, 9
```

A drawback with this technique, and the previous one, is the timing mechanism, which depends on the program sleeping. It would be better if the synthesizer managed the time spacing of MIDI messages, by working with MIDI events which use real time-stamps (tick values). This is done in the section 11.3

Control Change Messages

The FadeMidi and PanMidi examples in section 9.3 show how to access channel controllers via the synthesizer and MIDI channels. For example:

```
MidiChannel[] channels = synthesizer.getChannels();

// Set the volume controller for channel 4 to be full on (127)
int channelVol = channels[4].getController(VOLUME_CONTROLLER);
channels[4].controlChange(VOLUME_CONTROLLER, 127);
```

Another approach is to construct a MIDI message aimed at a particular channel and controller, and send it to the synthesizer's receiver.

```
// Set the volume controller for channel 4 to be full on (127)
ShortMessage volMsg = new ShortMessage();
volMsg.setMessage(ShortMessage.CONTROL_CHANGE, 4,
                                  VOLUME_CONTROLLER, 127);
receiver.send(volMsg, -1);
```

The second argument of the setMessage() is the channel ID (an index between 0 and 15, not 1 and 16), the third argument is the channel controller ID, and the fourth is the value.

11.3. Creating a Sequence

Rather than send individual notes to the synthesizer, the SeqSynth application creates a complete sequence which is passed to the sequencer, and then to the synthesizer.

The generation of a complete sequence is preferable if the music is going to be longer than just a few notes, but requires the programmer to understand the internals of a sequence. A graphical representation of a sequence's structure is given in Figure 12 (and Figure 8).

SeqSynth plays the first few notes of 'As Time Goes By' from the movie 'Casablanca'. The original MIDI note sequence was written by Heinz M. Kabutz (see http://www.javaspecialists.co.za/archive/Issue076.html).

The application constructs a sequence of MidiEvents containing NOTE_ON and NOTE_OFF messages for playing notes, and PROGRAM_CHANGE and CONTROL_CHANGE messages for changing instruments. The speed of playing is specified in terms of the ticks/beat (a PPQ resolution value) and beats/minute (the tempo setting).

The sequence only communicates with channel 0 (i.e. it only uses one musician), but this could be made more flexible.

Notes can be expressed as MIDI numbers or as note names (e.g. F4#). See http://www.phys.unsw.edu.au/~jw/notes.html for a chart linking the two. This support for note names by SeqSynth is the beginning of an application that could translate a text-based score into music.

SeqSynth's constructor:

```
public SeqSynth()
{
  createSequencer();
  // listInstruments();
  createTrack(4);       // 4 is the PPQ resolution

  makeSong();
  // makeScale(21);     // the key is "A0"

  startSequencer(60);   // tempo: 60 beats/min

  // wait for the sound sequence to finish playing
  try {
    Thread.sleep(600000);   // 10 mins in ms
  }
  catch(InterruptedException e)
  { System.out.println("Sleep Interrupted"); }
  System.exit(0);
} // end of SeqSynth()
```

createSequencer() is nothing new: it initializes the sequencer and synthesizer objects, which are assigned to global variables.

11.3.1. Instruments and Soundbanks

listInstruments() is a utility for listing all the instruments currently available to the synthesizer. The range of instruments depends on the currently loaded sound bank. The default soundbank is soundbank.gm, located in $J2SE_HOME/jre/lib/audio or $J2RE_HOME/lib/audio. It is possible to change soundbanks, for example, to improve the quality of the instruments. This is explained in the *Java Tech Tip* at http://java.sun.com/developer/JDCTechTips/2004/tt0309.html.

A soundbank, which is shown as a gray rectangle in Figure 18, can be viewed as a 2D-array, as in Figure 19.

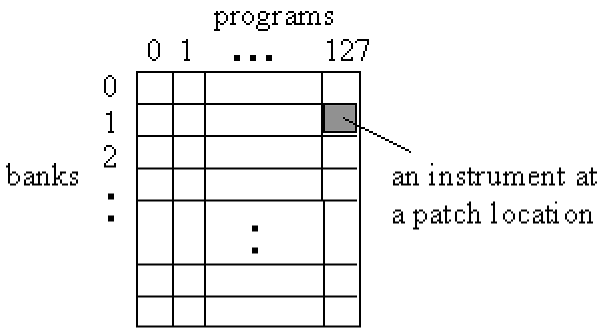


Figure 19. A Soundbank in More Detail.

Each box in the soundbank is an instrument (represented by an Instrument object), with its location stored in an associated Patch object. To utilize an instrument at run-time, it more be referred to using its Patch details. A patch holds two values: a bank number and a program number.

The General MIDI specification defines a set of instrument *names* that must be supported in bank 0, for program numbers 0 to 127 (e.g. see http://www.midi.org/about-midi/gm/gm1sound.shtml). These will be available on all MIDI synthesizers. The contents of banks 1, 2, etc. can vary.

Even within bank 0, only the names are prescribed, not the actual sound, so the output can differ from one synthesizer to another.

The General MIDI specification actually talks about banks 1-128 and programs 1-128, while Java uses 0-127 for bank and program numbers. For example, the dulcimer is in bank 1, program 16, in the specification, but is accessed using <0,15> in Java.

listInstruments() prints out the names and patch details for the extensive set of instruments in the default soundbank.

