Java Sound, Getting Started, Part 2, Capture using Specified Mixer

Baldwin shows you how to use the Java Sound API to capture audio data from a microphone and how to save that data in a ByteArrayOutputStream object. He also shows you how to identify the mixers available on your system, and how to specify a particular mixer for use in the acquisition of audio data from the microphone.

Published: February 18, 2003
By Richard G. Baldwin

Java Programming Notes # 2012

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 1, Playback. This lesson, entitled *Java Sound, Getting Started, Part 2, Capture using Specified Mixer,* is a follow-on to the previous lesson.

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 Java Sound API is based on the concept of *lines* and *mixers.* In this lesson, I will provide a program that you can use to first capture and then to play back sound.

The previous lesson provided a detailed discussion of the playback section of the program. However, that lesson didn't expose the use of a mixer. Rather, that lesson made use of class methods of the **AudioSystem** class, which abstract the use of mixers to the background.

In this lesson, I will provide a detailed explanation of the code used to capture audio data from a microphone. Even though it isn't necessary, this lesson will also expose the specification of a particular mixer to capture the audio data.

Discussion and Sample Code

Mixers

Here is part of what Sun has to say about a mixer:

> *"A mixer is an audio device with one or more lines. It need not be designed for mixing audio signals. A mixer that actually mixes audio has multiple input (source) lines and at least one output (target) line. The former are often instances of classes that implement* ***SourceDataLine****, and the latter,* ***TargetDataLine****.* ***Port*** *objects, too, are either source lines or target lines. A mixer can accept prerecorded, loopable sound as input, by having some of its source lines be instances of objects that implement the* ***Clip*** *interface."*

Lines

Sun has this to say about the **Line** interface:

> *"A line is an element of the digital audio "pipeline," such as an audio input or output port, a mixer, or an audio data path into or out of a mixer. The audio data flowing through a line can be mono or multichannel (for example, stereo). ... A line can have* ***controls****, such as gain, pan, and reverb."*

Some important terms

The above quotations from Sun mention the following terms:

- TargetDataLine
- Mixer
- Port
- Controls

Figure 1 shows how these terms can come together to form a simple audio input system.

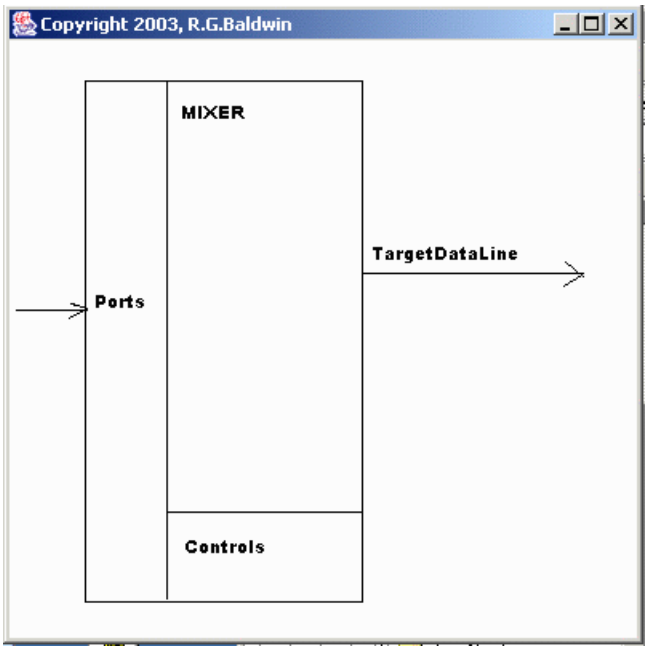

Figure 1 An audio input system

In Figure 1, a **Mixer** object is configured with one or more ports, some controls, and a **TargetDataLine** object.

What is a TargetDataLine?

The terminology used here 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.

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 a **ByteArrayInputStream** object in memory.)*

Based on a program from a previous lesson

In a previous lesson, I provided and discussed a program that captures audio data from a microphone port, stores that data in memory, and plays the captured data back through a speaker port. That lesson discussed the playback portion of the program in detail, but did not discuss the data-capture portion of the program.

