Java Sound, Capturing Microphone Data into an Audio File

Baldwin shows you how to use the Java Sound API to capture audio data from a microphone and how to save that data in an audio file type of your choosing.

Published: March 18, 2003
By Richard G. Baldwin

Java Programming Notes # 2014

- 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, Getting Started, Part 2, Capture Using Specified Mixer.

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

In this lesson, I will provide and explain a program that you can use to capture audio data from a microphone and to write that data into an audio file type of your choosing. You should then be able to play the audio file back using any of a variety of readily available media players, such as the Windows Media Player.

I will show you how to write a Java program to play back the data stored in the audio file in a future lesson.

Discussion and Sample Code

Less complicated program

The previous two lessons showed you how to capture audio data from a microphone, store it in a memory buffer, and play the data back on demand. As you will see in this lesson, because of the relatively high-level support for writing audio files in Java, the code required to store the captured data in an audio file is somewhat simpler than the code required to store the data in a memory buffer.

What is a TargetDataLine?

I will be using a **TargetDataLine** object in this program. The terminology used with the Java sound API can be very confusing. A **TargetDataLine** object is a *streaming* mixer output object.

> *(The object provides output from the mixer, not output from the program. In fact, it often serves as input to the program.)*

An object of this type delivers audio data from the mixer, serving as input to other parts of the program. This concept is discussed in detail in the lesson entitled Java Sound, Getting Started, Part 2, Capture Using Specified Mixer.

Audio data input to the program

The data provided by the **TargetDataLine** object can be pushed into some other program construct in real time. The actual destination of the audio data can be any of a variety of destinations such as an audio file, a network connection, or a buffer in memory.

> *(A sample program in this lesson reads audio data from a TargetDataLine object and writes it into an audio output file.)*

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 two regular buttons:

- Capture
- Stop

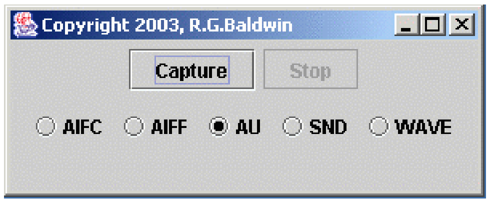

Figure 1 Program GUI

In addition, the GUI contains five radio buttons labeled:

- AIFC
- AIFF
- AU
- SND
- WAVE

These are five audio file format types supported by the Java SDK, version 1.4.1. I will have more to say about these file types later when I discuss the code.

Operation of the program

When the user clicks the **Capture** button, input data from a microphone is captured and saved in an audio output file of the type specified by the selected radio button. *(Clicking the Capture button also enables the Stop button and disables the Capture button.)*

When the user clicks the **Stop** button *(while recording data)*, data capture is terminated and the output file is closed. The file is then suitable for playing back using any standard media player that will accommodate the specified file format.

Will discuss in fragments

As usual, I will discuss this program in fragments. A complete listing of the program is shown in Listing 22 near the end of the lesson.

The class named AudioRecorder02

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

```
public class AudioRecorder02 extends
JFrame{

  AudioFormat audioFormat;
  TargetDataLine targetDataLine;
```

Listing 1

This class declares several instance variables, which are used later in the program. The two instance variables shown in Listing 1 are used later to hold references to **AudioFormat** and **TargetDataLine** objects. I will discuss those objects in more detail when we get to the point in the program where they are used.

The GUI buttons

As you saw in Figure 1, the GUI presents two ordinary buttons, which are used to start and stop the audio data capture operation. The instance variables in Listing 2 are used to hold references to **JButton** objects used for this purpose.

```
  final JButton captureBtn =
                          new
JButton("Capture");
  final JButton stopBtn = new
JButton("Stop");
```

Listing 2

The radio buttons

Also, as you saw in Figure 1, the GUI presents five radio buttons. Using radio buttons is a little more complicated than using ordinary buttons.

The normal user expectation for the behavior of radio buttons is that they will be logically grouped into mutually-exclusive groups *(only one button in a group can be selected at any point in time)*. This mutually-exclusive behavior is achieved in Java Swing by adding **JRadioButton** objects to a **ButtonGroup** object.

(A somewhat different approach is used to group radio buttons in the Java AWT.)

Not a physical group

Note, however, that adding the buttons to the **ButtonGroup** object does not create a physical grouping. Adding the radio buttons to the **ButtonGroup** object simply creates a mutually-exclusive logical grouping.

