Java Sound, Writing More Robust Audio Programs

Baldwin shows you how to write more robust audio programs by using the getAudioFileTypes method of the AudioSystem class to limit the file-type choices presented to the user. This eliminates the possibility that the user will select a file type that is not supported by the system.

Published: May 6, 2003
By Richard G. Baldwin

Java Programming Notes # 2020

- 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, Using Audio Line Events.

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.

Material in earlier lessons

Earlier lessons in the series showed you how to:

- Play back audio files, including those that you create using a Java program, and those that you acquire from other sources.
- Capture microphone data into audio files types of your own choosing.
- Capture microphone data into a **ByteArrayOutputStream** object.
- Use the Sound API to play back previously captured audio data.
- Identify the mixers available on your system.
- Specify a particular mixer for use in the acquisition of audio data from a microphone.
- Understand the use of lines and mixers in the Java Sound API.

Preview

Previous programs were simple and not very robust

In the interest of simplicity, the sample programs that I have provided in earlier lessons have not been particularly robust. For example, in the previous programs, if the program attempts to capture data using an audio format that is not supported by the system, the program simply throws an error and aborts. Similarly, if the program attempts to write an audio file as a file type that is not supported by the system, the program throws an error and aborts.

Your programs need to be more robust

Obviously, programs that you write for the real world must be more robust than those that I have provided. Fortunately, the **AudioSystem** class provides methods, such as **getAudioFileTypes**, **isFileTypeSupported**, and **isConversionSupported**, which can be used to write more robust programs. Methods such as this can be used to limit the choices presented to the user, or to test the choices made by the user before trying to execute code that implements those choices.

Limiting the choice of output file types

In this lesson, I will teach you how to use the **getAudioFileTypes** method of the **AudioSystem** class to limit the file-type choices presented to the user, thus eliminating the possibility that the user will select an output file type that is not supported by the system. Hopefully, this example will suggest other ways in which you can use methods of the **AudioSystem** class to make your audio programs more robust.

Discussion and Sample Code

The user interface

The user interface for the sample program that I will discuss in this lesson is shown in Figure 1.

Figure 1 GUI for current version of the program

More robust update of an earlier program

This is an update of the program named **AudioRecorder02**, which was discussed in a previous lesson entitled Java Sound, Capturing Microphone Data into an Audio File.

This updated version is more robust than the original version. In particular, this updated version demonstrates how to limit the file-type choices to only those that are supported by the system. This eliminates the possibility that the user might select an audio file type that is not supported by the system.

Compare GUI with earlier program

The GUI for the earlier program named **AudioRecorder02** is shown in Figure 2. In both Figure 1 and Figure 2, the user selects a file type by selecting a radio button from among those exposed by the GUI.

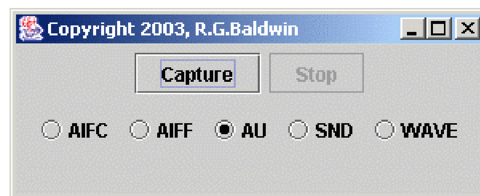

Figure 2 GUI from earlier version of the program

If you compare Figure 2 with Figure 1, you will see that the GUI shown in Figure 1 limits the user to the three file types supported by my system, whereas the GUI in Figure 2 allows the user to select file types that are not supported by my system. Selecting file types that are not supported by the system results in runtime errors which, if not handled properly, will cause the program to abort.

(Because your system may support a different set of file types, your GUI may not look exactly like Figure 1. In particular, your GUI may display a different set of radio buttons.)

Basic operation

This program demonstrates the capture of audio data from a microphone into an audio file type of the user's choosing.

When the program starts, a GUI appears on the screen containing the following buttons, as shown in Figure 1:

- Capture
- Stop

In addition, up to five radio buttons appear in the GUI, allowing the user to select from among the following five audio file types:

- AIFC
- AIFF
- AU
- SND
- WAVE

(These are the common file types supported by Java SDK version 1.4.1. A future version of the SDK might support additional file types that are not included in the above list. If so, this program will default to writing file type AU in place of the new file types that are not included in the above list.)