A slightly modified version of that program will be discussed in this lesson. A copy of the modified program is shown in Listing 12 near the end of this lesson. This lesson will discuss the data-capture portion of the program in detail.

An explicit Mixer object

The previous version of the program didn't make explicit use of a **Mixer** object. Therefore, it wasn't possible to see how the concept of a **Mixer** entered into the program. *(A Mixer was implicitly used, but was not identified as such.)*

Even though it wasn't necessary for the successful operation of the program, I modified this version of the program to show the explicit use of a **Mixer** object.

The graphical user interface (GUI)

A large portion of this program is dedicated to creating a graphical user interface, which is used to control the operation of the program. Since that code is straightforward, I won't discuss those parts of the program.

The data-capture side of the program

As mentioned earlier, I provided a detailed discussion of the playback side of the program in the previous lesson. I will provide a detailed discussion of the data-capture side of the program in this lesson.

As you will see later, the data-capture portion of the program captures audio data from the microphone and stores it in a **ByteArrayOutputStream** object. Then the playback method named **playAudio**, *(which was discussed in a previous lesson)*, plays back the audio data that is stored in the **ByteArrayOutputStream** object. I made very few changes to the playback side of the program. Therefore, the discussion in the previous lesson should suffice for your understanding of the playback side of the program.

The user interface

When this program is executed, the GUI shown in Figure 2 appears on the screen. As you can see, this GUI contains three buttons:

- Capture
- Stop
- Playback

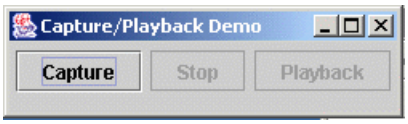

Figure 2 Program GUI

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 of the captured data begins when the user clicks the **Playback** button.

Available Mixers

Not all computers provide the same set of mixers. This version of the program displays a list of mixers available on the machine at runtime. The following list of mixers was produced when the program was run on my machine:

```
Java Sound Audio Engine
Microsoft Sound Mapper
Modem #0 Line Record
ESS Maestro
```

Thus, my machine had the four mixers listed above available at the time the program was run *(your computer may display a different list of mixers)*

Using a specific mixer

After displaying the list of available mixers, the program gets and uses one of the available mixers from the list. This is different from the version of the program discussed in a previous lesson. That version of the program simply asked for a compatible mixer rather than identifying a specific mixer.

Some mixers work and some don't

I determined experimentally that either of the following mixers could be successfully used in this program on my machine:

```
Microsoft Sound Mapper
ESS Maestro
```

I also determined experimentally that neither of the following mixers would work in this program on my machine:

```
Java Sound Audio Engine
Modem #0 Line Record
```

These two mixers fail at runtime for different reasons.

The *Java Sound Audio Engine* mixer failed due to a data format compatibility problem *(it may have been possible to correct this failure by specifying a different data format, but I didn't try).*

The *Modem #0 Line Record* mixer failed due to an *"Unexpected Error."*

The program was tested using Java SDK 1.4.1_01 under Win2000.

Will discuss in fragments

Those of you who follow my work will not be surprised to learn that I will discuss this program in fragments. A complete listing of the program is shown in Listing 12 near the end of the lesson.

The class named AudioCapture02

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

```
public class AudioCapture02 extends
JFrame{

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

Listing 1
```

The instance variables

The code in Listing 1 declares several instance variables. One of those variables, named **stopCapture**, controls the starting and stopping of the data capture process.

The value of this variable is initialized to **false**. The value is later changed to **true** when the user clicks the **Stop** button in the GUI. The code in the data-capture thread terminates when the value of this variable changes to **true**.

The use of the other instance variables declared in Listing 1 will become obvious as they appear in the code.

The method named captureAudio

Now I am going to discuss the method named **captureAudio**. This is the method that is invoked when the user clicks the **Capture** button in the GUI. This method captures audio input data from a microphone and saves that data in a **ByteArrayOutputStream** object for later playback.

The beginning of the method named **captureAudio** is shown in Listing 2.

