Java Sound, Audio File Conversion

Baldwin shows you how to convert audio data from one audio file type to another.

Published: September 2, 2003
By Richard G. Baldwin

Java Programming Notes # 2024

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, Creating, Playing, and Saving Synthetic Sounds.

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 this series showed you how to:

- Create, play, and save synthetic sounds, making use of the features of the java.nio package to help with the byte manipulations.
- Use methods of the **AudioSystem** class to write more robust audio programs.
- 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.

This lesson will show you how to perform file conversions among different audio file types.

Preview

Audio file type is different from audio encoding

Numerous audio file types have been defined in recent years, including AU files, AIF files, and WAV files. However, when trying to determine if a particular audio file will be satisfactory for a particular application, simply knowing the file type isn't sufficient. You must also know how the audio data is encoded within the file.

Stated simply, the file type specification indicates how the bytes are physically arranged within the file. The encoding specification indicates how the audio information is arranged within the bytes. Not all file types can accommodate all encodings. However, many file types can accommodate several different encodings.

As it turns out, file types are the less complex of the two topics. I will deal with file types in this lesson. I will begin dealing with encodings in the next lesson.

Information on different file types

Here are descriptions of some of the file types supported by Sun's Java, as found at the High-Tech Dictionary of File Types:

- AU - A sound file format used on Sun Microsystems or other UNIX computers.

- AIF - Audio Interchange File Format or AIFF *(filename extension).* A format developed by Apple Computer for storing high-quality sampled audio and musical instrument information. It can be played on PC and Mac. *(Note that the Sun Java API treats the common filename extension for this type as AIF.)*
- WAV - Sound file. *(As you can see, the HighTech Dictionary doesn't have much to say about this file type. I will add that most of the sound files that are provided by Microsoft in a typical Windows installation are WAV files.)*

Format descriptions

You can view a technical description of the format of an AU file, including information about how the bytes are arranged in the file, at Header file for Audio, .au.

You can view a similar technical description for an AIFF file format here. Finally, you can view a technical description of the format of a WAV file at The Canonical WAVE File Format.

Of course, if you fire up your Google search engine, you can find many other descriptions of these and other file formats as well.

General information about sampled sound

You will find some very interesting information about sampled sound published by Marc Boots-Ebenfield at Sound Formats. Included on the web site is the following factoid regarding CD quality music.

> *"On a Music CD the music is sampled at 44.1 KHz using 16 bit words or 705,600 bits for each second of sound. At 8 bits to the byte that would mean that 1 second of CD quality music would occupy 88,200 bytes or 88 Kb of your floppy disc which holds 1.2 Mb of data. That means that you could hold 13 seconds of CD quality music on a floppy- (uncompressed)!"*

If the above estimate is correct, then about fifteen floppy disks would be required to contain a typical three-minute song in uncompressed CD quality format. *(That fact will be more important in the future lessons on encoding than in this lesson.)*

A non-technical aside

As another interesting factoid, The American Heritage® Book of English Usage is not very fond of this usage of the word *factoid.*

Discussion and Sample Code

The user interface

The user interface for this program is very simple. This program is designed to be executed from the command line as follows:

Usage: java AudioFileConvert01 inputFile outputFile

The program named AudioFileConvert01

Before getting into the details of the program code, I will describe the program and show you some examples produced by running the program.

This program demonstrates the ability to write a Java program to convert one audio file type to a different audio file type. Run the program by entering the following at the command line:

```
java AudioFileConvert01 inputFile outputFile
```

Input and output file types

The type of output file that is created depends on the output file name extension, such as **au**, **wav**, or **aif**.

On the other hand, the type of the input file does not depend on the input file name or extension. The actual type of the input file is determined by the program irrespective of the name of the file or the extension given to that file.

Playback of the output file

You should be able to play the output file with any standard media player that can handle the file type, or with a program written in Java, such as the program named AudioPlayer02 that was developed in an earlier lesson.

Operational examples

The following paragraphs show sample screen outputs for different input and output file types. Note that line breaks were manually inserted to force the material to fit in this narrow publication format.

Valid input file with invalid file extension

In the first example, shown in Figure 1, a valid input **wav** file named **ringout** was forced to have the invalid extension **.txt**. However, the program successfully determined the type of the **wav** file on the fly, and the **wav** file was successfully converted to an **au** file.