User's choice is limited

Only those file types that are supported by the system are presented to the user in the GUI *(see Figure 1).* Therefore, only those file types supported by the system can be selected by the user.

Capturing data from the microphone

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 common file name extension for the specified file type. You can easily modify the program to 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.

Playing back the audio data

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

Using a Java audio player

I showed you how to write a Java program to play back audio files in the lesson entitled Java Sound, Playing Back Audio Files using Java. You can also use that program to play back the audio file produced by this program.

Will discuss the program in fragments

As usual, I will discuss this program in fragments. A complete listing of the program is shown in Listing 14 near the end of the lesson.

Updated version of a previously-discussed program

The program that I will discuss in this lesson is an updated version of the program named **AudioRecorder02**, which I discussed in detail in the lesson entitled Java Sound, Capturing Microphone Data into an Audio File.

Although I will discuss the entire program briefly to establish the context, I will concentrate my detailed discussion on those aspects of the new program that were updated to make the program more robust. I recommend that you refer back to the lesson listed above for a detailed discussion of the other parts of the program.

The program named AudioRecorder03

The new program, named **AudioRecorder03**, demonstrates the use of a Java program to capture audio data from a microphone into an audio file type of the user's choosing, where the choices presented to the user are limited to only those file types supported by the system. Thus, the user is prevented from choosing file types that are not supported by the system, which would result in runtime errors if selected.

The controlling class named AudioRecorder03

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

```
public class AudioRecorder03 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();
```

Listing 1

The class definition begins by declaring *(and initializing)* several instance variables. The instance variables in Listing 1 were discussed in the earlier lesson, so I won't discuss them further here.

New instance variables

The code in Listing 2 declares two new instance variables that were not included in the program in the earlier lesson.

```
  JRadioButton[] radioBtnArray;
  AudioFileFormat.Type[] fileTypes;
```

Listing 2

An array of JRadioButton objects

The program in the earlier lesson displayed five radio buttons on the user interface, regardless of the number of audio file types supported by the system. Since the number of radio buttons was known at compile time, a different instance variable was declared to contain a reference to each of the five radio buttons.

Number of radio buttons is unknown at compile time

In this version of the program, one radio button is required for each file type supported by the system. Therefore, the number of required radio buttons is not known at compile time. The number of file types supported by the system cannot be determined until runtime.

Since the required number of radio buttons cannot be determined until runtime, it is necessary to accommodate the uncertainty in the program code.

A reference variable for an array object

The code in Listing 2 declares a reference variable capable of holding a reference to an array object of type **JRadioButton**. The actual size of the array is established at runtime when it is determined how many radio buttons are to be displayed.

An array object is instantiated at runtime, and that object's reference is stored in the reference variable named **radioBtnArray** in Listing 2. Then the references to the individual radio buttons are stored in the elements of the array object. We will see the code that accomplishes this later.

An array of tile types

The second instance variable declared in Listing 2, named **fileTypes**, is a reference variable capable of holding a reference to an array object. The array object is capable of holding references to objects of type **AudioFileFormat.Type**.

An array object will be instantiated at runtime with the size of the array equal to the number of file types supported by the system. The array object's reference will be assigned to the reference variable named **fileTypes**. A reference to one of the supported file types will be stored in each element of the array.

The main method

The **main** method shown in Listing 3 is identical to that used in the program in the earlier lesson, so I won't discuss it further.

```
  public static void main( String
args[]){
    new AudioRecorder03();
  }//end main
```

Listing 3

The constructor

The constructor begins in Listing 4. That portion of the constructor shown in Listing 4 is the same as was used in the program in the earlier lesson. If you don't understand something in Listing 4, you should refer back to the earlier lesson for a detailed discussion.

```
  public
AudioRecorder03(){//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);
```

Listing 4

Get the supported file types

Continuing with the discussion of the constructor, the code in Listings 5 through 8 is new to this version of the program.

```
    fileTypes =
AudioSystem.getAudioFileTypes();
```

Listing 5

The code in Listing 5 invokes the static **getAudioFileTypes** method of the **AudioSystem** class to get and save the file types for which file writing support is provided by the system.