Display available mixers

The code fragment in Listing 2 uses the **getMixerInfo** method of the **AudioSystem** class to get and display a list of the available mixers in the system at the time the program is run.

```
  private void captureAudio(){
    try{
      Mixer.Info[] mixerInfo =

AudioSystem.getMixerInfo();
      System.out.println("Available
mixers:");
      for(int cnt = 0; cnt <
mixerInfo.length;

cnt++){

        System.out.println(mixerInfo[cn
t].

getName());
      }//end for loop
```

Listing 2

An array of Mixer.Info objects

The **getMixerInfo** method populates and returns a reference to an array object, which contains references to objects of type **Mixer.Info**. Each such object contains information about one of the available mixers. The **length** property of the array object indicates the number of available mixers. According to Sun:

*"The **Mixer.Info** class represents information about an audio mixer, including the product's name, version, and vendor, along with a textual description. This information may be retrieved through the **getMixerInfo** method of the **Mixer** interface."*

I told you some of what Sun has to say about the Mixer interface earlier.

Display the list of available mixers

The code in Listing 2 also iterates on the array object to display the name of each mixer currently available in the system. As indicated earlier, this code produced the following screen output when the program was run on my machine, indicating that four mixers were currently available:

```
Available mixers:
Java Sound Audio Engine
Microsoft Sound Mapper
Modem #0 Line Record
ESS Maestro
```

A reference to a **Mixer.Info** object describing each of these mixers is contained in the array object referred to by the reference variable named **mixerInfo** in Listing 2. I will make use of the contents of one of the array elements later to select a specific mixer for use by the program.

The audio data format

Quite a lot of setup is required to facilitate the capture of audio data. Listing 3 begins the process of getting everything set up to capture audio data from the microphone.

One of the things that are required is a specification of the format of the audio data. The code in Listing 3 invokes the **getAudioFormat** method to get an object of type **AudioFormat** and save its reference in the instance variable named **audioFormat**.

```
      audioFormat = getAudioFormat();
```

Listing 3

The method named getAudioFormat

At this point, I will briefly discuss the method named **audioFormat**. *(This discussion will be brief because I discussed this method in a previous lesson.)*

The entire **getAudioFormat** method is shown in Listing 4.

```
  private AudioFormat
getAudioFormat(){
    float sampleRate = 8000.0F;
```

```
    int sampleSizeInBits = 16;
    int channels = 1;
    boolean signed = true;
    boolean bigEndian = false;

    return new AudioFormat(sampleRate,

sampleSizeInBits,
                           channels,
                           signed,
                           bigEndian);
  }//end getAudioFormat

Listing 4
```

Aside from some initialized variable declarations, the code in Listing 3 consists of a single executable statement.

An AudioFormat object

The **getAudioFormat** method creates and returns an object of the **AudioFormat** class.

> *(I don't believe that there is any guarantee that a given set of audio format parameters will work on all systems, since the system sound card is a part of the process. There are many brands and types of system sound cards. If these format parameters don't work for you, try some of the other allowable parameter values, which are presented below.)*

What does Sun have to say?

Here is part of what Sun has to say about the **AudioFormat** class:

> *"**AudioFormat** is the class that specifies a particular arrangement of data in a sound stream. By examining the information stored in the audio format, you can discover how to interpret the bits in the binary sound data."*

Two constructors are available

The **AudioFormat** class has two constructors. *(I elected to use the simpler of the two.)* For this constructor, the required parameters are:

- Sample rate in samples per second. *(Allowable values include 8000, 11025, 16000, 22050, and 44100 samples per second.)*
- Sample size in bits. *(Allowable values include 8 and 16 bits per sample.)*
- Number of channels. *(Allowable values include 1 channel for mono and 2 channels for stereo.)*
- Signed or unsigned data. *(Allowable values include true and false for signed data or unsigned data.)*

- Big-endian or little-endian order. *(This has to do with the order in which the data bytes are stored in memory. You can learn about this topic here.)*

As you can see in Listing 4, this method specifies the following parameters for the new **AudioFormat** object:

- 8000 samples per second
- 16 bits per sample
- 1 channel *(mono)*
- Signed data
- Little-endian order

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. 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).*

Now back to the captureAudio method

Having established an audio data format, the next step is to get an object of type **DataLine.Info**, as shown in Listing 5.

