Java Sound, Using Audio Line Events

Baldwin shows you how to use audio line events. You can use this approach to register listeners and to receive notifications whenever an audio line is opened, started, stopped, or closed.

Published: April 15, 2003
By Richard G. Baldwin

Java Programming Notes # 2018

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, Playing Back Audio Files using Java.

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 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.

Earlier lessons in the series showed you how to:

- Capture microphone data into audio files types of your own choosing.
- Capture microphone data into a **ByteArrayOutputStream** object, and how to use the Sound API to play back previously captured audio data.
- Identify the mixers available on your system, and how to specify a particular mixer for use in the acquisition of audio data from a microphone.
- Understand the use of lines and mixers in the Java Sound API.

In this lesson, I will teach you how to use events fired by audio lines to synchronize audio activities with other activities.

Discussion and Sample Code

Synchronizing other activities with sound

Suppose you need to play some music, and to cause an animation to start and stop in synchronism with the beginning and the ending of the music. How would you synchronize those two activities?

There are probably several ways to accomplish this task. One way would be to use the events fired by the audio line to synchronize the beginning and the ending of the animation.

What is an audio line event?

Audio lines fire events when they are opened, started, stopped, or closed. You can instantiate listener objects and register them on the **Line** object to be notified each time the line fires an event.

You can define event handler methods belonging to the listener objects to perform whatever action you need to perform each time the line fires an event. I will provide a sample program and explain how to do this in this lesson.

The user interface

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

- A Capture button
- A Stop button
- A Playback button

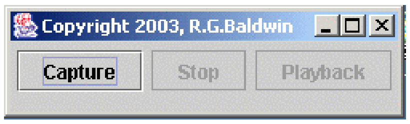

Figure 1 Program GUI

Operation of the program

The program demonstrates the capture and subsequent playback of audio data, and also demonstrates the instantiation, registration, and operation of line event listeners as well. The event listeners display messages on the screen when the various audio line events occur.

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.

Event handler output is displayed

Line events are fired each time the user clicks one of the buttons and causes the line to be opened, started, stopped, or closed. The registered event handler methods display information on the screen about the event and the line each time such an event is fired. I will provide sample screen output at the appropriate points in the discussion.

Will discuss in fragments

As usual, I will discuss this program in fragments. A complete listing of the program is shown in Listing 16 near the end of the lesson.

Updated version of a previously-discussed program

The program that I will discuss in this lesson is an updated version of the program that I discussed in the following lessons:

- Java Sound, Getting Started, Part 1, Playback
- Java Sound, Getting Started, Part 2, Capture using Specified Mixer

I will discuss the entire program very briefly to establish the context. However, I will concentrate my detailed discussion on those aspects of the program that were updated to support audio line event handling.

> *(I strongly recommend that you refer back to the two lessons listed above for the detailed discussion of those parts of the program that don't involve event handling.)*

The program named AudioEvents01

The program named **AudioEvents01** demonstrates the use of a Java program to:

- Capture audio data from a microphone into a **ByteArrayOutputStream** object.
- Register an event listener object on the line used to capture the data.
- Handle an event each time the capture line is opened, started, stopped, or closed.
- Play back the data stored in the **ByteArrayOutputStream** object.
- Register an event listener object on the line used to play back the captured data.
- Handle an event each time the playback line is opened, started, stopped, or closed.

Behavior of the event handler methods

The behavior of the event handler methods is very simple in this sample program. The behavior is provided solely for purposes of illustration. You are reminded that the behavior of the event handler methods can be as simple or as complex as your needs may dictate. The important point is that the event handler methods are invoked when the events are fired. You can design those methods to provide whatever behavior is appropriate for your situation.

Should you spawn a new thread?

As in all forms of event handling in Java, if the task to be performed by your event handler method will require a significant amount of time to complete, you should probably spawn a new thread to carry out that task and cause your event handler method to return as soon as possible.