The normal user expectation is also that a mutually-exclusive group of radio buttons will be physically grouped together in the GUI. This physical grouping requires the use of a container of some type, such as a **JPanel** for example.

The instance variables in Listing 3 hold references to:

- One **JPanel** object
- One **ButtonGroup** object
- Five **JRadioButton** objects

```
  final JPanel btnPanel = new
JPanel();

  final ButtonGroup btnGroup = new
ButtonGroup();

  final JRadioButton aifcBtn =
                        new
JRadioButton("AIFC");
  final JRadioButton aiffBtn =
                        new
JRadioButton("AIFF");
  final JRadioButton auBtn =//selected
at startup
                     new
JRadioButton("AU",true);
  final JRadioButton sndBtn =
                         new
JRadioButton("SND");
  final JRadioButton waveBtn =
                        new
JRadioButton("WAVE");

Listing 3
```

Constructing radio buttons

The constructor used for four of the **JRadioButton** objects in Listing 3 allows the button's label to be specified when the object is instantiated. The constructor used for the button referred to by **auBtn** not only allows for the label to be provided, but also allows for a **boolean** parameter that causes the button to be ***selected*** when it appears first on the screen.

We will see the code later that uses these instance variables to create a logical group and a physical group of mutually-exclusive radio buttons.

The main method

The **main** method for this Java application, shown in Listing 4, is extremely simple.

```
  public static void main( String
args[]){
    new AudioRecorder02();
  }//end main

Listing 4
```

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 takes over at that point and provides the operational behavior of the program.

The constructor

The constructor, which begins in Listing 5, is fairly long. I will break it up and discuss it in fragments.

```
  public
AudioRecorder02(){//constructor
    captureBtn.setEnabled(true);
    stopBtn.setEnabled(false);

Listing 5
```

When the program first starts running, the **Capture** button is enabled, and the **Stop** button is disabled, as shown in Figure 1. As you can see in Listing 5, this is accomplished by initializing the **enabled** property on each of the two buttons to values of **true** and **false** respectively.

We will see later that when the user clicks the **Capture** button to cause data capture to begin, the values of the **enabled** property are switched to cause the **Capture** button to become disabled, and to cause the **Stop** button to become enabled.

An anonymous listener from an anonymous inner class

You may, or may not be familiar with the rather cryptic programming style shown in Listing 6. The code in Listing 6 instantiates an **ActionListener** object and registers it on the **Capture** button.

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

ActionEvent e){

captureBtn.setEnabled(false);
          stopBtn.setEnabled(true);
          //Capture input data from
the
```

```
          // microphone until the Stop
button is
          // clicked.
          captureAudio();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

Listing 6
```

If you are already familiar with this programming style, just note that the **actionPerformed** method, which is invoked when the user clicks the **Capture** button,

- Switches the enabled properties of the two buttons.
- Invokes the **captureAudio** method to cause the audio capture process to begin.

A cryptic programming style

If you are not familiar with this programming style, think of it this way. The code in Listing 6 instantiates an object from an unnamed class, *(which implements the **ActionListener** interface),* and passes that object's reference to the **addActionListener** method of the **Capture** button to register the object as a listener on the button. Whenever the user clicks the **Capture** button, the **actionPerformed** method belonging to the listener object is invoked, behaving as described above.

Another anonymous listener

The code in Listing 7 uses the same cryptic syntax to instantiate and register an **ActionListener** object on the **Stop** button.

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

ActionEvent e){
          captureBtn.setEnabled(true);
          stopBtn.setEnabled(false);
          //Terminate the capturing of
input data
          // from the microphone.
          targetDataLine.stop();
          targetDataLine.close();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

Listing 7
```

The behavior of the **actionPerformed** event-handler method in this case is to **stop** and **close** the **TargetDataLine** object.

> *(Recall that the **TargetDataLine** object is the object that acquires data from the microphone and delivers it to the program. We will see how that is accomplished later.)*

The stop method

Here is what Sun has to say about invoking the **stop** method on a **TargetDataLine** object.

> *"Stops the line. A stopped line should cease I/O activity. If the line is open and running, however, it should retain the resources required to resume activity. A stopped line should retain any audio data in its buffer instead of discarding it, so that upon resumption the I/O can continue where it left off, if possible. ... If desired, the retained data can be discarded by invoking the `flush` method."*

Sun's description seems to be fairly self-explanatory.