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

TargetDataLine.class,

audioFormat);
```

Listing 5

The **DataLine.Info** class extends the **Line.Info** class. Let's begin our investigation by taking a look at some of what Sun has to say about the **Line.Info** class.

> *"A **Line.Info** object contains information about a line. The only information provided by **Line.Info** itself is the Java class of the line. A subclass of **Line.Info** adds other kinds of information about the line. This additional information depends on which Line subinterface is implemented by the kind of line that the **Line.Info** subclass describes."*

The code in Listing 5 instantiates a new object of the **DataLine.Info** class, which is one of the subclasses of **Line.Info**.

DataLine.Info class

Here is part of what Sun has to say about the **DataLine.Info** class:

> *"Besides the class information inherited from its superclass, **DataLine.Info** provides additional information specific to data lines. This information includes:*
>
> - *the audio formats supported by the data line*
> - *the minimum and maximum sizes of its internal buffer*
>
> *Because a **Line.Info** knows the class of the line it describes, a **DataLine.Info** object can describe **DataLine** subinterfaces such as **SourceDataLine**, **TargetDataLine**, and **Clip**. You can query a mixer for lines of any of these types, passing an appropriate instance of **DataLine.Info** as the argument to a method such as **Mixer.getLine(Line.Info)**."*

DataLine.Info constructor

Three overloaded constructors are available for a **DataLine.Info** object. Two of them allow you to specify buffer size information. I elected to use the simplest of the three, which doesn't require the specification of buffer information, but uses default buffer sizes instead.

According to Sun, the constructor that I elected to use:

> *"Constructs a data line's info object from the specified information, which includes a single audio format."*

Note the two parameters passed to the constructor for the new **DataLine.Info** object in Listing 5. As you can see, the **DataLine.Info** object instantiated in Listing 5 describes a line of type **TargetDataLine**, with the format that was specified earlier.

What is a TargetDataLine?

TargetDataLine is a sub-interface of **DataLine**, which in turn, is a sub-interface of **Line**. Therefore, before getting into the details of **TargetDataLine**, we need to take a look at the **DataLine** interface.

The DataLine interface

Here is part of what Sun has to say about the **DataLine** interface:

> *"**DataLine** adds media-related functionality to its superinterface, **Line**. This functionality includes transport-control methods that start, stop, drain, and flush the audio data that passes through the line."*

For example, the **drain** method is used in the playback side of this program to ensure that the internal buffer of a line is empty before closing the line.

> *"Data lines are used for output of audio by means of the subinterfaces **SourceDataLine** or **Clip**, which allow an application program to write data. Similarly, audio input is handled by the subinterface **TargetDataLine**, which allows data to be read."*

This quotation from Sun is of particular interest to us because we will be using **TargetDataLine** to capture audio input data from a microphone.

There are several other interesting aspects of the **DataLine** interface, which we will use in future lessons. Therefore, I won't discuss them in this lesson.

The TargetDataLine interface

Figure 5 instantiates a **DataLine.Info** object that describes a line of type **TargetDataLine**. Here is part of what Sun has to say about the **TargetDataLine** interface:

> *"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.)"*

We are discussing the code in this program that captures audio data from a microphone. In concert with the above quotation, the combination of the microphone and a mixer can be viewed as an *audio capture device*, which captures audio data from a microphone and writes that data to a **TargetDataLine** object. Later on, you will see code that reads the audio data from the **TargetDataLine** object and transfers it to a **ByteArrayOutputStream** object.

Confusing terminology

It is very important to keep the naming convention straight, because it may be just the reverse of what you would expect. Here is what Sun has to say regarding the naming convention for the **TargetDataLine** interface:

> *"Note that the naming convention for this interface reflects the relationship between the line and its mixer. From the perspective of an application, a target data line may act as a source for audio data."*

Similarly, here is what Sun has to say about the **SourceDataLine** interface with respect to the naming convention:

> *"From the perspective of an application, a source data line may act as a target for audio data."*

The target is a source and the source is a target.

Are you confused yet?

From the viewpoint of the application, *(as opposed to the viewpoint of the mixer)* a **TargetDataLine** is the source of audio data *(such as data captured from a microphone).*

From the viewpoint of the application *(as opposed to the viewpoint of the mixer),* a **SourceDataLine** is a target for audio data *(such as a speaker).*

Getting a TargetDataLine object

Sun goes on to tell us:

> *"The target data line can be obtained from a mixer by invoking the **getLine** method of **Mixer** with an appropriate **DataLine.Info** object."*

That is exactly what we are going to do later.

An internal buffer ...

A **TargetDataLine** object has an internal buffer that is used to temporarily store the input audio data until it is read by the application. Sun has a few cautions for us regarding the use of that buffer:

> *"The TargetDataLine interface provides a method for reading the captured data from the target data line's buffer. Applications that record audio should read data from the target data line quickly enough to keep the buffer from overflowing, which could cause discontinuities in the captured data that are perceived as clicks. ... If the buffer does overflow, the oldest queued data is discarded and replaced by new data."*

Hopefully your computer will be fast enough to capture the input audio data from the microphone without buffer overflow. My computer is not a particularly fast one, and it seems to capture the data at 8000 samples per second with no problems.

Selecting an available mixer

As I promised earlier, this program is going to select one of the available mixers, which was not the case in the version of the program discussed in a previous lesson. The earlier version of the program simply requested access to a compatible mixer without specifying any particular mixer.

Also, as I mentioned earlier, I determined experimentally that only two of the four available mixers on my machine would work in this program.

An array of Mixer.Info data

Earlier in the program, we created and populated an array object whose elements refer to **Mixer.Info** objects that describe the four available mixers on my machine *(your machine may contain different mixers).* The code in Listing 6 gets a reference to a **Mixer** object described by the **Mixer.Info** object at index 3 of the array.