> *(You may recognize the primary name of this file as being one of the sound files commonly included in a standard Windows installation. I simply made a copy of the file named **ringout.wav** and changed the extension before running this experiment.)*

```
java AudioFileConvert01 ringout.txt
junk.au
```

```
 Input file: ringout.txt
 Output file: junk.au
 Output type: au
 Output type is supported
 Input file format:
 WAVE (.wav) file, byte length: 5212,
 data format: PCM_UNSIGNED, 11025.0
Hz,
 8 bit, mono, audio data
 Bytes written: 5191
 Output file format:
 AU (.au) file, byte length: 5191,
 data format: PCM_SIGNED, 11025.0 Hz,
 8 bit, mono, audio data, frame
length: 5167
Figure 1
```

Encoding information is displayed

You will see the code that produced the output in Figure 1 later when I discuss the program. As you can see from the output, the code in this program gets and displays encoding information *(PCM_UNSIGNED, 8 bit, mono, etc.)* on both the input file and the output file in addition to the file type. However, this program makes no attempt to purposely change the encoding. *(As I mentioned earlier, I will begin dealing with encoding in the next lesson.)*

Conversion of an AU file to a WAV file

In the example shown in Figure 2, the input file was a stereo **au** file produced by a sample program from an earlier lesson. The **au** file was successfully converted to a **wav** file.

```
java AudioFileConvert01 junk3.au
junk.wav
 Input file: junk3.au
 Output file: junk.wav
 Output type: wav
 Output type is supported
 Input file format:
 AU (.au) file, byte length: 64024,
 data format: PCM_SIGNED, 16000.0 Hz,
16 bit,
 stereo, big-endian, audio data,
 frame length: 16000
 Bytes written: 64044
 Output file format:
 WAVE (.wav) file, byte length: 64044,
 data format: PCM_SIGNED, 16000.0 Hz,
16 bit,
 stereo, little-endian, audio data
Figure 2
```

The fact that these files are stereo *(two-channel)* files is indicated by the encoding information that is displayed in Figure 2.

Conversion of a WAV file to an AIF file

A standard Windows monaural **wav** file was successfully converted to an **aif** file, as shown in Figure 3

```
java AudioFileConvert01 ringout.wav
junk.aif
 Input file: ringout.wav
 Output file: junk.aif
 Output type: aif
 Output type is supported
 Input file format:
 WAVE (.wav) file, byte length: 5212,
 data format: PCM_UNSIGNED, 11025.0
Hz, 8 bit,
 mono, audio data
 Bytes written: 5221
 Output file format:
 AIFF (.aif) file, byte length: 5221,
 data format: PCM_SIGNED, 11025.0 Hz,
8 bit,
 mono, audio data, frame length: 5167
Figure 3
```

An unsupported output file type

In the example shown in Figure 4, the specified output file type, **xyz**, is not supported by the Java Sound API *(nor by any other system that I am aware of).* Therefore, the program aborted, providing a list of the output file types that are supported for writing by the system.

```
java AudioFileConvert01 junk3.au
junk.xyz
 Input file: junk3.au
 Output file: junk.xyz
 Output type: xyz
 Output type not supported.
 Supported audio file types: au aif
wav
Figure 4
```

Note that the Java implementation on my system at the time of this writing only supports file types **au**, **aif**, and **wav**.

An unsupported input file type

In the example shown in Figure 5, the input file claimed by virtue of its name and extension to be a **wav** file. However, it was not a valid audio file. Rather, it was simply a text file that I renamed to cause it to impersonate a **wav** file. This caused the program to throw an **UnsupportedAudioFileException** and abort.

Once again, the program determined the type of the input file by examining the contents of the file, and not by examining the file's name or extension.

```
java AudioFileConvert01 invalidFile.wav
junk.au
 Input file: invalidFile.wav
 Output file: junk.au
 Output type: au
 Output type is supported
 javax.sound.sampled.
 UnsupportedAudioFileException: could not
get
 audio input stream from input stream
 at javax.sound.sampled.AudioSystem.
 getAudioInputStream(AudioSystem.java:756)
 at AudioFileConvert01.
 main(AudioFileConvert01.java:84)
Figure 5
```

Getting usage information

In Figure 6, the program was run with no command-line arguments, causing the program to provide usage information and abort.

```
java AudioFileConvert01
Usage: java AudioFileConvert01
                              inputFile
outputFile
Figure 6
```

This program was tested using SDK 1.4.1 under WinXP

The class named AudioFileConvert01

The controlling class for the program begins in Listing 1. As usual, I will discuss the program in fragments. You can view a listing of the entire program in Listing 11 near the end of the lesson.

