Java Sound, An Introduction

Baldwin presents the first lesson in a new miniseries that will teach you how to use the Java Sound API.

Published: January 7, 2003
By Richard G. Baldwin

Java Programming Notes # 2004

Preface

What is sound?

From a human perspective, sound is the sensation that we experience when pressure waves impinge upon the small parts contained within our ears. Normally, this is the result of pressure waves being transmitted in air. However, sound pressure waves are not limited to air. For example, if you are an underwater swimmer, sound pressure waves may reach your ear by way of water.

From the perspective of the *Java Sound API*, the word *Sound* takes on a somewhat different meaning. However, it is probably fair to say that the ultimate purpose of the Sound API is to assist you in writing programs that will cause sound pressure waves impinge upon the ears of targeted individuals at specific times.

What does Sun have to say about the API?

Here is what Sun has to say about the Java Sound API:

> *"The Java Sound API is a low-level API for effecting and controlling input and output of audio media. It provides explicit control over the capabilities commonly required for audio input and output in a framework that promotes extensibility and flexibility."*

Sun also tells us:

"Java Sound provides the lowest level of audio support on the Java platform. It provides a high degree of control over audio-specific functionality. ... It does not include sophisticated sound editors and GUI tools; rather, it provides a set of capabilities upon which such applications can be built. It emphasizes low-level control beyond that commonly expected by the end user, who benefits from higher-level interfaces built on top of Java Sound."

Thus, your mission as a Java programmer is to use the Sound API to produce higher-level user interfaces built on top of Java Sound.

Not a trivial API

There are a number of fairly complex issues involved in the use of the Sound API. This tutorial lesson will provide a very brief introduction to some of those issues. Future lessons will explore many of them in detail.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings and figures while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

Preview

This lesson provides a description of *sound* from both a physical and a programming viewpoint.

The lesson introduces the *Java Sound API,* which provides a high degree of control over audio-specific functionality in Java programs.

It identifies the important packages incorporated in the Sound API and explains the difference between the *sampled* packages and the *MIDI* packages.

The lesson provides a description of *sampled audio,* and explains the typical steps used to capture sampled audio. It also explains the typical steps used to render sampled audio.

Finally, the lesson provides a program that you can use to first capture and then to play back audio sound.

Discussion and Sample Code

Packages

Two significantly different types of audio *(or sound)* data are supported by the API:

- Sampled audio data
- Musical Instrument Digital Interface (MIDI) data

Sampled audio data

Sampled audio data can be thought of as a series of digital values that represent the amplitude or intensity of sound pressure waves. This will be the primary topic of the first several lessons in this miniseries. This type of audio data is supported by the following two Java packages:

- javax.sound.sampled
- javax.sound.sampled.spi

According to Sun, the first of these two packages *"specifies interfaces for capture, mixing, and playback of digital (sampled) audio."* I will have more to say about the second *(spi)* package shortly.

MIDI data

MIDI data can be thought of as sound *(usually musical sound or special sound effects)* created from a recipe. This type of audio data is covered by the following two Java packages:

- javax.sound.midi
- javax.sound.midi.spi

According to Sun, the first of these two packages *"provides interfaces for MIDI synthesis, sequencing, and event transport."*

What about the spi packages?

According to Sun, each of the spi packages *"permits service providers (as opposed to application developers) to create custom components that can be installed on the system"*

Because the two types of audio data are so significantly different, I will defer any detailed discussion of MIDI audio data until later.

What is sampled data?

I am going to refer you to another publication of mine entitled Digital Signal Processing (DSP) in Java, Sampled Time Series for a discussion of sampled data in general. Because DSP

techniques are often used in the processing of sampled audio data, you may also be interested in my other publications on DSP as well.

What is sampled audio data?

Sampled audio data is a special case of sampled data in general. For sampled audio data, a series of digital numeric values is used to represent the intensity of a sound pressure wave. In other words, a set of numeric values is used to represent the actual waveform of a sound pressure wave. Typically, the sound pressure wave *(or an electronic representation thereof)* is sampled at a uniform series of points in time.

An example

For example, the graph in Figure 1 might represent a set of sampled audio data values produced by a wide-band noise generator, such as the noise at an airport.

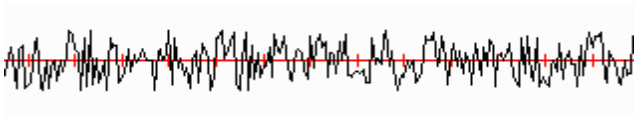

Figure 1 Sampled audio data

You can think of Figure 1 as the result of connecting a series of dots with short straight lines. The vertical position of each dot relative to the red horizontal axis would represent the intensity of a sound wave at a particular point in time. The location of each dot along the horizontal axis would represent the point in time at which the measurement of intensity was made.