```
      Mixer mixer = AudioSystem.

getMixer(mixerInfo[3]);

Listing 6
```

A brute force experiment

By simply recompiling and running the program with different index values in Listing 6, I identified the two mixers that will work on my machine as those whose description was stored at index 1 and index 3 in the array. However, that may not be the case on your machine. You may need to perform a similar experiment to identify the compatible mixers.

> *(There are more elegant ways to identify compatible mixers, but I decided to do it by brute force now and to discuss the more elegant ways in a future lesson.)*

After the code in Listing 6 executes, the variable named **mixer** contains a reference to an object of type **Mixer** described as **ESS Maestro** on my machine.

> *(For whatever its worth, the sound subsystem on my machine is described in the hardware properties as ESS Maestro2E MPU-401 Compatible.)*

Get a TargetDataLine object

Now that we have our mixer, the next thing we need to do is to get a line. The code in Listing 7 invokes the **getLine** method on the **Mixer** object to get a **TargetDataLine** object.

```
      targetDataLine =
(TargetDataLine)

mixer.getLine(dataLineInfo);


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

Listing 7
```

According to Sun, the **getLine** method of the **Mixer** interface requires an incoming parameter of type **Line.Info.** The method:

> *"Obtains a line that is available for use and that matches the description in the specified **Line.Info** object."*

As you will recall from Listing 5, our **Line.Info** object describes a **TargetDataLine**. The code in Listing 7 passes the **Line.Info** object that we created earlier in Listing 5 as a parameter to the **getLine** method.

The **getLine** method returns a reference to an object as type **Line**. We must downcast it to type **TargetDataLine** in order to use it.

Prepare the line for use

Once we have the **TargetDataLine** object, there are a couple more steps required to prepare it for use. The code in Listing 7 invokes the **open** method on the line object passing our format object as a parameter. According to Sun, this version of the **open** method

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

Two overloaded versions of the **open** method are available. The version that I elected to use chooses a buffer size automatically. The other version requires the programmer to specify the buffer size.

Invoke the start method

The code in Listing 7 also invokes the **start** method on the **TargetDataLine** object. According to Sun, the **start** method

> *"Allows a line to engage in data I/O."*

At this point, the audio system begins capturing data from the microphone, storing it in an internal buffer, and making it available to the program.

Don't allow the internal buffer to overflow

The program must start reading data from that internal buffer very quickly, or the internal buffer may overflow. As discussed earlier, the program must continue to read data from the internal buffer at a sufficiently fast rate to keep the internal buffer from overflowing.

Capture some audio data

At this point, we have finally prepared everything necessary to make it possible to acquire audio data from the microphone. The next step is to create a **Thread** object *(to capture and save the data)*, and to start the thread running.

The code in Listing 8 creates an object of the **CaptureThread** class, and starts it running.