Returns an array of type AudioFileFormat.Type

This method returns a reference to an array object of type **AudioFileFormat.Type**. The reference is assigned to the instance variable named **fileTypes**, which was declared in Listing 2.

According to the documentation, *"If no file types are supported, an array of length 0 is returned."*

> *(That is a possibility that I didn't explicitly take into account in this version of the program, and affords another opportunity to make the program more robust.)*

Once the statement in Listing 5 has finished execution, the program has identified the number and the types of files that can be written by the system. That information is contained in the array object referred to by the reference variable named **fileTypes**. We will make use of this information at several points later in the program.

Create and array of JRadioButton objects

Now we know *(or can easily determine)* how many radio buttons we need. We need one radio button for each element in the array of supported file types.

```
    radioBtnArray = new JRadioButton[

fileTypes.length];

    for(int cnt = 0; cnt <
fileTypes.length;

cnt++){
      String strType =
fileTypes[cnt].toString();
      if(cnt == 0){
        radioBtnArray[cnt] = new
JRadioButton(

strType,true);
      }else{
        radioBtnArray[cnt] = new
JRadioButton(

strType);
      }//end else

radioBtnArray[cnt].setActionCommand(

strType);
    }//end for loop

Listing 6
```

The code in Listing 6:

- Instantiates a new array object of type **JRadioButton**.
- Assigns the object's reference to the instance variable named **radioBtnArray** *(see Listing 2).*

- Populates each element in the array with a reference to a new **JRadioButton** object.

Populate the array

A **for** loop is used in Listing 6 to populate each of the elements in the array with a reference to a new **JRadioButton** object. There are some special requirements that apply to the radio buttons:

- It must be possible for the user to identify the file type associated with each radio button.
- It must be possible for the program to identify the file type associated with a radio button that has been selected by the user.

I accomplished both of these requirements using a **String** representation of the file type associated with each button.

During each iteration of the **for** loop in Listing 6, a **String** representation of the file type stored in the corresponding element of the array of file types was created by invoking the **toString** method on the file-type element.

User identification of the radio buttons

The string returned by the **toString** method was passed to the constructor for the corresponding **JRadioButton** object. This produced a label next to the radio button, which provides the visual relationship between the button and the file type.

> *(Note also that the constructor used for the first radio button that was instantiated requires an incoming **boolean** parameter in addition to the **String** parameter. This causes the first radio button to be in the "selected" state when the group of radio buttons first appears on the screen. See Figure 1.)*

Program identification of a selected button

The **String** returned by the **toString** method was also passed to the **setActionCommand** method of the new **JRadioButton** object in Listing 6.

If you are familiar with JavaBeans component properties, you will recognize that this sets the value of the **actionCommand** property of the radio button to the specified **String** value. This value can later be retrieved by invoking the **getActionCommand** method on a reference to the model that represents a selected radio button *(more on this later).*

Assuming that the **String** value returned by the **toString** method is unique, this provides a methodology for uniquely identifying the button that was selected by the user. This is probably a safe assumption, since the **String** value that is returned is intended to uniquely identify a specific audio file type.

Include the radio buttons in a group

As explained in the earlier lesson, the radio buttons are caused to participate in a mutually-exclusive group by adding them to a **ButtonGroup** object. This is accomplished in Listing 7, which uses a **for** loop to add each radio button to the group.

```
    for(int cnt = 0; cnt < 
fileTypes.length;

cnt++){

btnGroup.add(radioBtnArray[cnt]);
    }//end for loop

Listing 7
```

Add the radio buttons to the JPanel object

Also, as explained in the earlier lesson, simply adding the radio buttons to a **ButtonGroup** object doesn't accomplish a physical grouping of the radio buttons on the screen. This is accomplished in Listing 8 by adding the radio buttons to a **JPanel** object.

```
    for(int cnt = 0; cnt < 
fileTypes.length;

cnt++){

btnPanel.add(radioBtnArray[cnt]);
    }//end for loop

Listing 8
```

Finish the GUI and the constructor

The code in Listings 5 through 8 was different from the code in the similar program discussed in the earlier lesson, due to upgrading the program to make it more robust.