It is worth noting that this is the method call that causes the output **write** method *(to be discussed later)* to close the output file.

The close method

Here is what Sun has to say about invoking the **close** method on a **TargetDataLine** object.

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

This also seems to be fairly self-explanatory. I will have more to say about the impact of invoking the **stop** and **close** methods later when I discuss the **write** method used to write the data to the output audio file.

Put the buttons in the JFrame

The two statements in Listing 8 cause the **Capture** button and the **Stop** button to be placed in the **JFrame** object. This is straightforward GUI construction code. If you are unfamiliar with it, you can read about it on my web site.

```
    getContentPane().add(captureBtn);
    getContentPane().add(stopBtn);
```

Listing 8

Include the radio buttons in a mutually-exclusive group

The code in Listing 9 causes the five radio buttons to be included in a mutually-exclusive logical group.

```
    btnGroup.add(aifcBtn);
    btnGroup.add(aiffBtn);
    btnGroup.add(auBtn);
    btnGroup.add(sndBtn);
    btnGroup.add(waveBtn);
```

Listing 9

The ButtonGroup class

The variable named **btnGroup** holds a reference to an object of the **ButtonGroup** class. Here is part of what Sun has to say about the **ButtonGroup** class in general.

> *"This class is used to create a multiple-exclusion scope for a set of buttons. Creating a set of buttons with the same ButtonGroup object means that turning "on" one of those buttons turns off all other buttons in the group.*
>
> *Initially, all buttons in the group are unselected. Once any button is selected, one button is always selected in the group."*

As a clarification to the above, it is possible to cause a particular button to be initially *"on"* using the constructor shown for the **AU** button in Listing 3. When this program starts, the radio button labeled **AU** is initially *"on"*.

The use of the **add** method shown in Listing 9 causes each button to be included in the group.

It is important to note that simply adding buttons to the **ButtonGroup** object does not cause them to be physically grouped in the GUI. Additional effort is required to cause the buttons to be grouped in a physical sense.

Adding the radio buttons to the JPanel

Physical grouping of the radio buttons is accomplished by the code in Listing 10, which adds the radio buttons to a **JPanel** container object.

```
    btnPanel.add(aifcBtn);
    btnPanel.add(aiffBtn);
    btnPanel.add(auBtn);
    btnPanel.add(sndBtn);
    btnPanel.add(waveBtn);
```

Listing 10

The default layout manager for a **JPanel** object is **FlowLayout**. This causes the radio buttons to appear in the panel in a row from left to right as shown in Figure 1. Because the panel is transparent and doesn't have a visible border, only the buttons are visible in Figure 1. The underlying panel is not visible.

Add the JPanel to the JFrame

The code in Listing 11 causes the panel containing the radio buttons to be placed in the **JFrame** GUI object as shown in Figure 1.

```
    getContentPane().add(btnPanel);
```

Listing 11

Finish the GUI and make it visible

The code in Listing 12 takes care of a few remaining odds and ends regarding the GUI.

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

setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(300,120);
    setVisible(true);
  }//end constructor
```

Listing 12

The code in Listing 12 accomplishes the following tasks:

- Set the layout manager on the **JFrame** GUI object to **FlowLayout**. This causes the components to be added to the container from left to right, top to bottom. *(Remember that the **JPanel** is itself a component that contains the radio buttons, so the panel appears as a single component in the layout.)*
- Set the title on the **JFrame** GUI object.
- Cause the button with the X in the upper right corner of the **JFrame** object to terminate the program when it is clicked.
- Set the size of the **JFrame** object to 300 pixels by 120 pixels.
- Make the whole thing visible on the screen.

The code in Listing 12 also ends the constructor.

Capture the audio data from the microphone

Recall that the event handler on the **Capture** button invokes the method named **captureAudio** to cause the data capture process to start. The beginning of the **captureAudio** method is shown in Listing 13.

```
  private void captureAudio(){
    try{
      audioFormat = getAudioFormat();
```

Listing 13

The purpose of the **captureAudio** method is to capture audio input data from a microphone and to cause the data to be saved in an audio file of the type specified by the selected radio button in Figure 1.

Establishing the audio data format

The first step in the capture process is to establish the format of the captured audio data.

The format of the audio data is not the same thing as the format of the audio file. The format of the audio data establishes how the bits and bytes will be used to represent the values of the audio data. The format of the audio file establishes how those bits and bytes will be saved in a physical file on the disk.

While it is probably not true that every file format can accommodate every data format, it is true that some data formats can be accommodated by more than one file format.

(I plan to publish a future lesson that deals with audio formats and file formats in detail.)

The getAudioFormat method

The code in Listing 13 invokes the **getAudioFormat** method in order to establish the audio format. I'm going to set the discussion of the **captureAudio** method aside momentarily and discuss the **getAudioFormat** method. I will return to a discussion of the **captureAudio** method after I explain the **getAudioFormat** method.

The entire **getAudioFormat** method is shown in Listing 14.