```
      Thread captureThread = new
CaptureThread();
```

```
      captureThread.start();

Listing 8
```

Continue running until Stop

This thread will continue running and saving audio data until the user presses the **Stop** button.

> *(Note however, that the captured data is being saved in memory. If you allow it to capture too much data, you may run out of memory.)*

If you examine the code in Listing 12 near the end of the lesson, you will see that except for a **catch** block, this is the end of the method named **captureAudio**. The code in the **catch** block is very simple, so I won't discuss it here.

The CaptureThread class

Listing 9 shows the beginning of a class named **CaptureThread**. This class extends the **Thread** class, and is used to read data from the line's internal buffer. The audio data read from that buffer is saved in an object of type **ByteArrayOutputStream**.

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

Listing 9
```

The **CaptureThread** class declares one instance variable, which refers to an array object of type **byte**. This object is used as an intermediate buffer in the process of moving audio data from the line's internal buffer to the **ByteArrayOutputStream** object. The size of this array was set rather arbitrarily to 10,000 bytes.

Concurrent operation

Because an object of this class is a **Thread**, it runs concurrently with the other threads in the program. Thus, it runs concurrently with the thread that handles events resulting from clicking the buttons on the GUI.

The run method

Every **Thread** class must define a method named **run**, which determines the behavior of the thread. The beginning of the **run** method for the **CaptureThread** class is shown in Listing 10.

```
  public void run(){
    byteArrayOutputStream =
                      new
ByteArrayOutputStream();
```

```
    stopCapture = false;

Listing 10
```

The code in the **run** method *(Listing 10)* begins by instantiating a new **ByteArrayOutputStream** object and storing that object's reference in the instance variable named **byteArrayOutputStream**.

If you use the GUI buttons to repeatedly cycle this program through the **Capture/Stop/Playback** cycle, a new **ByteArrayOutputStream** object will be created and used each time you press the **Capture** button.

The control variable named stopCapture

The most interesting thing in Listing 10 is the initialization of the **boolean** instance variable *(named **stopCapture**)* to a value of **false**. This variable is used to control the duration of audio data capture. Its value is switched from **false** to **true** by the event handler when the user presses the **Stop** button.

As you will see shortly, when the value of **stopCapture** switches to **true**, the audio data capture process is terminated.

> *(To simplify the discussion, I am going to omit the exception handling code in the **run** method. That code is straightforward, and you can view it in Listing 12 near the end of the lesson.)*

Remaining code in the run method

Other than the exception handling code, the remaining code in the **run** method is shown in Listing 11.

```
      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();

  }//end run
}//end inner class CaptureThread
```

Listing 11

The code in the **run** method loops and captures audio data until the event handler on the **Stop** button switches the value of **stopCapture** from **false** to **true**.

Here is what happens during each iteration of the **while** loop.

Get audio data from the TargetDataLine's internal buffer

The **read** method is invoked on the **TargetDataLine** object in an attempt to read enough bytes from that object's internal buffer to fill the array object referred to by **tempBuffer**.

> *(It may not always be possible to read that many bytes from the line's internal buffer. There simply may not be that many bytes of audio data available in the internal buffer.)*

The **read** method transfers the available bytes from the internal buffer into the array provided as an incoming parameter.