The code in Listing 9 is very similar to the code that was discussed in the earlier lesson, and won't be discussed further in this lesson.

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

    //Finish the GUI and make it visible
    getContentPane().setLayout(new 
FlowLayout());
    setTitle("Copyright 2003, 
R.G.Baldwin");
```

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

Listing 9
```

The captureAudio method

The method named **captureAudio** is invoked by the event handler on the **Capture** button *(see the boldface statement in Listing 4).* This method captures audio data from a microphone and causes that data to be saved in an audio file. The version of the **captureAudio** method used in this lesson is very similar to the version used in the corresponding program in the earlier lesson. Therefore, I will discuss this method only briefly in this lesson.

An abbreviated version of the **captureAudio** method is shown in Listing 10.

> *(Note that much of the code has been deleted from Listing 10 for brevity. The entire method can be viewed in Listing 14 near the end of the lesson.)*

```
  private void captureAudio(){
    try{
      //Code deleted for brevity.

      new CaptureThread().start();

      //Code deleted for brevity
    }//end catch
  }//end captureAudio method

Listing 10
```

Spawn a thread to do the actual work

The code in Listing 10 spawns a new **Thread** object and starts it running. The **run** method of the **Thread** object actually does the work to capture audio data from a microphone and to write that data into an audio file. The thread's **run** method will continue running and capturing audio data until the **Stop** button is clicked by the user *(see Figure 1).*

Spawning a thread is very important

When writing event-driven programs in Java, if an event handler is going to require any significant amount of time to complete, that handler should spawn a thread to do the work of responding to the event, and then return as quickly as possible. This makes it possible for the system to respond to other events that may occur on the event-handling thread.

The **captureAudio** method returns control to the event handler on the **Capture** button immediately after starting the thread. The event handler on the **Capture** button terminates and

returns very quickly thereafter, making it possible for the event-handling thread to respond when the **Stop** button is clicked by the user.

The CaptureThread class

The **Thread** object used to actually capture the audio data from the microphone and store it in an audio file is instantiated from the class named **CaptureThread**. The definition of the **CaptureThread** class and its **run** method begins in Listing 11. Portions of this class are significantly different from the similar but less-robust version discussed in the earlier lesson. I will discuss those portions that are different in detail.

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

Listing 11

The run method

The **run** method in this version begins just like the earlier version, by declaring two local variables and initializing their values to null.

The first local variable, named **fileType**, will be used later to hold a reference to the selected file type as **AudioFileFormat.Type**.

The second local variable, named **audioFile**, will be used later to hold a reference to a **File** object that represents the physical audio file.

Get the selected file type

The code in Listing 12 is significantly different from the code in the similar program in the earlier lesson.

In operation, the user selects a radio button *(see Figure 1)* from a group of radio buttons to specify the type of the audio file. The code in Listing 12 gets the selected file type identified as a **String**.

```
    String strType =
btnGroup.getSelection().

getActionCommand();
```

Listing 12

The getSelection method

Recall that all of the radio buttons belong to a **ButtonGroup** object. That object's reference is held in a reference variable named **btnGroup**.

The code in Listing 12 begins by invoking the **getSelection** method on the reference to the **ButtonGroup** object. According to Sun, this method *"Returns the model of the selected button"* as type **ButtonModel**.

What is a ButtonModel?

Swing components, such as **JRadioButton**, are created using a modified *model-view-control* paradigm. Briefly, the code behind each component consists of a model and a view *(which contains a built-in control).*

> *(If you want to know more about the general model-view-control paradigm, see the lessons on that topic on my web site.)*

The model contains the information that describes the component, while the view is responsible for rendering the component according to the information stored in the model.

What does Sun have to say?

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

> *"State Model for buttons. This model is used for check boxes and radio buttons, which are special kinds of buttons, as well as for normal buttons. For check boxes and radio buttons, pressing the mouse selects the button... In use, a UI will invoke setSelected(boolean) when a mouse click occurs over a check box or radio button. ..."*

Hopefully, Sun's description, when combined with my earlier explanation, will help you to understand the concept of a *model.*

The getActionCommand method

My objective is to first identify the selected radio button, and then to identify the audio file type associated with that button. The **getSelection** method discussed above returns a reference to the model that represents the selected radio button.