An audio compact disk (CD)

As another example, the data on an audio CD is sampled audio data. As I understand it, an electronic representation of the sound pressure waves produced by the artist is sampled 44,100 times per second. Each sample is represented as a 16-bit signed integer value.

Other sources of sampled audio data

Although sampled audio data most commonly results from actually sampling sound pressure waves, such data could be synthetically generated by a computer.

For example, computerized speech synthesizers can be used to produce sampled audio data. When that data is converted to sound pressure waves *(rendered)*, which impinge on a human ear, the human experiences the sensation of sound representing human speech.

According to Sun:

"The term "sampled audio" refers to the type of data, not its origin. Sampled audio can be thought of as the sound itself, whereas MIDI data can be thought of as a recipe for creating musical sound.) "

Capturing sampled audio

Typically sampled audio data is captured in two steps:

- Use a microphone to convert the sound pressure waves to electrical voltages that mimic the waveform of the sound pressure wave.
- Use an analog-to-digital converter to measure the voltage at specific points in time and to convert that measurement to a digital value.

Rendering sampled audio

The rendering of sampled audio data is also typically accomplished in two steps:

- Use a digital-to-analog converter to convert a series of digital values into an analog voltage whose amplitude waveform reflects the digital values.
- Apply this voltage to a speaker, a set of headphones, or some other similar device that converts the analog voltage to sound pressure waves that mimic the waveform of the voltage.

Run the Program

At this point, you may find it useful to compile and run the program in Listing 1 near the end of the lesson.

(I have provided this sample program with very little in the way of an explanation as to how the program works. I wanted to give you some code to get you started using the API. I will explain this program, or other very similar programs in future lessons.)

Capture and playback audio data

This program demonstrates the ability to capture audio data from a microphone and to play it back through the speakers on your computer. The usage instructions are simple:

- Start the program running. A simple GUI will appear on the screen.
- Click the **Capture** button and speak into the microphone.
- Click the **Stop** button to terminate capturing data.
- Click the **Playback** button to play your captured voice back through the system speakers.

If you don't hear anything during playback, you may need to increase your speaker volume.

This program saves the data that it captures in memory, so be careful. If you attempt to save too much data, you may run out of memory.

Summary

In this lesson, I provided a description of *sound* from both a physical and a programming viewpoint.

I introduced the *Java Sound API,* which provides a high degree of control over audio-specific functionality in Java programs.

I identified the important packages incorporated in the Sound API, and explained the difference between the *sampled* packages and the *MIDI* packages.

I provided a description of *sampled audio,* and explained the typical steps used to capture sampled audio. I also explained the typical steps used to render sampled audio into sound pressure waves.

Finally, I provided a relatively simple program that you can use to first capture and then to playback audio sound.

What's Next?

In the next lesson, I will explain the overall architecture of the Sound API, introducing such terms and concepts as:

- Lines
- TargetDataLine
- SourceDataLine
- Mixers
- Ports
- Audio format
- File format
- Audio stream
- Clip
- Controls