```
  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
```

Listing 14

The **getAudioFormat** method creates and returns an **AudioFormat** object for a given set of format parameters.

> ***May not work for you***
> *(Not all audio formats are supported on all systems. 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.)*

An AudioFormat object

As I mentioned earlier, I plan to publish a future tutorial lesson that explains audio formats in detail. Therefore, this discussion will be very brief.

As you can see, the code in Listing 14 instantiates and returns an object of the **AudioFormat** class. The constructor used in Listing 14 to instantiate the object requires the following parameters:

- sampleRate - The number of samples that will be acquired each second for each channel of audio data.
- sampleSizeInBits - The number of bits that will be used to describe the value of each audio sample.
- channels - Two channels for stereo, and one channel for mono.
- signed - Whether the description of each audio sample consists of both positive and negative values, or positive values only.
- bigEndian - See a discussion of this topic here.

My format

As you can see in Listing 14, my version of the programs samples a single channel 8000 times per second, dedicating sixteen bits to each sample, using signed numeric values, and not using BigEndian format. *(As mentioned earlier, you may need to use a different format on your system.)*

Default data encoding is linear PCM

There are several ways that binary audio data can be encoded into the available bits. The simplest way is known as linear PCM. By default, the constructor that I used constructs an **AudioFormat** object with a linear PCM encoding and the parameters listed above *(I will have more to say about linear PCM encoding and other encoding schemes in future lessons).*

A DataLine.Info object

Now that we have the audio format established, let's return to our discussion of the **captureAudio** method shown in Listing 15.

```
//Continuing with the captureAudio
method

      DataLine.Info dataLineInfo =
                          new
DataLine.Info(

TargetDataLine.class,

audioFormat);
```

Listing 15

The next step in the process is to create a new **DataLine.Info** object that describes the data line that we need to handle the acquisition of the audio data from the microphone.

As you can see in Listing 15, I used a constructor for this object that requires two parameters.

TargetDataLine Class object

The first parameter required by the constructor is a **Class** object. In general, a **Class** object represents the type of an object. In this case, the parameter represents the type of an object instantiated from a class that implements the **TargetDataLine** interface.

According to Sun,

> *"A target data line is a type of* ***DataLine*** *from which audio data can be read. The most common example is a data line that gets its data from an audio capture device. (The device is implemented as a mixer that writes to the target data line.)"*

As you can see, **TargetDataLine** matches our needs exactly.

The audio format parameter

The second parameter required by the constructor in Listing 15 is a specification of the required audio format. That specification is achieved by passing the **AudioFormat** object's reference that was obtained by invoking the **getAudioFormat** method in Listing 13.

Getting a TargetDataLine object

The next step in the process is to get a **TargetDataLine** object to handle data acquisition from the microphone that matches the information encapsulated in the **DataLine.Info** object instantiated in listing 15.

This is accomplished in Listing 16 by invoking the static **getLine** method of the **AudioSystem** class, passing the **DataLine.Info** object as a parameter.

```
      targetDataLine =
(TargetDataLine)

AudioSystem.getLine(dataLineInfo);

Listing 16
```

The AudioSystem class

Here is part of what Sun has to say about the **AudioSystem** class.

> *"AudioSystem includes a number of methods for converting audio data between different formats, and for translating between audio files and streams.*
>
> ***It also provides a method for obtaining a Line directly from the AudioSystem without dealing explicitly with mixers."***

The boldface portion of the above quotation is what interests us in this program. According to Sun, the static **getLine** method:

> *"Obtains a line that matches the description in the specified Line.Info object."*

Note that the **getLine** method returns a reference to an object of type **Line**, which must be downcast to type **TargetDataLine** in Listing 16.

Spawn and start a thread to handle the data capture

The code in Listing 17 creates a new **Thread** object from the **CaptureThread** class and starts it running. As you will see later, the behavior of the thread is to capture audio data from the microphone and to store it in an output audio file.

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