Once I have identified the model belonging to the selected radio button, I can invoke the **getActionCommand** method on that model to get the value of the **actionCommand** property.

In Listing 6 discussed earlier, I set the **actionCommand** property belonging to each radio button to a value that identifies the file type represented by that button. The code in Listing 12 retrieves that value and saves it in the **String** variable named **strType.**

Set the file type and extension

Next, I need to set the file type and file extension based on the selected radio button. The code in Listing 13 tests the specified file type against the five common audio file types supported by the Java SDK version 1.4.1. If a match is found, the file type and extension is set accordingly. If a match is not found, the file type and extension is set to the default file type AU.

```
    if(strType.equals("AIFC")){
      fileType =
AudioFileFormat.Type.AIFC;
      audioFile = new File("junk." +

fileType.getExtension());
    }else if(strType.equals("AIFF")){
      fileType =
AudioFileFormat.Type.AIFF;
      audioFile = new File("junk." +

fileType.getExtension());
    }else if(strType.equals("AU")){
      fileType =
AudioFileFormat.Type.AU;
      audioFile = new File("junk." +

fileType.getExtension());
    }else if(strType.equals("SND")){
      fileType =
AudioFileFormat.Type.SND;
      audioFile = new File("junk." +

fileType.getExtension());
    }else if(strType.equals("WAVE")){
      fileType =
AudioFileFormat.Type.WAVE;
      audioFile = new File("junk." +

fileType.getExtension());
    }else{
      System.out.println(
         "File type not recognized by
program.");
      System.out.println(
                         "Creating default
type AU");
      fileType =
AudioFileFormat.Type.AU;
      audioFile = new File("junk." +

fileType.getExtension());
    }//end else
```

Listing 13

Although the code in Listing 13 is long, it is repetitive and relatively straightforward.

The code in Listing 13 consists of a series of **if-else** statements. If the **String** representation of the file type **equals** the literal **String** in the conditional clause of an **if** statement:

- The file type is set to a matching type using a constant provided by the **AudioFileFormat.Type** class.
- A new **File** object is created that specifies the name and extension of the audio file as explained below.

The name of the file

The name and extension of the audio file are created using **String** concatenation. The name of the audio file is the literal string **"junk"**.

> *(Obviously you could add a text field to the GUI in Figure 1, and use the contents of the text field as the file name in place of **junk** if you so choose.)*

The file extension

The file extension is created by invoking the **getExtension** method on the **AudioFileFormat.Type** object created in the previous statement.

The **getExtension** method returns a reference to a **String** object that encapsulates the extension. The **String** returned by the **getExtension** method is concatenated onto the literal string **"junk"** to form the entire file name and extension.

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

> *"Obtains the common file name extension for this file type."*

The default case

In the event that the **String** representation of the file type fails to match any of the five common types, the final **else** clause is executed. The code in this clause displays a message on the screen notifying the user that the default type **AU** will be created, and then proceeds to do just that.

> *(This could happen if the **getAudioFileTypes** method of the **AudioSystem** class were to return a supported file type that is not one of the five common file types supported by the Java SDK version 1.4.1. For example, this could happen with a later version of the SDK if Sun decides to support additional file types. Unfortunately, in that case, this program would have to be modified to make it able to write the new file types. With a little additional thought, it should be possible to rewrite this program to make it handle that eventuality as well. It seems there are enumerable opportunities to make a program more robust.)*

The remaining code

The remaining code in the **Thread** class is very similar to that discussed in the program in the earlier lesson. Therefore, I won't discuss this code further in this lesson.

Similarly, the method named **getAudioFormat** is identical to that used in the program in the earlier lesson, so I won't discuss it here.

Possible audio format compatibility problems

As mentioned earlier, a complete listing of this program is provided in Listing 14 near the end of the lesson. If this program fails to run on your machine due to an audio format compatibility problem, you should examine the comments in the **getAudioFormat** method and try modifying the program to use a different audio format. I have been advised by some readers of the previous lessons that the audio format returned by the **getAudioFormat** method doesn't work well on all systems.