> *(I didn't do that in this program due to the simplicity and speed of the behavior of my event handling methods.)*

The controlling class named AudioEvents01

The class definition for the controlling class begins in Listing 1.

```
public class AudioEvents01 extends
JFrame{

  boolean stopCapture = false;
  ByteArrayOutputStream
byteArrayOutputStream;
```

```
  AudioFormat audioFormat;
  TargetDataLine targetDataLine;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;

  public static void main(String
args[]){
    new AudioEvents01();
  }//end main

Listing 1
```

The code in Listing 1 includes the declaration of some instance variables and the **main** method. The behavior of the **main** method is simply to instantiate a new object of the controlling class.

The constructor

The constructor for the controlling class begins in Listing 2.

```
  public AudioEvents01(){//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);

Listing 2
```

The code in Listing 2 creates the button objects shown in Figure 1, and sets the initial enabled and disabled properties of those buttons.

Action event handlers

The code in Listing 3 instantiates three **ActionListener** objects and registers each of those listener objects on one of the three buttons shown in the GUI in Figure 1.

```
    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);
```

Listing 3

Code very similar to that shown in Listing 3 was discussed in detail in the two lessons listed earlier. Therefore, I won't repeat that discussion here.

Complete the GUI

The code in Listing 4 takes care of a few more details required to complete the GUI and make it visible.

```
    getContentPane().setLayout(new
FlowLayout());
    setTitle("Copyright 2003,
R.G.Baldwin");

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

Listing 4

The code in Listing 4 signals the end of the constructor.

The captureAudio method

If you refer back to Listing 3, you will see that the event handler on the **Capture** button invokes the method named **captureAudio** to cause the actual data capture operation to be carried out. The method named **captureAudio** captures audio input from a microphone and saves it in a **ByteArrayOutputStream** object.

The code for the method named **captureAudio** begins in Listing 5

```
  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);
```

Listing 5

Nothing new so far

So far, there is still nothing new to discuss. The code in Listing 5 was discussed in detail in the two lessons listed earlier, so I won't repeat that discussion here.

The main thing to note in Listing 5 is the creation of an object of type **Line**, and the storage that object's reference in an instance variable of type **TargetDataLine** named **targetDataLine.**

The open, start, stop, and close methods

The previous lessons have discussed the methods of the **TargetDataLine** object named **open start**, **stop**, and **close** in detail. However, here is some information from Sun that I did not discuss in those lessons.

With respect to the **open** method, Sun tells us,

> *"If this operation succeeds, the line is marked as open, and an OPEN event is dispatched to the line's listeners."*

Similarly, with respect to the **start** method, Sun tells us,

> *"When audio capture or playback starts, a START event is generated."*

As you have probably guessed by now, Sun has this to say about the **stop** method.

> *"When audio capture or playback stops, a STOP event is generated."*

Finally, Sun has this to say about the **close** method.

> *"If this operation succeeds, the line is marked closed and a CLOSE event is dispatched to the line's listeners."*

Event registration methods

If you are familiar with Java event handling in general, and JavaBeans design patterns in particular, you have probably already predicted that the **TargetDataLine** object provides the following event registration methods:

- addLineListener(LineListener listener)
- removeLineListener(LineListener listener)

The addLineListener method

Here is what Sun has to say about the **addLineListener** method.

> *"Adds a listener to this line. Whenever the line's status changes, the listener's* ***update****() method is called with a* ***LineEvent*** *object that describes the change."*

Similarly, the **removeLineListener** method

> *"Removes the specified listener from this line's list of listeners."*

The LineListener interface

Note that both of the registration methods require an incoming parameter of the **LineListener** type. Stated differently, in each case, the incoming parameter must be a reference to an object instantiated from a class that implements the **LineListener** interface.

Sun has this to say about the **LineListener** interface.

> *"Instances of classes that implement the **LineListener** interface can register to receive events when a line's status changes."*

The **LineListener** interface declares a single method named **update**, which receives an incoming parameter of type **LineEvent**.

The update method

Every class that implements the **LineListener** interface must provide a concrete definition of the update method. Sun has this to say about that method.

> *"Informs the listener that a line's state has changed. The listener can then invoke **LineEvent** methods to obtain information about the event."*

In other words, whenever a **LineEvent** occurs, the **Line** object notifies all registered listeners by invoking the **update** method on each registered listener object, passing a **LineEvent** object's reference as a parameter to the **update** method. The **LineEvent** object encapsulates information about the event.

The LineEvent class

That brings us to the crux of the matter involving audio line event firing and handling. Sun has this to say about the **LineEvent** class.

> *"The **LineEvent** class encapsulates information that a line sends its listeners whenever the line opens, closes, starts, or stops. Each of these four state changes is represented by a corresponding type of event. A listener receives the event as a parameter to its update method. By querying the event, the listener can learn the type of event, the line responsible for the event, and how much data the line had processed when the event occurred."*

The LineEvent methods

As of Java SDK 1.4.1, an object of the **LineEvent** class provides the following methods that an event handler can use to obtain information about the event:

- **getFramePosition** - Returns the position of the line's audio data when the event occurred, expressed in sample frames.
- **getLine** - Returns a reference to the **Line** object that fired the event.
- **getType** - Returns the type of the event *(open, start, stop, or close)* as **LineEvent.Type**.
- **toString** - Returns a string representation of the event.

Register a line listener on the TargetDataLine object

Finally, we are going to see the above discussion rendered in code. The somewhat cryptic code in Listing 6 instantiates an anonymous listener object from an anonymous class that implements the **LineListener** interface, and registers that listener object on the **TargetDataLine** object that was created in Listing 5.