```
  }//end captureAudio method
```

Listing 17

The thread starts running when the user clicks the **Capture** button in Figure 1, and will continue running until the user clicks the **Stop** button in Figure 1.

Once the new thread has been created and started, the code in Listing 17, *(which is part of the **captureAudio** method)*, returns control to the **actionPerformed** event handler method that is registered on the **Capture** button.

The **captureAudio** method ends in Listing 17.

The run method of the CaptureThread class

The **CaptureThread** class is an inner class. An object of this class is used to perform the actual capture of the audio data from the microphone and the storage of that data in the output audio file.

Every thread object has a **run** method, which determines the behavior of the thread. Listing 18 shows the beginning of the **run** method of the **CaptureThread** class, including the declaration of two instance variables that will be used later.

```
class CaptureThread extends Thread{
  public void run(){
    AudioFileFormat.Type fileType =
null;
    File audioFile = null;
```

Listing 18

The output file type

The first step in the execution of the **run** method is to determine the type of output file specified by the user.

> *(Recall that the user specifies a particular audio file type by the selection of a particular radio button in Figure 1.)*

The code in Listing 19 is rather long, but very repetitive and simple.

```
    if(aifcBtn.isSelected()){
      fileType =
AudioFileFormat.Type.AIFC;
      audioFile = new
File("junk.aifc");
    }else if(aiffBtn.isSelected()){
      fileType =
AudioFileFormat.Type.AIFF;
      audioFile = new
File("junk.aif");
    }else if(auBtn.isSelected()){
      fileType =
AudioFileFormat.Type.AU;
      audioFile = new File("junk.au");
    }else if(sndBtn.isSelected()){
      fileType =
AudioFileFormat.Type.SND;
      audioFile = new
File("junk.snd");
    }else if(waveBtn.isSelected()){
      fileType =
AudioFileFormat.Type.WAVE;
      audioFile = new
File("junk.wav");
    }//end if
```

Listing 19

Set file type based on selected radio button

This code in Listing 19 examines the radio buttons to determine which radio button was selected by the user prior to clicking the **Capture** button. Depending on which radio button was selected, the code in Listing 19:

- Sets the type of the audio output file.
- Establishes the file name and extension for the audio output file. *(With a little extra programming effort, you could cause the file name to be provided by the user via a **JTextField** object in the GUI.)*

The AudioFileFormat.Type class

The code in Listing 19 uses constants from the **AudioFileFormat.Type** class. Here is what Sun has to say about that class.

> *"An instance of the **Type** class represents one of the standard types of audio file. Static instances are provided for the common types."*

The class provides the following common types as public static final variables *(constants).*

- AIFC
- AIFF
- AU
- SND
- WAVE

(Because I plan to publish a future tutorial lesson that deals with audio data formats and audio file types in detail, I won't comment further on the five common types in the above list in this lesson.)

The GUI in Figure 1 has one radio button for each of the file types in the above list. Therefore, the code in Listing 19 establishes a file type, name, and extension for each of the five common types.

Not all file types are supported on all systems

I need to point out that not all of the above-listed file types can be created on all systems. For example, types AIFC and SND produce a *"type not supported"* error on my system.

Start acquiring audio data from the microphone

The two statements in Listing 20 cause the audio data acquisition process to begin.

```
    try{

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

Listing 20

The open method

Here is part of what Sun has to say about the **open** method of the **TargetDataLine** interface.

> *"Opens the line with the specified format, causing the line to acquire any required system resources and become operational. The implementation chooses a buffer size ..."*

The start method

Note that this refers to the **start** method of the **TargetDataLine** object, *(not the **start** method of the **CaptureThread** object, which was invoked in Listing 17).*

Here is part of what Sun has to say about the **start** method invoked in Listing 20.

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

The AudioSystem.write method

Everything discussed so far has been leading up to a discussion of the **write** method of the **AudioSystem** class, which is invoked in the **run** method of the thread object in Listing 21.

```
      AudioSystem.write(
            new
AudioInputStream(targetDataLine),
            fileType,
            audioFile);

Listing 21
```

This is a very significant, very high-level method. However, for such a significant method, the information provided by Sun is amazingly sparse. There are two overloaded versions of the **write** method in Java SDK 1.4.1. Here is what Sun has to say about the version shown in Listing 21.

