Java Sound, Playing Back Audio Files using Java

Baldwin shows you how to write a program that you can use to play back audio files, including those that you create using a Java program, and those that you acquire from other sources.

Published: April 1, 2003
By Richard G. Baldwin

Java Programming Notes # 2016

- Preface
- Preview
- Discussion and Sample Code
- Run the Program
- Summary
- Complete Program Listing

Preface

This series of lessons is designed to teach you how to use the Java Sound API. The first lesson in the series was entitled Java Sound, An Introduction. The previous lesson was entitled Java Sound, Capturing Microphone Data into an Audio File.

Two types of audio data

Two different types of audio data are supported by the Java Sound API:

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

The two types of audio data are very different. I am concentrating on sampled audio data at this point in time. I will defer my discussion of MIDI until later.

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

The previous lesson showed you how to use the Java Sound API to write programs to capture microphone data into audio files types of your own choosing. At that time, I told you that you should be able to play the audio files back using readily available media players, such as the Windows Media Player.

In this lesson, I will provide and explain a program that you can use to play back audio files, including those that you create using a Java program, as well as those that you acquire from other sources.

Discussion and Sample Code

What is a SourceDataLine?

I will be using a **SourceDataLine** object in this program. A **SourceDataLine** object is a streaming mixer input object. *(A previous lesson explained lines and mixers in detail.)*

A **SourceDataLine** object can accept a stream of audio data and push that audio data into a mixer in real time. The actual audio data can derive from a variety of sources, such as an audio file, *(which will be the case in this program)* a network connection, or a buffer in memory.

Confusing terminology

The terminology used with the Java sound API can be very confusing. Here is part of what Sun has to say about a **SourceDataLine** object.

> *"A source data line is a data line to which data may be written. It acts as a source to its mixer. An application writes audio bytes to a source data line, which handles the buffering of the bytes and delivers them to the mixer. The mixer may ... deliver the mix to a target such as an output port (which may represent an audio output device on a sound card).*
>
> *Note that the naming convention for this interface reflects the relationship between the line and its mixer. From the perspective of an application, a source data line may act as a target for audio data."*

This concept is discussed in detail in the lesson entitled Java Sound, Getting Started, Part 1, Playback.

Audio data output

The data written to the **SourceDataLine** object can be pushed into a mixer in real time. The actual destination of the audio data can be any of a variety of destinations.

> *(The sample program in this lesson writes audio data into a SourceDataLine object, which pushes the data to the computer speakers.)*

The user interface

When this program is executed, the GUI shown in Figure 1 appears on the screen. As you can see, this GUI contains the following components:

- A text field
- A Play button
- A Stop button

Figure 1 Program GUI

Operation of the program

The user enters the path and file name for an audio file into the text field, and then clicks the **Play** button. The contents of the audio file are played back through the system speakers.

Playback continues until the system encounters the end of the audio file, or the user clicks the **Stop** button, whichever occurs first.

Although the user can terminate the playback of the audio file by clicking the **Stop** button, because the audio data is buffered in a large buffer in the playback loop, there may be a noticeable delay between the time that the **Stop** button is clicked and the time that the playback actually terminates.

Audio data format is displayed

The program also displays the format of the audio data before playing the file. The format is displayed on the command- line screen.

Will discuss in fragments

As usual, I will discuss this program in fragments. A complete listing of the program is shown in Listing 17 near the end of the lesson.

The class named AudioPlayer02

The program named **AudioPlayer02** demonstrates the use of a Java program to play back the contents of an audio file. The class definition for the controlling class begins in Listing 1.

```
public class AudioPlayer02 extends
JFrame{

  AudioFormat audioFormat;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;
  boolean stopPlayback = false;

  final JButton stopBtn = new
JButton("Stop");
  final JButton playBtn = new
JButton("Play");
  final JTextField textField =
                        new
JTextField("junk.au");
```

Listing 1

Instance variables

Several instance variables are declared in Listing 1. Some of those instance variables are also initialized. It is worth noting that the text field object is initialized to contain the default file name **junk.au**. This makes it possible to run the program without the requirement to type the file name in the text field, provided that an audio file named **junk.au** resides in the same directory as the controlling class file. *(A previous lesson contained a program that created an audio file named junk.au.)*

The main method

The **main** method for this Java application, shown in Listing 2, is extremely simple.

```
  public static void main(String
args[]){
    new AudioPlayer02();
  }//end main
```

Listing 2

The code in the **main** method simply instantiates an object of the controlling class. Code in the constructor, some other methods, and some inner classes take over at that point and provide the operational behavior of the program.