The program is relatively straightforward consisting of the **main** method and the following static methods *(these methods were declared **static** so that they can be invoked from inside the **main** method):*

- **getTargetTypesSupported** - returns a list of the audio file types that can be written by the system.
- **getTargetType** - returns the type of a specified output file based on the filename extension.
- **showFileType** - Examines a **File** object representing a physical audio file and displays information about the file.

The main method

Listing 1 contains the beginning of the **main** method. The code in Listing 1 examines the number of command-line arguments entered by the user, and displays usage information if the user didn't enter any arguments.

```
public class AudioFileConvert01{

  public static void main(String[]
args){
    if(args.length != 2){
      System.out.println(
                "Usage: java
AudioFileConvert01 "
                        + "inputFile
outputFile");
      System.exit(0);
    }//end if

    System.out.println("Input file: "
+ args[0]);
    System.out.println("Output file:
"+ args[1]);

Listing 1
```

In addition, the code in Listing 1 displays the input and output file names provided by the user when those file names are entered as command-line arguments.

Get and test output file type

The output file type is determined by the filename extension provided by the user. The code in Listing 2 isolates the filename extension as type **String** and displays the extension on the screen.

```
    String outputTypeStr =

args[1].substring(args[1].

lastIndexOf(".") + 1);
    System.out.println("Output type: "
                                  +
outputTypeStr);
```

```
    AudioFileFormat.Type outputType =

getTargetType(outputTypeStr);

Listing 2
```

More importantly, the code in Listing 2 invokes the **getTargetType** method, passing the filename extension as a **String** parameter to that method.

The getTargetType method

The **getTargetType** method checks to see if the system is capable of writing the file type indicated by the extension. If so, it returns an **AudioFileFormat.Type** object matching that extension. If not, it returns **null**.

At this point, I am going to put the **main** method on hold and discuss the method named **getTargetType**.

The AudioFileFormat.Type class

Listing 3 contains the entire method named **getTargetType**.

```
  private static AudioFileFormat.Type
                 getTargetType(String
extension){
    AudioFileFormat.Type[]
typesSupported =

AudioSystem.getAudioFileTypes();
    //System.out.println("length: " +

typesSupported.length);
    for(int i = 0; i <
typesSupported.length;

i++){

if(typesSupported[i].getExtension().

equals(extension)){
        return typesSupported[i];
      }//end if
    }//end for loop
    return null;//no match
  }//end getTargetType

Listing 3
```

The first thing to note about the code in Listing 3 is that the **getTargetType** method returns a reference to an object of type **AudioFileFormat.Type**.

*(In case you are unfamiliar with the notation where there is a period in a class name, this indicates that the **Type** class is an inner class of the class named **AudioFileFormat**. If you are unfamiliar with inner classes, see the tutorial lessons on that topic on my web site.)*

What does Sun have to say?

Here is what Sun has to say about this class:

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

Static instances are provided for the following types:

- AIFC
- AIFF
- AU
- SND
- WAVE

It is interesting to note that even though five different audio file types are identified as *the common types* in this class, only three of those types are currently supported for writing on my machine running SDK 1.4.1 under WinXP.

An array of AudioFileFormat.Type object references

The code in Listing 3 invokes the method named **getAudioFileTypes**, which is a **static** method of the **AudioSystem** class. This method returns a list containing *"the file types for which file writing support is provided by the system."* This list is stored in an array of type **AudioFileFormat.Type**.

(Listing 3 contains a statement with a call to the println method that has been commented out. When this statement is enabled on my system, it reports that the length of the array is three, indicating that only three file types are currently supported for writing on my System. You will see the names of those three types later.)

The getExtension method

The **AudioFileFormat.Type** class provides a method named **getExtension**, which returns *"the common file name extension"* for an object of the type. The code in Listing 3 uses a **for** loop to search the array of **AudioFileFormat.Type** objects looking for a match to the file name extension received as an incoming parameter by the **getTargetType** method.