```
      targetDataLine.addLineListener(
        new LineListener(){

          public void update(LineEvent
e){
            System.out.println(
             "Event handler for
TargetDataLine");
            System.out.println(
                   "Event type: " +
e.getType());
            System.out.println("Line
info: " +

e.getLine().getLineInfo());

System.out.println();//blank line
          }//end update

        }//end LineListener
      );//end addLineListener()

Listing 6
```

The update method

The code in Listing 6 defines the **update** method that will be invoked on the registered listener object each time a line event occurs.

The code in the **update** method begins by printing **"Event handler for TargetDataLine"** on the screen.

Then it invokes the **getType** method to get and display the type of the line event.

Following that, it invokes the **getLine** method to get a reference to the **Line** object that fired the event. It uses that reference to invoke the **getLineInfo** method on the line and display information about the line.

Sample screen output

Figure 2 shows the screen output that appears following a click on the **Capture** button on my system. *(The line info output on your system may be different.)*

> *(Note that line breaks were manually inserted in Figure 2 to accommodate this narrow format. The boldface was also manually added for emphasis.)*

```
Event handler for TargetDataLine
Event type: Open
Line info: interface TargetDataLine
supporting
 64 audio formats

Event handler for TargetDataLine
Event type: Start
Line info: interface TargetDataLine
supporting
 64 audio formats
```

Figure 2

Later on, when we examine the code that is used to capture the data, you will see the statements that open and start the line, causing the **OPEN** and **START** events shown in Figure 2 to be fired.

Create and start a thread to capture the audio data

The code in Listing 7 instantiates a new thread object and starts it running. The purpose of this thread is to perform the actual data capture. The **run** method of the thread will continue to capture audio data from the microphone until the **Stop** button is clicked by the user.

```
      new CaptureThread().start();
    }catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end captureAudio method
```

Listing 7

Once the thread is running to capture the data, the **captureAudio** method returns control to the event handler on the **Capture** button, which terminates shortly thereafter. This frees up the event-handling thread to handle an event when the **Stop** button is clicked.

The CaptureThread class

At this point, I am going to discuss an inner class, an object of which is used to capture data from the microphone. Once again, this class is very similar to one that was discussed in detail in the two lessons listed earlier. Therefore, this discussion will be very brief, emphasizing only those aspects of the code in the class that is important to audio line event handling.

The beginning of the class named **CaptureThread** and its **run** method is shown in Listing 8.