The constructor

The constructor begins in Listing 3.

```
  public AudioPlayer02(){//constructor

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

Listing 3
```

The code in Listing 3 causes the **Stop** button to be disabled and causes the **Play** button to be enabled when the program first starts running. This is the situation shown in Figure 1.

Action listener on the Play button

The code in Listing 4 instantiates and register an action listener on the **Play** button.

```
    playBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(

ActionEvent e){
          stopBtn.setEnabled(true);
          playBtn.setEnabled(false);
          playAudio();//Play the file
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

Listing 4
```

I explained this anonymous inner class syntax in the previous lesson. The most significant thing about the code in Listing 4 is the statement highlighted in boldface. This statement invokes the **playAudio** method to play the contents of the audio file when the user clicks the **Play** button.

The other two statements in Listing 4 simply reverse the enabled/disabled state of the **Play** and **Stop** buttons.

Action listener on the Stop button

We are still discussing the constructor for the controlling class. The code in Listing 5 instantiates and registers an action listener on the **Stop** button.

```
    stopBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(
```

```
ActionEvent e){
          //Terminate playback before
EOF
          stopPlayback = true;
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

Listing 5
```

When the user clicks the **Stop** button, the code in Listing 5 sets a flag named **stopPlayback** to true. As you will see later, the thread that handles the actual playback operation periodically tests this flag. If the flag switches from false to true, the code in the **run** method of the playback thread terminates playback at that point in time.

There may be a time delay before termination of playback

However, there are some rather large buffers used in the playback loop. These buffers must be emptied before playback actually terminates. Therefore, there may be a time delay between the point in time that this flag is set to true and the point in time that playback actually terminates.

Finish constructing the GUI

The code in Listing 6 finishes the construction of the GUI.

```
getContentPane().add(playBtn,"West");

getContentPane().add(stopBtn,"East");

getContentPane().add(textField,"North");

    setTitle("Copyright 2003,
R.G.Baldwin");

setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(250,70);
    setVisible(true);
  }//end constructor

Listing 6
```

This code does the following:

- Places the text field and the two buttons in their respective positions in the GUI.
- Sets the title and the size of the GUI.
- Enables the X button in the upper-right corner of the frame to terminate the program when it is clicked.
- Makes the whole thing visible.

The playAudio method

Recall that the event handler on the **Play** button invokes the **playAudio** method. The **playAudio** method plays back audio data from an audio file, where the name of the audio file is specified in the text field of the GUI. The **playAudio** method begins in Listing 7.

```
  private void playAudio() {
    try{
      File soundFile =
                  new
File(textField.getText());

Listing 7
```

A File object

The first thing that we need to do is to create a new **File** object that provides an abstract representation of the file and directory pathname entered by the user in the text field of the GUI.

This is accomplished in Listing 7 by:

- Getting a **String** object that encapsulates the text entered by the user into the text field.
- Passing that **String** object's reference to the constructor of the **File** class.

An AudioInputStream object

Next, we need to get an **AudioInputStream** object that is connected to the audio file. This is accomplished in Listing 8.

```
      audioInputStream = AudioSystem.

getAudioInputStream(soundFile);

Listing 8
```

Listing 8 passes the **File** object's reference to the static **getAudioInputStream** method of the **AudioSystem** class. There are several overloaded versions of the **getAudioInputStream** method in Java SDK version 1.4.1. Here is part of what Sun has to say about the version of the method used in Listing 8.