If a match is found, the **AudioFileFormat.Type** object corresponding to that match is returned by the **getTargetType** method. Otherwise, **null** is returned by the method.

Testing the return value

Returning our attention to the code in the **main** method, the code in Listing 4 tests to determine if a **null** value was returned by the **getTargetType** method. If not, the program displays the message:

```
Output type is supported
```

```
//Continue with main method
    if(outputType != null){
      System.out.println(
                    "Output type is
supported");
    }else{
      System.out.println(
                   "Output type not
supported.");
      getTargetTypesSupported();
      System.exit(0);
    }//end else
```

Listing 4

If a **null** value was returned by the **getTargetType** method, the program displays the following message and then invokes the method named **getTargetTypesSupported** to display a list of the file types that are supported for writing by the system.

```
Output type not supported.
```

The getTargetTypesSupported method

The purpose of the method named **getTargetTypesSupported** is to get and display a list of the file types supported for writing by the system.

Once again, I'm going to put the **main** method on hold while I discuss the method named **getTargetTypesSupported**, shown in Listing 5.

```
  private static void
getTargetTypesSupported(){
    AudioFileFormat.Type[]
typesSupported =

AudioSystem.getAudioFileTypes();
    System.out.print(
                  "Supported audio
file types:");
    for(int i = 0; i <
typesSupported.length;
```

```
i++){
      System.out.print(" " +

typesSupported[i].getExtension());
    }//end for loop
    System.out.println();
  }//end getTargetTypesSupported

Listing 5
```

Get and display common filename extensions

The code in Listing 5 shouldn't require much in the way of an explanation. This code is very similar to the code in Listing 3. In Listing 5, however, after getting an array of **AudioFileFormat.Type** objects representing the file types supported for writing by the system, the code simply gets and displays the common filename extension for each of those types.

For the example discussed previously *(see Figure 4)* where I purposely told the program to write an unsupported output file type *(xyz)*, the code in Listings 4 and 5 produced the output shown in Figure 7.

```
 Output type not supported.
 Supported audio file types: au aif
wav
Figure 7
```

> *(Note that the list of supported file types in Figure 7 includes only three of the five types identified by **static** instances of the **AudioFileFormat.type** class.)*

The input file type

Returning once again to the **main** method, the code in Listing 6 begins dealing with the input file.

> *(Note that the determination of the input file type does not depend on the file name or extension. Rather, the program determines the type of the input file by extracting information about the file from the information contained in the file itself.)*

Get an AudioInputStream object

The code in Listing 6 begins by getting a **File** object that represents the input file.

```
//Continue with main method
    File inputFileObj = new
File(args[0]);
```

```
    AudioInputStream audioInputStream
= null;
    try{
      audioInputStream = AudioSystem.

getAudioInputStream(inputFileObj);
    }catch (Exception e){
      e.printStackTrace();
      System.exit(0);
    }//end catch

Listing 6
```

Then the code in Listing 6 uses that **File** object to get an **AudioInputStream** object that can be used to read the audio data in the input file.

I have discussed code involving **AudioInputStream** objects in several previous lessons. Therefore, I won't bore you by discussing it again here.

Display file type information

The code in Listing 7 invokes the **showFileType** method for the purpose of displaying information about the input file type.

```
    System.out.println("Input file
format:");
    showFileType(inputFileObj);

Listing 7
```

The showFileType method

Once again, I'm going to put the **main** method on hold while I discuss the method named **showFileType**, shown in Listing 8.

```
  private static void
showFileType(File file){
    try{
      System.out.println(AudioSystem.

getAudioFileFormat(file));
    }catch(Exception e){
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end showFileFormat

Listing 8
```

There isn't much to the **showFileType** method. It simply invokes the method named **getAudioFileFormat**, which is a **static** method of the **AudioSystem** class, passing the **File** object that represents the input file as a parameter to the method.

The getAudioFileFormat method

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

> *"Obtains the audio file format of the specified File. The File must point to valid audio file data."*

This method returns an object of type **AudioFileFormat**, whose reference is passed to the **println** method for display.