```
class CaptureThread extends Thread{
  //An arbitrary-size temporary
holding buffer
  byte tempBuffer[] = new byte[10000];
  public void run(){

    byteArrayOutputStream =
                     new
ByteArrayOutputStream();
    stopCapture = false;
    try{
```

Listing 8

If you have studied the two lessons listed previously, there is nothing new or exciting about the code in Listing 8, so I won't discuss it further. I show it here simply to establish the context for the discussion that follows.

Open and start the line

The code in Listing 9 isn't new, but it is very interesting in the context of this lesson.

```

targetDataLine.open(audioFormat);
      targetDataLine.start();
```

Listing 9

The code in Listing 9 first invokes the **open** method on the **TargetDataLine** object. Following that, it invokes the **start** method on the **TargetDataLine** object.

I discussed these two methods in detail in previous lessons. In addition, I told you earlier in this lesson that the invocation of these methods causes the **OPEN** and **START** events to be fired. Therefore, it is the invocation of these two methods that causes the **update** method registered on the **TargetDataLine** object to be invoked. This, in turn, produces the screen output shown in Figure 2.

Behavior of the update method

Once again, the behavior that I designed into my version of the **update** method is very simple. It just displays some information about the event and the line that fired it.

However, the behavior that you design into your version of the **update** method can be as simple or as complex as your needs may dictate.

Loop until stopCapture variable goes true

Continuing with the **run** method of the **CaptureThread** class, the code in Listing 10 loops until the value of the variable named **stopCapture** changes from false to true. *(This happens when the user clicks 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();
```

Listing 10

During this period, the code in Listing 10 continues to capture audio data from the microphone and to store that data in the **ByteArrayOutputStream** object. *(I discussed the operation of this **while** loop in detail in the previous lessons.)*

Stop and close the TargetDataLine object

When the **while** loop terminates, the two statements in Listing 11 invoke the **stop** and **close** methods, respectively, on the **TargetDataLine** object.

```
      targetDataLine.stop();
      targetDataLine.close();
```

Listing 11

Fire STOP and CLOSE events

As explained earlier, this causes the **TargetDataLine** object to fire **STOP** and **CLOSE** events respectively. This, in turn causes the **update** method in the listener object to be invoked twice in succession, producing the screen output shown in Figure 3.

```
Event handler for TargetDataLine
Event type: Stop
Line info: interface TargetDataLine
supporting
 64 audio formats

Event handler for TargetDataLine
Event type: Close
Line info: interface TargetDataLine
supporting
 64 audio formats
```

Figure 3

The end of the run method

Except for a required **catch** block, the code in Listing 11 signals the end of the **run** method of the **CaptureThread** class. That code also signals the end of the class as well.

Finally, the code in Listing 11 signals the end of the discussion of the data capture portion of this program.

The playback portion of the program

With respect to audio line event handling, there is very little difference between the playback portion of this program and the data capture portion discussed above.

I discussed the event handling aspects of the data capture portion of the program in detail in the preceding paragraphs. I discussed the other aspects of the playback portion of the program in detail in the lessons listed earlier. Therefore, in the remainder of this lesson, I will discuss the event handling aspects of the playback portion of the program only briefly. As mentioned earlier, you can view a complete listing of the program in Listing 16 near the end of the lesson.

The playAudio method

This program uses a method named **playAudio** to play back the data that was captured and saved in a **ByteArrayOutputStream** object.

Get a SourceDataLine object

The capture portion of the program uses a **TargetDataLine** object to capture microphone data in real time.

Similarly, the playback portion uses a **SourceDataLine** object to deliver the audio data to the speakers in real time.