> *"Writes a stream of bytes representing an audio file of the specified file type to the external file provided."*

No loops or buffers

To fully appreciate the significance of this method, you should first note that there are no loops and there is no buffer manipulation involved in the code in Listing 21. Rather, the code in Listing 21 consists of a single statement that invokes the static **write** method of the **AudioSystem** class.

By way of comparison, Listing 22 shows the code from a similar program in Java Sound, Getting Started, Part 2, Capture Using Specified Mixer. This program was used to capture microphone data and to store that data in an object of type **ByteArrayOutputStream** *(a memory buffer)*.

```
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
```

Listing 22

More complex code from earlier program

As you can see, the code in Listing 22 from the earlier program was required to loop while monitoring for a signal to stop data capture. In addition, the code was required to invoke interlaced **read** and **write** methods, while dealing with the internal buffer of the **TargetDataLine** object and a temporary buffer object as well.

All of these details are handled automatically by the single invocation of the **write** method in Listing 21.

Features and characteristics of the write method

In addition to its other features, the **AudioSystem.write** method knows how to detect that the **stop** method has been invoked on the **TargetDataLine** object *(see Listing 7)* and to close the output file when that happens. Therefore, it was not necessary for me to monitor for a signal to stop data capture and close the output file.

The required parameters for the **write** method in Listing 21 are:

- An audio input stream of type **AudioInputStream** containing audio data to be written to the file.
- The type of audio file to write, specified as type **AudioFileFormat.Type**
- The external file to, which the data should be written, specified as type **File**.

The second two parameters were already available, having been created earlier. However, a little extra work was required to create the first parameter.

The AudioInputStream parameter

According to Sun,

> *"An audio input stream is an input stream with a specified audio format and length."*

As of Java SDK 1.4.1, the **AudioInputStream** class provides the two constructors shown in Figure 2.

```
AudioInputStream(InputStream stream
                 AudioFormat format,
                 long length)
This constructor constructs an audio
input stream that has the requested
format and length in sample frames,
using audio data from the specified
input stream.

AudioInputStream(TargetDataLine line)
Constructs an audio input stream that
reads its data from the target data
line indicated.

Figure 2 Constructors for the
AudioInputStream class
```

Since I already had a **TargetDataLine** object, I elected to use the second constructor to create the **AudioInputStream** object required for the first parameter of the **write** method in Listing 21.

End of the run method and end of the class definition

Except for a **catch** block, that ends the definition of the **run** method of the **CaptureThread** class that begins in Listing 18.

That is also the end of the **CaptureThread** class, and the end of the program.

You can view a complete listing of the program in Listing 22 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 22 near the end of the lesson.

Not all file types and data formats are supported

As I mentioned earlier, not all file types can be created on all systems. For example, types AIFC and SND produce a *"type not supported"* error on my system.

Also, not all data formats are supported on all systems. If the format parameters that I used in Listing 14 don't work well for you, try some of the other allowable parameters, which are listed in comments following the variable declarations.

The program GUI at startup

When you start the program, the GUI shown in Figure 1 should appear on your screen. Select an audio file type by selecting one of the radio buttons. Then click the **Capture** button and start talking into the microphone. When you have recorded enough audio data, click the **Stop** button.

An output audio file should be created

At that point in time, an audio file named **junk.xx** should be in the folder containing the compiled version of your program. The file extension will depend on the type of audio file you selected with the radio buttons before you clicked the **Capture** button *(see Listing 19 for an identification of the possible extensions).*

Play back the audio file

You should be able to play the audio file back using a standard media player such as Windows Media Player.

If you run the program two times in succession for the same audio file type, be sure to release the old file from the media player before attempting to create a new file with the same name and extension. Otherwise, it won't be possible to create the new file with the same name and extension, and you will get an error message.

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

Summary

In this lesson, I showed you how to use the Java Sound API to capture audio data from a microphone and how to save that data in an audio file type of your choosing.