```
private void listInstruments()
{
  Instrument[] instrument = synthesizer.getAvailableInstruments();
  System.out.println("No. of Instruments: " + instrument.length);
  for (int i=0; i < instrument.length; i++) {
    Patch p = instrument[i].getPatch();
    System.out.print("(" + instrument[i].getName() +
              " <" + p.getBank() + "," + p.getProgram() + ">) ");
    if (i%3 ==0)
      System.out.println();
  }
  System.out.println();
} // end of listInstruments()
```

The output on my machine reports on four banks (0 to 3), holding a total of 411 instruments.

11.3.2. Making a Sequence

createTrack() creates a sequence with a single empty track, and specifies its MIDI event timing to be in ticks per beat (PPQ). This allows its tempo to be defined in beats/minute (see later), and also permits the tempo to be changed during execution with methods such as Sequencer.setTempoFactor(). The other time-stamp format is based on ticks/frame and frames/second (SMPTE).

```
private void createTrack(int resolution)
{ try {
    sequence = new Sequence(Sequence.PPQ, resolution);
  }
  catch (InvalidMidiDataException e) {
    e.printStackTrace();
  }
  track = sequence.createTrack();  // track is global
}
```

makeSong() fills the sequence's single track with MIDI events. In this case, reproducing the first few notes of "As Time Goes By".

```
private void makeSong()
{ changeInstrument(0,33);    // set bank and program; bass
  addRest(7);

  add("F4"); add("F4#"); add("F4"); add("D4#");
  add("C4#"); add("D4#", 3);  add("F4"); add("G4#");
  add("F4#"); add("F4"); add("D4#"); add("F4#", 3);
  add("G4#"); add("C5#"); add("C5"); add("A4#");
  add("G4#"); add("A4#", 4); add("G4", 4); add("G4#", 2);

  changeInstrument(0,15);   // dulcimer
  addRest(1);

  add("C5"); add("D5#"); add("C5#"); add("C5"); add("A4#");
  add("C5", 2); add("C5#", 2); add("G4#", 2); add("G4#", 2);
  add("C4#", 2); add("D4#", 2); add("C4#", 2);

  addRest(1);
}
```

changeInstrument() is supplied with bank and program numbers to switch the instrument. addRest() inserts a period of quiet into the sequence, equal to the supplied number of ticks. add() adds a note, with an optional tick duration parameter.

Commented out in SeqSynth.java is a simpler example: makeScale() plays a rising scale followed by a falling one.

```
private void makeScale(int baseNote)
{
  for (int i=0; i < 13; i++) {   // one octave up
    add(baseNote);
```

```
      baseNote++;
    }
    for (int i=0; i < 13; i++) {    // one octave down
      add(baseNote);
      baseNote--;
    }
  }
```

makeScale() is called with the MIDI number 21 (note A0), and subsequent notes are calculated using addition and subtraction. This version of add() takes an integer argument rather than a string.

startSequencer() is the final method called from the constructor. It plays the sequence built in the preceding call to makeSong() (or makeScale()).