Much of the early code in the **playAudio** method was deleted from Listing 12 for brevity. The first statement in Listing 12 gets a **SourceDataLine** object's reference and assigns it to a reference variable named **sourceDataLine**.

```
// Code deleted from playAudio method
for brevity

      sourceDataLine =

(SourceDataLine)AudioSystem.getLine(

dataLineInfo);


      //Register a line listener on
the
      // SourceDataLine object
      sourceDataLine.addLineListener(
        new LineListener(){
          public void update(LineEvent
e){
            System.out.println(
             "Event handler for
SourceDataLine");
            System.out.println(
                   "Event type: " +
e.getType());
            System.out.println("Line
info: "
                    +
e.getLine().getLineInfo());

System.out.println();//blank line
          }//end update
        }//end LineListener
      );//end addLineListener()

      //Create a thread to play back
the data and
      // start it running.
      new PlayThread().start();

Listing 12
```

Register a line listener on the SourceDataLine object

Following that, the code in Listing 12 instantiates a **LineListener** object and registers it on the **SourceDataLine** object. The definition and behavior of the line listener instantiated in Listing 12 is essentially the same as that shown in Listing 6 earlier.

Start a playback thread running

Then the code in Listing 12 instantiates a playback thread object and starts it running. The **run** method of the playback thread will continue running until the audio data previously stored in the **ByteArrayOutputStream** object is exhausted.

The PlayThread class

The playback thread is instantiated from the **PlayThread** class, which begins in Listing 13.

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

  public void run(){
    try{
      int cnt;


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

Listing 13
```

The code in Listing 13 invokes the **open** and **start** methods on the **SourceDataLine** object, causing **OPEN** and **START** events to be fired.

The screen output

These events are handled by the **update** event handler method defined in Listing 12. This causes the output shown in Figure 4 to appear on the screen.

```
Event handler for SourceDataLine
Event type: Open
Line info: interface SourceDataLine
supporting
 8 audio formats

Event handler for SourceDataLine
Event type: Start
Line info: interface SourceDataLine
supporting
 8 audio formats

Figure 4
```

*(Note that the Line info regarding the **SourceDataLine** in Figure 4 is a little different from the similar information regarding the **TargetDataLine** in Figure 2. Information for your system may be different from that shown for my system.)*

The playback loop

As is the case for the data capture portion of this program, the playback portion uses a **while** loop to transfer data from the **ByteArrayOutputStream** object to the internal buffer of the **SourceDataLine** object.

```
      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
```

Listing 14

The **SourceDataLine** object delivers that audio data in real time to the speakers on the computer.

The data transfer from the **ByteArrayOutputStream** object to the **SourceDataLine** object continues until the data stored in the **ByteArrayOutputStream** object is exhausted, at which time the **read** method in Listing 14 returns -1.

The drain method

The **while** loop terminates when the data in the **ByteArrayOutputStream** object is exhausted. However, the program must not terminate at that point in time. In all likelihood, there is data still remaining in the internal buffer of the **SourceDataLine** object that needs to be sent to the speakers in real time. That is the purpose of the **drain** method that is invoked in Listing 15.

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

Listing 15
```

The **drain** method blocks until the internal buffer of the **SourceDataLine** object becomes empty, at which time it returns.

No stop method is invoked

Note that in this case, I did not explicitly invoke the **stop** method, but a **STOP** event was fired anyway. Apparently invoking the **close** method on a line that has been drained causes both a **STOP** event and a **CLOSE** event to be fired. The screen output produced by the code in Listing 15 is shown in Figure 5.

```
Event handler for SourceDataLine
Event type: Stop
Line info: interface SourceDataLine
supporting
 8 audio formats

Event handler for SourceDataLine
Event type: Close
Line info: interface SourceDataLine
supporting
 8 audio formats

Figure 5
```

The getAudioFormat method

There is one additional method that I haven't discussed in this lesson. The method named **getAudioFormat** is identical to the method with the same name used in the earlier version of the program. I explained the behavior of that method in detail in the lessons listed earlier. Therefore, I won't discuss this method in this lesson. You can view it in Listing 16 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 16 near the end of the lesson.

Capture the data

Start the program and click the **Capture** button. Talk into the microphone for a short period of time and then click the **Stop** button to terminate data capture. While you are doing this, observe the output on the command-line screen.

(Be careful and don't attempt to capture too much audio data. The data is being captured in memory, and if you attempt to capture too much data, you may run out of memory.)