Complete Program Listing

A complete listing of the program is shown in Listing 22.

```
/*File AudioRecorder02.java
Copyright 2003, Richard G. Baldwin

This program demonstrates the capture of audio
data from a microphone into an audio file.

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

In addition, five radio buttons appear on the
screen allowing the user to select one of the
following five audio output file formats:

  AIFC
  AIFF
  AU
  SND
  WAVE

When the user clicks the Capture button, input
data from a microphone is captured and saved in
```

```
an audio file named junk.xx having the specified
file format.  (xx is the file extension for the
specified file format.  You can easily change the
file name to something other than junk if you
choose to do so.)

Data capture stops and the output file is closed
when the user clicks the Stop button.

It should be possible to play the audio file
using any of a variety of readily available
media players, such as the Windows Media Player.

Not all file types can be created on all systems.
For example, types AIFC and SND produce a "type
not supported" error on my system.

Be sure to release the old file from the media
player before attempting to create a new file
with the same extension.

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 AudioRecorder02 extends JFrame{

  AudioFormat audioFormat;
  TargetDataLine targetDataLine;

  final JButton captureBtn =
                          new JButton("Capture");
  final JButton stopBtn = new JButton("Stop");

  final JPanel btnPanel = new JPanel();
  final ButtonGroup btnGroup = new ButtonGroup();
  final JRadioButton aifcBtn =
                        new JRadioButton("AIFC");
  final JRadioButton aiffBtn =
                        new JRadioButton("AIFF");
  final JRadioButton auBtn =//selected at startup
                     new JRadioButton("AU",true);
  final JRadioButton sndBtn =
                         new JRadioButton("SND");
  final JRadioButton waveBtn =
                        new JRadioButton("WAVE");

  public static void main( String args[]){
    new AudioRecorder02();
  }//end main
  public AudioRecorder02(){//constructor
    captureBtn.setEnabled(true);
    stopBtn.setEnabled(false);

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

    stopBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                                  ActionEvent e){
          captureBtn.setEnabled(true);
          stopBtn.setEnabled(false);
          //Terminate the capturing of input data
          // from the microphone.
          targetDataLine.stop();
          targetDataLine.close();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

    //Put the buttons in the JFrame
    getContentPane().add(captureBtn);
    getContentPane().add(stopBtn);

    //Include the radio buttons in a group
    btnGroup.add(aifcBtn);
    btnGroup.add(aiffBtn);
    btnGroup.add(auBtn);
    btnGroup.add(sndBtn);
    btnGroup.add(waveBtn);

    //Add the radio buttons to the JPanel
    btnPanel.add(aifcBtn);
    btnPanel.add(aiffBtn);
    btnPanel.add(auBtn);
    btnPanel.add(sndBtn);
    btnPanel.add(waveBtn);

    //Put the JPanel in the JFrame
    getContentPane().add(btnPanel);

    //Finish the GUI and make visible
    getContentPane().setLayout(new FlowLayout());
    setTitle("Copyright 2003, R.G.Baldwin");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(300,120);
    setVisible(true);
  }//end constructor

  //This method captures audio input from a
  // microphone and saves it in an audio file.
  private void captureAudio(){
    try{
      //Get things set up for capture
      audioFormat = getAudioFormat();
      DataLine.Info dataLineInfo =
                          new DataLine.Info(
                            TargetDataLine.class,
                            audioFormat);
      targetDataLine = (TargetDataLine)
               AudioSystem.getLine(dataLineInfo);

      //Create a thread to capture the microphone
      // data into an audio file and start the
      // thread running.  It will run until the
      // Stop button is clicked.  This method
      // will return after starting the thread.
      new CaptureThread().start();
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end captureAudio method

  //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
// and write it to an output audio file.
class CaptureThread extends Thread{
  public void run(){
    AudioFileFormat.Type fileType = null;
    File audioFile = null;

    //Set the file type and the file extension
    // based on the selected radio button.
    if(aifcBtn.isSelected()){
      fileType = AudioFileFormat.Type.AIFC;
      audioFile = new File("junk.aifc");
    }else if(aiffBtn.isSelected()){
      fileType = AudioFileFormat.Type.AIFF;
      audioFile = new File("junk.aif");
    }else if(auBtn.isSelected()){
      fileType = AudioFileFormat.Type.AU;
      audioFile = new File("junk.au");
    }else if(sndBtn.isSelected()){
      fileType = AudioFileFormat.Type.SND;
      audioFile = new File("junk.snd");
    }else if(waveBtn.isSelected()){
      fileType = AudioFileFormat.Type.WAVE;
      audioFile = new File("junk.wav");
    }//end if

    try{
      targetDataLine.open(audioFormat);
      targetDataLine.start();
      AudioSystem.write(
            new AudioInputStream(targetDataLine),
            fileType,
            audioFile);
    }catch (Exception e){
      e.printStackTrace();
    }//end catch

  }//end run
}//end inner class CaptureThread
//=============================================//

}//end outer class AudioRecorder02.java
```

Listing 22

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-