> *(If the number of available bytes in the internal buffer exceeds the size of the array, only the number required to fill the array are transferred. The surplus bytes remain in the internal buffer and are available for the next **read** operation.)*

Then the **read** method returns the number of bytes actually transferred as type **int**. That value is saved in the variable named **cnt.**

Transfer the data to the ByteArrayOutputStream object

Then the code in the **run** method in Listing 11 invokes the **write** method on the **ByteArrayOutputStream** object to transfer the bytes from the array referred to by **tempBuffer** to the stream object referred to by **byteArrayOutputStream**.

Go back to the top of the loop

Then control transfers back to the top of the **while** loop where the value of **stopCapture** is tested once again.

If the value of **stopCapture** is still **false**, that indicates that the **Stop** button has not been clicked by the user. The process is repeated through another iteration.

However, if the user has clicked the **Stop** button, thus causing the value of **stopCapture** to switch from **false** to **true**, the loop terminates. In this case, the **ByteArrayOutputStream** object is closed, and the **run** method terminates. This causes the thread to die a natural death and terminates the capture of audio data.

A clarification

Note, however, that I didn't invoke the **stop** method on the **TargetDataLine** object. As a result, the line will continue acquiring audio data and making that data available in its internal buffer until the program terminates.

Since the **run** method of the thread has ceased reading bytes from the line's internal buffer, the buffer will simply overflow. It should be possible to restart the line and read audio data that was acquired during the interval. However, this program was not designed to support that kind of operation.

If you are concerned about the **TargetDataLine** continuing to consume resources, you could insert the following statement in Listing 11 immediately following the end of the **while** loop:

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

According to Sun, invocation of the **stop** method on a line:

> *"Stops the line. A stopped line should cease I/O activity."*

What is a ByteArrayOutputStream object?

According to Sun, the **ByteArrayOutputStream** class

> *"implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it."*

It should be possible to run out of memory if an attempt is made to write too much data into the byte array. However, the Sun documentation for SDK 1.4.1_01 doesn't tell us what happens in that event. Hopefully, an **OutOfMemory** error would be thrown, which would cause this program to terminate abnormally.

Let's recap

Let's recap the steps involved in capturing audio data from a microphone and saving that data in a **ByteArrayOutputStream** object, as implemented by this program.

- Identify the available mixers, and save a **Mixer.Info** object describing a compatible mixer.
- Instantiate an **AudioFormat** object that specifies a particular arrangement of audio data bytes in a sound stream. Many options are available here.

- Instantiate a **DataLine.Info** object that describes an object of type **TargetDataLine** set up for the **AudioFormat** described above.
- Invoke the **getMixer** method of the **AudioSystem** class to get a reference to a **Mixer** object that matches the **Mixer.Info** object saved earlier.
- Invoke the **getLine** method on the **Mixer** object to get a **TargetDataLine** object that matches the characteristics of the **DataLine.Info** object instantiated earlier.
- Invoke the **open** method on the **TargetDataLine** object, passing the **AudioFormat** object as a parameter.
- Invoke the **start** method on the **TargetDataLine** object to cause the line to start acquiring data from the microphone.
- Spawn and start a **Thread** object to transfer audio data in real time from the internal buffer of the **TargetDataLine** object to a **ByteArrayOutputStream** object.
- When an appropriate amount of audio data has been captured, cause the **Thread** object to stop transferring data from the **TargetDataLine** object to the **ByteArrayOutputStream** object.
- Optionally invoke the **stop** method on the **TargetDataLine** object to cause it to stop acquiring audio data from the microphone.

Note that it isn't always necessary to explicitly specify a mixer as was done in this program. A similar program in a previous lesson simply invoked the **getLine** method of the **AudioSystem** class to get a **TargetDataLine** object for a particular data format on a *compatible* mixer. I elected to explicitly specify a mixer in this program for illustration purposes only.

Run the Program

At this point, you may find it useful to compile and run the program shown in Listing 12 near the end of the lesson.

Capture and playback audio data

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

- Start the program running. The simple GUI shown in Figure 2 will appear on the screen.
- Click the **Capture** button and speak into the microphone.
- Click the **Stop** button to terminate capturing data.
- Click the **Playback** button to play your captured voice back through the system speakers.

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

This program saves the data that it captures in memory, so be careful to avoid running out of memory.