Play the captured data back

Click the **Playback** button to cause the captured data to be played back through the speakers on your computer. Again, observe the output on the command-line screen while you are doing this.

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.

In case of a runtime error

If you get a runtime error while attempting to capture the audio data, see the comments in the **getAudioFormat** method in Listing 16. You may need to try using a different audio format. I have received feedback from some readers who tell me that the format that I used in this program doesn't work on all systems.

Summary

In this lesson, I have presented and explained a program that demonstrates the use of audio line events. You can use this approach to register listeners and to receive notifications whenever an audio line is opened, started, stopped, or closed. Once you receive the notification, you can learn the type of event, the line responsible for the event, information about that line, and how much data the line had processed when the event occurred.

Complete Program Listing

A complete listing of the program is shown in Listing 16.

```
/*File AudioEvents01.java
The main purpose of this program is to
demonstrate audio line event handling.

This program demonstrates the capture and
subsequent playback of audio data, and
demonstrates the instantiation and registration
of line event listeners as well.  The event
listeners display messages on the screen when
the various audio line events occur.

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.

Following is the screen output following the
click on the Capture button.  Note that line
breaks were manually inserted in this, and the
other output material shown below, to cause the
material to fit this narrow format.

Event handler for TargetDataLine
Event type: Open
Line info: interface TargetDataLine supporting
 64 audio formats

Event handler for TargetDataLine
Event type: Start
Line info: interface TargetDataLine supporting
 64 audio formats



Following is the screen output following the
click on the Stop button.

Event handler for TargetDataLine
Event type: Stop
Line info: interface TargetDataLine supporting
 64 audio formats

Event handler for TargetDataLine
Event type: Close
Line info: interface TargetDataLine supporting
 64 audio formats



Following is the screen output following the
click on the Playback button.

Event handler for SourceDataLine
Event type: Open
Line info: interface SourceDataLine supporting
 8 audio formats

Event handler for SourceDataLine
Event type: Start
Line info: interface SourceDataLine supporting
 8 audio formats

Event handler for SourceDataLine
Event type: Stop
Line info: interface SourceDataLine supporting
 8 audio formats

Event handler for SourceDataLine
Event type: Close
Line info: interface SourceDataLine supporting
 8 audio formats

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 AudioEvents01 extends JFrame{

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

  public static void main(String args[]){
    new AudioEvents01();
  }//end main

  public AudioEvents01(){//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("Copyright 2003, R.G.Baldwin");
    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);
      //Register a line listener on the
      // TargetDataLine object
      targetDataLine.addLineListener(
        new LineListener(){
          public void update(LineEvent e){
            System.out.println(
             "Event handler for TargetDataLine");
            System.out.println(
                   "Event type: " + e.getType());
            System.out.println("Line info: " +
                      e.getLine().getLineInfo());
            System.out.println();//blank line
          }//end update
        }//end LineListener
      );//end addLineListener()

      //Create a thread to capture the
      // microphone data and start it running. It
      // will run until the Stop button is
      // clicked.
      new 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);


      //Register a line listener on the
      // SourceDataLine object
      sourceDataLine.addLineListener(
        new LineListener(){
          public void update(LineEvent e){
            System.out.println(
             "Event handler for SourceDataLine");
            System.out.println(
                   "Event type: " + e.getType());
            System.out.println("Line info: "
                    + e.getLine().getLineInfo());
            System.out.println();//blank line
          }//end update
        }//end LineListener
      );//end addLineListener()

      //Create a thread to play back the data and
      // start it running.  It will run until all
      // the data has been played back, at which
      // time it will automatically stop the
      // line and fire a Stop event.
      new 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{
      targetDataLine.open(audioFormat);
      targetDataLine.start();

      //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();

      targetDataLine.stop();
      targetDataLine.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;

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

      //Loop 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 become empty.  When it
      // becomes empty, it will fire a Stop
      // event and return.
      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 AudioEvents01.java
```

Listing 16

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-