```
  private void startSequencer(int tempo)
  /* Start the sequence playing.
     The tempo setting is in BPM (beats per minute),
     which is combined with the PPQ (ticks / beat)
     resolution to determine the speed of playing. */
  {
    try {
      sequencer.setSequence(sequence);
    }
    catch (InvalidMidiDataException e) {
      e.printStackTrace();
    }
    sequencer.addMetaEventListener(this);
    sequencer.start();
    sequencer.setTempoInBPM(tempo);
  } // end of startSequencer()


  public void meta(MetaMessage meta)
  // called when a meta event occurs during sequence playing
  {
    if (meta.getType() == END_OF_TRACK) {
      System.out.println("End of the track");
      System.exit(0);    // not required in J2SE 1.5
    }
  }
```

startSequence() sets the tempo, and adds a meta event listener. The listener calls meta () when the track finishes playing, allowing the application to exit immediately instead of waiting for the full 10 minutes allocated by the constructor.

11.3.3. The add() Methods

The add() methods must deal with note name or MIDI number input, and an optional note playing period.

```
  // global used to time-stamp the MidiEvent messages
  private int tickPos = 0;
         :
```

```
private void add(String noteStr)
{  add(noteStr, 1);  }

private void add(int note)
{ add(note, 1);  }

private void add(String noteStr, int period)
// convert the note string to a numerical note, then add it
{ int note = getKey(noteStr);
  add(note, period);
}

private void add(int note, int period)
{ setMessage(ShortMessage.NOTE_ON, note, tickPos);
  tickPos += period;
  setMessage(ShortMessage.NOTE_OFF, note, tickPos);
}

private void addRest(int period)
// this will leave a period of no notes (i.e. silence) in the track
{ tickPos += period; }
```

The note name is converted into a MIDI number by getKey().

The core add() method takes a MIDI number and tick period, and creates two MIDI events with setMessage(), one a NOTE_ON message, the other a NOTE_OFF, time-stamped so they are separated by the required interval.

setMessage() build a MIDI message, places it inside a MIDI event, and adds it to the track.

```
// globals
private static final int CHANNEL = 0;  // always use channel 0
private static final int VOLUME = 90;  // fixed volume for notes
           :


private void setMessage(int onOrOff, int note, int tickPos)
{
  if ((note < 0) || (note > 127)) {
    System.out.println("Note outside MIDI range (0-127): " + note);
    return;
  }

  ShortMessage message = new ShortMessage();
  try {
    message.setMessage(onOrOff, CHANNEL, note, VOLUME);
    MidiEvent event = new MidiEvent(message, tickPos);
    track.add(event);
  }
  catch (InvalidMidiDataException e) {
    e.printStackTrace();
  }
} // end of setMessage()
```

11.3.4. Changing an Instrument

changeInstrument() is supplied with the bank and program numbers of the instrument that should be used by the channel from this point on.

```
private void changeInstrument(int bank, int program)
{
  Instrument[] instrument = synthesizer.getAvailableInstruments();

  for (int i=0; i < instrument.length; i++) {
    Patch p = instrument[i].getPatch();
    if ((bank == p.getBank()) && (program == p.getProgram())) {
       programChange(program);
       bankChange(bank);
       return;
    }
  }
  System.out.println("No instrument of type <" + bank +
                                        "," + program + ">");
}
```

The validity of the two numbers are checked before they are processed.

programChange() places a PROGRAM_CHANGE MIDI message onto the track.

```
private void programChange(int program)
{
  ShortMessage message = new ShortMessage();
  try {
    message.setMessage(ShortMessage.PROGRAM_CHANGE,
                                    CHANNEL, program, 0);
                         // the second data byte (0) is unused
    MidiEvent event = new MidiEvent(message, tickPos);
    track.add(event);
  }
  catch (InvalidMidiDataException e) {
     e.printStackTrace();
  }
}
```

bankChange() is similar, but uses the bank selection channel controller (number 0), so a CONTROL_CHANGE message is place on the track.

```
// global
// channel controller name for changing an instrument bank
private static final int BANK_CONTROLLER = 0;
    :


private void bankChange(int bank)
{
  ShortMessage message = new ShortMessage();
  try {
    message.setMessage(ShortMessage.CONTROL_CHANGE,
                            CHANNEL, BANK_CONTROLLER, bank);
    MidiEvent event = new MidiEvent(message, tickPos);
```

```
      track.add(event);
    }
    catch (InvalidMidiDataException e) {
       e.printStackTrace();
    }
  }
```

11.3.5. From Note Name to MIDI Number

The note name syntax used by SeqSynth is simple, and non-standard. Only a single letter-single octave combination is allowed (e.g. "C4", "A0"), so it's not possible to refer to the -1 octave. A sharp can be included, but only after the octave number (e.g. "G4#"); the normal convention is that a sharp follows the note letter. Flats are not available.

The calculations done by getKey() use several constants:

```
  private static final int[] cOffsets =  {9, 11, 0, 2, 4, 5, 7};
                                       // A  B  C  D  E  F  G

  private static final int C4_KEY = 60;
        // C4 is the "C" in the 4th octave on a piano

  private static final int OCTAVE = 12;    // note size of an octave
```

The note offsets in cOffsets[] use the C Major scale, which is ordered "C D E F G A B", but the offsets are stored in "A B C D E F G" order to simplify their lookup by getKey().

getKey() calculates a MIDI note number by examining the note letter, octave number, and optional sharp character in the supplied string.

```
  private int getKey(String noteStr)
  /* Convert a note string (e.g. "C4", "B5#" into a key. */
  {
    char[] letters = noteStr.toCharArray();

    if (letters.length < 2) {
      System.out.println("Incorrect note syntax; using C4");
      return C4_KEY;
    }

    // look at note letter in letters[0]
    int c_offset = 0;
    if ((letters[0] >= 'A') && (letters[0] <= 'G'))
      c_offset = cOffsets[letters[0] - 'A'];
    else
      System.out.println("Incorrect: " + letters[0] + ", using C");

    // look at octave number in letters[1]
    int range = C4_KEY;
    if ((letters[1] >= '0') && (letters[1] <= '9'))
      range = OCTAVE * (letters[1] - '0' + 1);
    else
      System.out.println("Incorrect: " + letters[1] + ", using 4");
    // look at optional sharp in letters[2]
    int sharp = 0;
    if ((letters.length > 2) && (letters[2] == '#'))
      sharp = 1;    // a sharp is 1 note higher
    int key = range + c_offset + sharp;