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 a **ByteArrayOutputStream** object. I also showed you how to identify the mixers available on your system, and how to specify a particular mixer for use in the acquisition of audio data from the microphone.

Complete Program Listing

A complete listing of the program is shown in Listing 12.

```
/*File AudioCapture02.java
This program demonstrates the capture and
subsequent playback of audio data.

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

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

Data capture stops when the user clicks the Stop
button.

Playback begins when the user clicks the Playback
button.

This version of the program gets and  displays a
list of available mixers, producing the following
output:

Available mixers:
Java Sound Audio Engine
Microsoft Sound Mapper
Modem #0 Line Record
ESS Maestro

Thus, this machine had the four mixers listed
above available at the time the program was run.

Then the program gets and uses one of the
available mixers instead of simply asking for a
compatible mixer as was the case in a previous
version of the program.

Either of the following two mixers can be used in
this program:

Microsoft Sound Mapper
ESS Maestro
```

```
Neither of the following two mixers will work in
this program.  The mixers fail at runtime for
different reasons:

Java Sound Audio Engine
Modem #0 Line Record

The Java Sound Audio Engine mixer fails due to a
data format compatibility problem.

The Modem #0 Line Record mixer fails due to an
"Unexpected Error"

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 AudioCapture02 extends JFrame{

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

  public static void main(String args[]){
    new AudioCapture02();
  }//end main

  public AudioCapture02(){//constructor
    final JButton captureBtn =
                          new JButton("Capture");
    final JButton stopBtn = new JButton("Stop");
    final JButton playBtn =
                         new JButton("Playback");

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

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

```
          // clicked.
          captureAudio();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()
    getContentPane().add(captureBtn);

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

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

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

  //This method captures audio input from a
  // microphone and saves it in a
  // ByteArrayOutputStream object.
  private void captureAudio(){
    try{
      //Get and display a list of
      // available mixers.
      Mixer.Info[] mixerInfo =
                      AudioSystem.getMixerInfo();
      System.out.println("Available mixers:");
      for(int cnt = 0; cnt < mixerInfo.length;
                                          cnt++){
        System.out.println(mixerInfo[cnt].
                                      getName());
      }//end for loop
```

```
      //Get everything set up for capture
      audioFormat = getAudioFormat();

      DataLine.Info dataLineInfo =
                           new DataLine.Info(
                           TargetDataLine.class,
                           audioFormat);

      //Select one of the available
      // mixers.
      Mixer mixer = AudioSystem.
                          getMixer(mixerInfo[3]);

      //Get a TargetDataLine on the selected
      // mixer.
      targetDataLine = (TargetDataLine)
                       mixer.getLine(dataLineInfo);
      //Prepare the line for use.
      targetDataLine.open(audioFormat);
      targetDataLine.start();

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

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

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

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

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

//Inner class to capture data from microphone
class CaptureThread extends Thread{
  //An arbitrary-size temporary holding buffer
  byte tempBuffer[] = new byte[10000];
  public void run(){
    byteArrayOutputStream =
                      new ByteArrayOutputStream();
    stopCapture = false;
    try{//Loop until stopCapture is set by
        // another thread that services the Stop
        // button.
      while(!stopCapture){
        //Read data from the internal buffer of
        // the data line.
```

```
        int cnt = targetDataLine.read(tempBuffer,
                                      0,
                                      tempBuffer.length);
        if(cnt > 0){
          //Save data in output stream object.
          byteArrayOutputStream.write(tempBuffer,
                                      0,
                                      cnt);
        }//end if
      }//end while
      byteArrayOutputStream.close();
    }catch (Exception e) {
      System.out.println(e);
      System.exit(0);
    }//end catch
  }//end run
}//end inner class CaptureThread
//===================================//
//Inner class to play back the data
// that was saved.
class PlayThread extends Thread{
  byte tempBuffer[] = new byte[10000];

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

}//end outer class AudioCapture02.java
```

Listing 12

Copyright 2003, Richard G. Baldwin. Reproduction in whole or in part in any form or medium without express written permission from Richard Baldwin is prohibited.

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-