Complete Program Listing

A complete listing of the program is shown in Listing 1.

```
/*File AudioCapture01.java
This program demonstrates the capture
and subsequent playback of audio data.
```

```
A GUI appears on the screen containing
the following buttons:
Capture
Stop
Playback

Input data from a microphone is
captured and saved in a
ByteArrayOutputStream object when the
user clicks the Capture button.

Data capture stops when the user clicks
the Stop button.

Playback begins when the user clicks
the Playback button.

Tested using SDK 1.4.0 under Win2000
***********************************/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.sound.sampled.*;

public class AudioCapture01
                        extends JFrame{

  boolean stopCapture = false;
  ByteArrayOutputStream
                  byteArrayOutputStream;
  AudioFormat audioFormat;
  TargetDataLine targetDataLine;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;

  public static void main(
                        String args[]){
    new AudioCapture01();
  }//end main

  public AudioCapture01(){//constructor
    final JButton captureBtn =
                 new JButton("Capture");
    final JButton stopBtn =
                   new JButton("Stop");
    final JButton playBtn =
                new JButton("Playback");

    captureBtn.setEnabled(true);
    stopBtn.setEnabled(false);
    playBtn.setEnabled(false);

    //Register anonymous listeners
    captureBtn.addActionListener(
```

```
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent e){
          captureBtn.setEnabled(false);
          stopBtn.setEnabled(true);
          playBtn.setEnabled(false);
          //Capture input data from the
          // microphone until the Stop
          // button is clicked.
          captureAudio();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()
    getContentPane().add(captureBtn);

    stopBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent e){
          captureBtn.setEnabled(true);
          stopBtn.setEnabled(false);
          playBtn.setEnabled(true);
          //Terminate the capturing of
          // input data from the
          // microphone.
          stopCapture = true;
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()
    getContentPane().add(stopBtn);

    playBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                        ActionEvent e){
          //Play back all of the data
          // that was saved during
          // capture.
          playAudio();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()
    getContentPane().add(playBtn);

    getContentPane().setLayout(
                      new FlowLayout());
    setTitle("Capture/Playback Demo");
    setDefaultCloseOperation(
                        EXIT_ON_CLOSE);
    setSize(250,70);
    setVisible(true);
  }//end constructor

  //This method captures audio input
  // from a microphone and saves it in
  // a ByteArrayOutputStream object.
```

```
  private void captureAudio(){
    try{
      //Get everything set up for
      // capture
      audioFormat = getAudioFormat();
      DataLine.Info dataLineInfo =
                new DataLine.Info(
                  TargetDataLine.class,
                   audioFormat);
      targetDataLine = (TargetDataLine)
                   AudioSystem.getLine(
                          dataLineInfo);
      targetDataLine.open(audioFormat);
      targetDataLine.start();

      //Create a thread to capture the
      // microphone data and start it
      // running.  It will run until
      // the Stop button is clicked.
      Thread captureThread =
                new Thread(
                  new CaptureThread());
      captureThread.start();
    } catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end captureAudio method

  //This method plays back the audio
  // data that has been saved in the
  // ByteArrayOutputStream
  private void playAudio() {
    try{
      //Get everything set up for
      // playback.
      //Get the previously-saved data
      // into a byte array object.
      byte audioData[] =
                 byteArrayOutputStream.
                         toByteArray();
      //Get an input stream on the
      // byte array containing the data
      InputStream byteArrayInputStream
            = new ByteArrayInputStream(
                             audioData);
      AudioFormat audioFormat =
                       getAudioFormat();
      audioInputStream =
        new AudioInputStream(
          byteArrayInputStream,
          audioFormat,
          audioData.length/audioFormat.
                        getFrameSize());
      DataLine.Info dataLineInfo =
                new DataLine.Info(
```

```
                    SourceDataLine.class,
                            audioFormat);
      sourceDataLine = (SourceDataLine)
                     AudioSystem.getLine(
                            dataLineInfo);
      sourceDataLine.open(audioFormat);
      sourceDataLine.start();

      //Create a thread to play back
      // the data and start it
      // running.  It will run until
      // all the data has been played
      // back.
      Thread playThread =
          new Thread(new PlayThread());
      playThread.start();
    } catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end playAudio

  //This method creates and returns an
  // AudioFormat object for a given set
  // of format parameters.  If these
  // parameters don't work well for
  // you, try some of the other
  // allowable parameter values, which
  // are shown in comments following
  // the declarations.
  private AudioFormat getAudioFormat(){
    float sampleRate = 8000.0F;
    //8000,11025,16000,22050,44100
    int sampleSizeInBits = 16;
    //8,16
    int channels = 1;
    //1,2
    boolean signed = true;
    //true,false
    boolean bigEndian = false;
    //true,false
    return new AudioFormat(
                      sampleRate,
                      sampleSizeInBits,
                      channels,
                      signed,
                      bigEndian);
  }//end getAudioFormat
//=================================//

//Inner class to capture data from
// microphone
class CaptureThread extends Thread{
  //An arbitrary-size temporary holding
  // buffer
  byte tempBuffer[] = new byte[10000];
```

```
  public void run(){
    byteArrayOutputStream =
           new ByteArrayOutputStream();
    stopCapture = false;
    try{//Loop until stopCapture is set
        // by another thread that
        // services the Stop button.
      while(!stopCapture){
        //Read data from the internal
        // buffer of the data line.
        int cnt = targetDataLine.read(
                    tempBuffer,
                    0,
                    tempBuffer.length);
        if(cnt > 0){
          //Save data in output stream
          // object.
          byteArrayOutputStream.write(
                    tempBuffer, 0, cnt);
        }//end if
      }//end while
      byteArrayOutputStream.close();
    }catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end run
}//end inner class CaptureThread
//===================================//
//Inner class to play back the data
// that was saved.
class PlayThread extends Thread{
  byte tempBuffer[] = new byte[10000];

  public void run(){
    try{
      int cnt;
      //Keep looping until the input
      // read method returns -1 for
      // empty stream.
      while((cnt = audioInputStream.
        read(tempBuffer, 0,
            tempBuffer.length)) != -1){
        if(cnt > 0){
          //Write data to the internal
          // buffer of the data line
          // where it will be delivered
          // to the speaker.
          sourceDataLine.write(
                    tempBuffer, 0, cnt);
        }//end if
      }//end while
      //Block and wait for internal
      // buffer of the data line to
      // empty.
      sourceDataLine.drain();
```

```
      sourceDataLine.close();
    }catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end run
}//end inner class PlayThread
//===================================//

}//end outer class AudioCapture01.java
```

Listing 1

About the author

Richard Baldwin is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

-end-