> *"Obtains an audio input stream from the provided File. The File must point to valid audio file data."*

The audio format

You learned in previous lessons that audio data comes in a variety of audio formats. The first statement in Listing 9 invokes the **getFormat** method on the **AudioInputStream** object to get and save the audio format of the specified audio file.

```
      audioFormat =
audioInputStream.getFormat();

      System.out.println(audioFormat);
```

Listing 9

Sample formats

The second statement in Listing 9 displays the reported format.

Here is the reported format for a file named **tada.wav**, which I found in the operating system media directory on my Win2000 system.

```
PCM_SIGNED, 22050.0 Hz, 16 bit, stereo, little-endian, audio data
```

Here is the reported format for a file named **junk.au**, which was produced by the program named **AudioRecorder02** in the previous lesson.

```
PCM_SIGNED, 8000.0 Hz, 16 bit, mono, big-endian, audio data
```

Be careful with the backslashes

If you are using a Windows system, you must either specify the path and file name in the GUI text field using forward slashes, or by using double backslashes. If the audio file is in the same directory as the compiled version of this program, you don't have to specify a path.

A DataLine.Info object

In the previous lesson, you learned of the requirement to create a **DataLine.Info** object that described the **TargetDataLine** that was needed to handle the acquisition of the audio data from the microphone.

A similar requirement exists in this program. However, in this program, we need a **DataLine.Info** object that describes the **SourceDataLine** object that we will use to feed the audio data to the speakers. The **DataLine.Info** object is instantiated in Listing 10.

```
      DataLine.Info dataLineInfo =
                          new
DataLine.Info(

SourceDataLine.class,
```

```
audioFormat);

Listing 10
```

The DataLine.Info constructor

Note that in this case, the first parameter to the **DataLine.Info** constructor is a **Class** object that represents the type of an object instantiated from a class that implements the **SourceDataLine** interface.

The second parameter to the constructor specifies the format of the audio data to be handled by the **SourceDataLine** object.

Getting a SourceDataLine object

Finally, the code in Listing 11 invokes the static **getLine** method of the **AudioSystem** class to get a **SourceDataLine** object that matches the description of the object provided by the **DataLine.Info** object.

```
      sourceDataLine =

(SourceDataLine)AudioSystem.getLine(

dataLineInfo);

Listing 11
```

I discussed the static **getLine** method of the **AudioSystem** class in detail in the previous lesson, so I won't discuss it further here.

Spawn and start a thread to handle the playback

The boldface statement in Listing 12 creates a thread *(to play back the data)* and starts the thread running. The thread will run until the end of the file is encountered, or the **Stop** button is clicked by the user, whichever occurs first. *(Because of the data buffers involved, there will normally be a delay between the click on the **Stop** button and the actual termination of playback.)*

```
      new PlayThread().start();
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end playAudio

Listing 12
```

Once the thread is started, the **playAudio** method returns control to the action event handler on the **Play** button, which terminates very shortly thereafter.

The remaining code in Listing 12 consists simply of a required **catch** block. The **playAudio** method ends in Listing 12.

The run method of the PlayThread class

The **PlayThread** class is an inner class. An object of this class is used to perform the actual playback of the data from the audio file.

Every thread object has a run method, which determines the behavior of the thread. Listing 13 shows the beginning of the **run** method of the **PlayThread** class.

```
class PlayThread extends Thread{
  byte tempBuffer[] = new byte[10000];

  public void run(){
    try{

sourceDataLine.open(audioFormat);
      sourceDataLine.start();

Listing 13
```

The array object of type **byte** that is instantiated in Listing 13 will be used to transfer the data from the audio file stream to the **SourceDataLine** object.

The open method of the SourceDataLine object

The code in Listing 13 invokes the **open** method on the **SourceDataLine** object. According to Sun, the **open** method of a **SourceDataLine** object

> *"Opens the line with the specified format, causing the line to acquire any required system resources and become operational."*

The start method of the SourceDataLine object

The code in Listing 13 also invokes the **start** method on the **SourceDataLine** object. According to Sun, the **start** method of a **SourceDataLine** object

> *"Allows a line to engage in data I/O. If invoked on a line that is already running, this method does nothing."*

Thus, when the code in Listing 13 has been executed, the **SourceDataLine** object has acquired all necessary system resources and is ready to transfer audio data to the system speakers.