    return key;
  }  // end of getKey()
```

The method could be improved by carrying out additional checking to prevent sharps being added to the 'E' and 'B' notes.

11.3.6. Extending SeqSynth

SeqSynth would be more flexible if it could read song operations (i.e. a score) from a text file, instead of having them 'wired' into methods such as makeSong().

The range of musical notation understood by SeqSynth could be enlarged. For example, David Flanagan's PlayerPiano application from "Java Examples in a Nutshell" covers similar ground to SeqSynth, but also supports flats, chords (combined notes), volume control, and the damper pedal (http://www.onjava.com/pub/a/onjava/excerpt/jenut3_ch17/index1.html). The resulting sequence can be played or saved to a file.

There are several ASCII-notations for representing scores, such as the abc language (http://www.gre.ac.uk/~c.walshaw/abc/). abc is widely used for notating and distributing music. Many tools exist for playing abc notated music, converting it into MIDI sequences or sheet music, and so on. Wil Macaulay has written Skink, a Java application, which supports the abc 1.6 standard with some extensions. It can open, edit, save, play, and display abc files (http://www.geocities.com/w_macaulay/skink.html). Skink generates a MIDI sequence using similar techniques as in SeqSynth.

12. Audio Synthesis Libraries

The sampled audio synthesis carried out by NotesSynth in section 10.3, and the MIDI sequence generation in SeqSynth could be greatly expanded to turn the applications into general-purpose synthesis tools, classes, or libraries. However, Java audio synthesis libraries already exist, but not as part of J2SE.

JSyn (http://www.softsynth.com/jsyn/) generates sound effects by employing inter-connected unit generators. It includes an extensive library of generators, including oscillators, filters, envelopes, and noise generators. For example, a wind sound can be built by connecting a white noise generator to a low pass filter modulated by a random contour generator.

JSyn comes with a graphical editor, called Wire, for connecting unit generators together. The result can be exported as Java source.

jMusic (http://jmusic.ci.qut.edu.au/) is aimed at musicians rather than engineers. Its libraries provide a music data structure based around note and sound events, with associated methods. jMusic can read and write MIDI and audio files.

13. Comparisons with JMF and JOAL

JMF (http://java.sun.com/products/java-media/jmf/). The Java Media Framework supports streaming multimedia, such as video and audio, with an emphasis on streaming over a network where bandwidth and latency are issues. This means support for time-based protocols, such as RTP, and services such as compression and media streams synchronization.

The 'performance pack' version of JMF use Java Sound to play and capture sound data, so Java Sound techniques can be utilized. However, the 'all-java' version of JMF uses sun.audio classes to play sound rather than Java Sound (capture is not available).

JMF supports more sound formats than Java Sound, including MPEG-1 (see http://java.sun.com/products/java-media/jmf/2.1.1/formats.html for an extensive list). However, it is possible to plug additional codecs into Java Sound via the service provider interface. For example, MP3 (MPEG 1/2/2.5 Layer 1/2/3) and Ogg Vorbis formatted files can be read through an AudioInputStream by utilizing plug-ins from JavaZoom (http://www.javazoom.net/projects.html).

JMF can be utilized with JDK 1.1 or later, so is suitable for applets running inside JVMs on older browsers; Java Sound requires J2RE 1.3 or higher.

JOAL (https://joal.dev.java.net/). Java OpenAL is a set of Java bindings for OpenAL, a 3D sound API for OpenGL. JOAL's area of strength is 3D positional audio, and offers little support for audio mixing or synthesis; consequently, it doesn't 'compete' with Java Sound.

A combination of JOAL and Java Sound may replace the buggy audio elements of Java 3D in the future.

14. Java Sound Resources

The lengthy Java Sound programmer's guide comes with the J2SE documentation, and can be found at http://java.sun.com/j2se/1.4.2/docs/guide/sound/programmer_guide/. It's a little bit old now, but still very informative.

The best place for examples, links, and a great FAQ is the Java Sound resources site (http://www.jsresources.org/).

Lots of specialized information can be extracted from the javasound-interest mailing list at http://archives.java.sun.com/archives/javasound-interest.html. The Java Games Forum on Java Sound may also be useful (and is searchable); visit http://www.javagaming.org/cgi-bin/JGNetForums/YaBB.cgi?board=Sound

Sun's Java Sound site at http://java.sun.com/products/java-media/sound/ contains links to articles, a FAQ, a large demo, instruments soundbacks, and service provider plug-ins for non-standard audio formats.

The Java Almanac offers code fragments illustrating various techniques (http://javaalmanac.com/egs/?). Look under the javax.sound.sampled and javax.sound.midi package headings.

An excellent set of Java Sound examples can be found in "Java Examples in a Nutshell" by David Flanagan (O'Reilly, 3rd edition, 2004). They include a MIDI synthesizer based around the processing of musical notes, covering similar ground to my SeqSynth example, but with additional features. All the examples can be downloaded from O'Reilly's Web site at http://www.oreilly.com/catalog/jenut3/index.html?CMP=ILC-0PY480989785, and there are two excerpts from the Java Sound chapter (chapter 17) at http://www.onjava.com/pub/a/onjava/excerpt/jenut3_ch17/index.html and http://www.onjava.com/pub/a/onjava/excerpt/jenut3_ch17/index1.html.

Extended coverage of Java Sound appears in chapter 22 of "Java: How to Program" by Harvey and Paul Deitel (Deitel Int., **fourth** edition, 2002). The MIDI example combines synthesis, playback, recording, and saving. The code can be downloaded from http://www.deitel.com/books/downloads.html. Unfortunately, the Java Sound material has been cut from the fifth edition of this book.

Dick Baldwin has written several Java Sound tutorials, found at http://dickbaldwin.com/tocadv.htm; topics include "Capturing microphone data into an audio file", and "Creating, playing, and saving synthetic sounds".

A good starting point for Java Sound software, including tools and libraries, is the Google directory http://directory.google.com/Top/Computers/Multimedia/Music_and_Audio/Software/Java/?il=1

15. Audio Resources

I've already mentioned various editing tools, including WavePad (http://nch.com.au/wavepad/) and Anvil Studio (http://www.anvilstudio.com/).

The source for all things MIDI is http://www.midi.com, with articles, the official specification, a search engine, and links to other sites.

Harmony Central has a good MIDI resource section, including a very nice tutorial, http://www.harmony-central.com/MIDI/.

FindSounds (http://www.findSounds.com) offers a versatile search engine for sampled audio (AIFF, AU, WAV). MusicRobot (http://www.musicrobot.com) has a MIDI search engine, and a WAV search engine (http://www.musicrobot.com/cgi-bin/windex.pl)

Audio clip Web sites with plenty of sound effects, such as gun shots, sirens, and explosions, include http://www.freeaudioclips.com, http://www.wavsource.com, and http://www.a1freesoundeffects.com/.