> *(Obviously, this is another area where the program could be made more robust by making certain that the program uses an audio format that is supported on the system.)*

Run the Program

At this point, you may find it useful to compile and run the program shown in Listing 14 near the end of the lesson. Operating instructions were provided earlier in the section entitled **Basic operation**.

If you use a media player, such as the Windows Media Player, to play back your file, 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, the program will not be able to create the new file, and a runtime error will occur.

Summary

In this lesson, I showed you how to use the **getAudioFileTypes** method of the **AudioSystem** class to limit the file-type choices presented to the user, thus eliminating the possibility that the user will select an output file type that is not supported by the system. Hopefully, this example will suggest how you can use other methods of the **AudioSystem** class to deal with other issues involving the need for improved robustness in your code.

Complete Program Listing

A complete listing of the program is shown in Listing 14.

```
/*File AudioRecorder03.java
Copyright 2003, Richard G. Baldwin

This is an update of the program named
AudioRecorder02.  This version demonstrates how
to limit the file type choices to those that are
supported by the system.

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, up to 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

Only those file formats supported by the system
are presented to the user.  Therefore, only those
file formats supported by the system can be
selected.

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.

Be sure to release the old file from the media
player before attempting to create a new file
with the same extension.  Otherwise, a runtime
error will occur when the program attempts to
create the new file.

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 AudioRecorder03 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();
  JRadioButton[] radioBtnArray;
  AudioFileFormat.Type[] fileTypes;

  public static void main( String args[]){
    new AudioRecorder03();
  }//end main

  public AudioRecorder03(){//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);

    //Get the file types for which file writing
    // support is provided by the system.
    fileTypes = AudioSystem.getAudioFileTypes();

    //Create an array of radio buttons
    radioBtnArray = new JRadioButton[
                              fileTypes.length];

    for(int cnt = 0; cnt < fileTypes.length;
                                        cnt++){
      String strType = fileTypes[cnt].toString();
      if(cnt == 0){
        radioBtnArray[cnt] = new JRadioButton(
                                  strType,true);
      }else{
        radioBtnArray[cnt] = new JRadioButton(
                                       strType);
      }//end else
      radioBtnArray[cnt].setActionCommand(
                                       strType);
    }//end for loop

    //Include the radio buttons in a group
    for(int cnt = 0; cnt < fileTypes.length;
                                        cnt++){
      btnGroup.add(radioBtnArray[cnt]);
    }//end for loop

    //Add the radio buttons to the JPanel
    for(int cnt = 0; cnt < fileTypes.length;
                                        cnt++){
      btnPanel.add(radioBtnArray[cnt]);
    }//end for loop

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

    //Finish the GUI and make it 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;

    //Get the selected file type described as
    // a String
    String strType = btnGroup.getSelection().
                              getActionCommand();
    //Set the file type and the file extension
    // based on the selected radio button.  Test
```

```
    // for the common audio file types supported
    // by Java SDK version 1.4.1.  If the type
    // doesn't match one of the common types,
    // create a file of the default type AU.
    if(strType.equals("AIFC")){
      fileType = AudioFileFormat.Type.AIFC;
      audioFile = new File("junk." +
                          fileType.getExtension());
    }else if(strType.equals("AIFF")){
      fileType = AudioFileFormat.Type.AIFF;
      audioFile = new File("junk." +
                          fileType.getExtension());
    }else if(strType.equals("AU")){
      fileType = AudioFileFormat.Type.AU;
      audioFile = new File("junk." +
                          fileType.getExtension());
    }else if(strType.equals("SND")){
      fileType = AudioFileFormat.Type.SND;
      audioFile = new File("junk." +
                          fileType.getExtension());
    }else if(strType.equals("WAVE")){
      fileType = AudioFileFormat.Type.WAVE;
      audioFile = new File("junk." +
                          fileType.getExtension());
    }else{
      System.out.println(
         "File type not recognized by program.");
      System.out.println(
                       "Creating default type AU");
      fileType = AudioFileFormat.Type.AU;
      audioFile = new File("junk." +
                          fileType.getExtension());
    }//end else

    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 AudioRecorder03.java
```

Listing 14

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-