The data-transfer loop

The conditional clause of the **while** loop in Listing 14 is a little complex. The **while** loop in Listing 14 loops until:

- The **read** method of the **AudioInputStream** object returns a minus 1, indicating empty stream *(end of file)*, or
- The user clicks the **Stop** button causing **stopPlayback** to switch from false to true.

```
      int cnt;

      while((cnt =
audioInputStream.read(

tempBuffer,0,tempBuffer.length)) != -1
                       && stopPlayback
== false){
        if(cnt > 0){
          sourceDataLine.write(

tempBuffer, 0, cnt);
        }//end if
      }//end while loop

Listing 14
```

A positive value from the read method

If the **read** method returns a positive value, that value indicates the number of bytes that were read from the audio file into the temporary buffer named **tempBuffer**.

If the number of bytes that were read is greater than zero, the **write** method of the **SourceDataLine** object is invoked to transfer those bytes to an internal buffer of the **SourceDataLine** object.

Convert sampled audio data to sound pressure waves

The **SourceDataLine** object causes the sampled audio data values to be converted to analog voltages, which are delivered to the speakers where the voltages are converted to sound pressure waves.

In order to accomplish this, the **SourceDataLine** object must know about the format of the audio data, including the sample rate, the number of channels, the number of bits per sample, etc. That information was provided to the **SourceDataLine** object by way of the **DataLine.Info** object in Listing 10.

Drain the SourceDataLine buffer

When the **while** loop terminates for one of the reasons discussed earlier, there may still be data in the internal buffer of the **SourceDataLine** object. The code in Listing 15 invokes the **drain** method on the **SourceDataLine** object to continue the data transfer in real time until this data is exhausted.

```
      sourceDataLine.drain();
      sourceDataLine.close();

Listing 15
```

Here is part of what Sun has to say about the **drain** method.

> *"Drains queued data from the line by continuing data I/O until the data line's internal buffer has been emptied. This method blocks until the draining is complete."*

The close method

The code in Listing 15 also invokes the **close** method on the **SourceDataLine** object. Here is part of what Sun has to say about the **close** method.

> *"Closes the line, indicating that any system resources in use by the line can be released."*

Thus, when the code in Listing 15 has finished executing, all of the data delivered to the **SourceDataLine** object has been passed along to the speakers, and the **SourceDataLine** object has release system resources.

Prepare to play back another file

The three statements in Listing 16 reset some values to prepare the program to play back another file.

```
      stopBtn.setEnabled(false);
      playBtn.setEnabled(true);
      stopPlayback = false;

Listing 16
```

With the exception of a **catch** block, the code in Listing 16:

- Signals the end of the **run** method.
- Signals the end of the **PlayThread** inner class.
- Signals the end of the program.

You can view the **catch** block in Listing 17 near the end of the lesson.

Run the Program

At this point, you may find it useful to compile and run the program shown in Listing 17 near the end of the lesson.

Specify the audio file

Start the program and enter the path and file name of an audio file in the text field that appears on the GUI shown in Figure 1. You should separate the directories in the path using forward slashes instead of backslashes. If you are running under Windows and you want to use backslashes, use two backslashes in each case in place of one backslash. If the audio file is in the same directory as the program *(the current directory)*, you don't need to enter the path.

Start the playback

Click the **Play** button after entering the file name. Playback should begin, and should continue to the end of the audio file unless you click the **Stop** button while the file is being played.

The Stop button

If you click the **Stop** button while the file is being played, playback should terminate. However, because some relatively large buffers are used in the playback loop, there may be a noticeable delay between the time that you click the **Stop** button and the time that the playback actually terminates.

Volume control

If you don't hear anything during playback, you may need to increase your speaker volume. My laptop computer has a manual volume control in addition to the software volume controls that are accessible via the speaker icon in the system tray.

Summary

In this lesson, I have explained a program that you can use to play back audio files, including those that you create using a Java program, as well as those that you acquire from other sources.

Complete Program Listing

A complete listing of the program is shown in Listing 17.

```
/*File AudioPlayer02.java
Copyright 2003 Richard G. Baldwin
```

```
Demonstrates playback of an audio file.  The
path and name of the audio file is specified by
the user in a text field.

A GUI appears on the screen containing
the following components:
  Text field for the file name
  Play
  Stop

After entering an audio file name in the text
field, the user can click the Play button to
cause the program to play the audio file.  By
default, the program will play the entire file
and then get ready to play another file, or to
play the same file again.

If the user clicks the Stop button while the file
is being played, playback will terminate.
However, because the audio data is buffered in a
large buffer in the playback loop, there may be a
noticeable delay between the time that the Stop
button is clicked and the time that the playback
actually terminates.

The text field contains the default audio file
name, junk.au, when the GUI first appears on the
screen.

The program displays the format of the audio
data in the file before playing the file.  The
format is displayed on the command- line screen.

Tested using SDK 1.4.1 under Win2000
*********************************************/

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

public class AudioPlayer02 extends JFrame{

  AudioFormat audioFormat;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;
  boolean stopPlayback = false;
  final JButton stopBtn = new JButton("Stop");
  final JButton playBtn = new JButton("Play");
  final JTextField textField =
                       new JTextField("junk.au");

  public static void main(String args[]){
    new AudioPlayer02();
  }//end main
```

```
  //---------------------------------------//

  public AudioPlayer02(){//constructor

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

    //Instantiate and register action listeners
    // on the Play and Stop buttons.
    playBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                                  ActionEvent e){
          stopBtn.setEnabled(true);
          playBtn.setEnabled(false);
          playAudio();//Play the file
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

    stopBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                                  ActionEvent e){
          //Terminate playback before EOF
          stopPlayback = true;
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

    getContentPane().add(playBtn,"West");
    getContentPane().add(stopBtn,"East");
    getContentPane().add(textField,"North");

    setTitle("Copyright 2003, R.G.Baldwin");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(250,70);
    setVisible(true);
  }//end constructor
  //---------------------------------------//

  //This method plays back audio data from an
  // audio file whose name is specified in the
  // text field.
  private void playAudio() {
    try{
      File soundFile =
                   new File(textField.getText());
      audioInputStream = AudioSystem.
                  getAudioInputStream(soundFile);
      audioFormat = audioInputStream.getFormat();
      System.out.println(audioFormat);

      DataLine.Info dataLineInfo =
                          new DataLine.Info(
                            SourceDataLine.class,
```

```
                                      audioFormat);

      sourceDataLine =
             (SourceDataLine)AudioSystem.getLine(
                                      dataLineInfo);

      //Create a thread to play back the data and
      // start it running.  It will run until the
      // end of file, or the Stop button is
      // clicked, whichever occurs first.
      // Because of the data buffers involved,
      // there will normally be a delay between
      // the click on the Stop button and the
      // actual termination of playback.
      new PlayThread().start();
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end playAudio


//===========================================//
//Inner class to play back the data from the
// audio file.
class PlayThread extends Thread{
  byte tempBuffer[] = new byte[10000];

  public void run(){
    try{
      sourceDataLine.open(audioFormat);
      sourceDataLine.start();

      int cnt;
      //Keep looping until the input read method
      // returns -1 for empty stream or the
      // user clicks the Stop button causing
      // stopPlayback to switch from false to
      // true.
      while((cnt = audioInputStream.read(
           tempBuffer,0,tempBuffer.length)) != -1
                      && stopPlayback == false){
        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();

      //Prepare to playback another file
```

```
      stopBtn.setEnabled(false);
      playBtn.setEnabled(true);
      stopPlayback = false;
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end run
}//end inner class PlayThread
//=================================//

}//end outer class AudioPlayer02.java
```

Listing 17

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-