The AudioFileFormat class

Here is what Sun has to say about an object of this class:

> *"An instance of the **AudioFileFormat** class describes an audio file, including the file type, the file's length in bytes, the length in sample frames of the audio data contained in the file, and the format of the audio data."*

As is frequently the case, this class has an overridden **toString** method, which facilitates displaying information about the contents of the object.

The screen output for a supported input file type

Figure 8 shows the screen output produced by Listings 7 and 8 for a supported audio input file of type WAV:

```
 Input file format:
 WAVE (.wav) file, byte length: 5212,
 data format: PCM_UNSIGNED, 11025.0
Hz, 8 bit,
 mono, audio data
Figure 8
```

Note that this output contains the number of channels and the sampling frequency, which is not mentioned in the quotation from sun in the previous section.

The screen output for an unsupported input file type

Figure 9 shows the screen output produced by Listing 6 when an attempt was made to get an **AudioInputStream** object on a file that was not a valid audio file. *(It was a text file of the type produced by the Windows **NotePad** program.)*

```
 javax.sound.sampled.
 UnsupportedAudioFileException: could not
get
 audio input stream from input stream
 at javax.sound.sampled.AudioSystem.
 getAudioInputStream(AudioSystem.java:756)
 at AudioFileConvert01.
 main(AudioFileConvert01.java:84)
Figure 9
```

In this case, the program didn't even make it far enough to invoke the **showFileType** method for the purpose of displaying information about the file. Rather, it threw an **UnsupportedAudioFileException** when the attempt was made to get an **AudioInputStream** object on the input file.

The bottom line on file conversion

Returning once more to the **main** method, the code in Listing 9 illustrates the bottom line on audio file conversion in Java.

> *(Note that I deleted the try and catch from Listing 9 in order to simplify the presentation. You can view that code in Listing 11 near the end of the lesson.)*

```
//Continue with main method
    int bytesWritten = 0;
    //delete try
      bytesWritten = AudioSystem.

write(audioInputStream,

outputType,
                               new
File(args[1]));
    //delete catch
    System.out.println("Bytes written:
"
                                  +
bytesWritten);
Listing 9
```

The write method of the AudioSystem class

As it turns out, doing audio file conversion using the Java Sound API is relatively simple, as long as you aren't trying to change encodings in the process. *(As mentioned earlier, I will begin discussing encodings in the next lesson.)*

> *(Much of the code in this program was provided to help you to understand what is going on. A version of the program named AudioFileConvert02, with most of the unnecessary code deleted, is shown in Listing 12 near the end of the lesson.)*

The basics of file conversion

All that is really necessary to do a file conversion using the Java Sound API is:

- Get the names of the input and output files.
- Get an object of the class **AudioFileFormat.Type**, which defines the type of the output file.
- Get an **AudioInputStream** object on the input file.
- Invoke the method named **write** shown in Listing 9 passing the above information as parameters to the **write** method.

This will cause the input file to be read, and will cause the data from the input file to be written into the output file in the specified format.

The write method

Here is what Sun has to say about the **write** method used in Listing 9, which is a **static** method of the **AudioSystem** class:

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

Can be more general

In reality, this code could be made much more general than it is in this program. For example, the **AudioInputStream** object doesn't have to be based on a file. The **AudioInputStream** object could be based on a **TargetDataLine** object, or on any **InputStream** object capable of supplying audio data according to a known **AudioFormat**.

Similarly, another overloaded version of the **write** method allows you to replace the **File** object in the third parameter with any **OutputStream** object capable of accepting a stream of audio data in a specified format.

The number of bytes written into the output file

The **write** method returns the number of bytes actually written. This value is displayed on the screen by the last statement in Listing 9.

Information about the output file format

The code in Listing 10 displays information about the output file format.

```
    System.out.println("Output file
format:");
    showFileType(new File(args[1]));
```

```
  }//end main

Listing 10
```

Figure 10 shows a sample of the output produced by Listing 10 for one of the example cases discussed earlier in this lesson:

```
 Output file format:
 WAVE (.wav) file, byte length: 64044,
 data format: PCM_SIGNED, 16000.0 Hz,
16 bit,
 stereo, little-endian, audio data
Figure 10
```

Run the Program

At this point, you may find it useful to compile and run the programs shown in Listings 11 and 12 near the end of the lesson. Operating instructions were provided earlier in the section entitled **The user interface**.

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.

Also be aware that these programs were tested using SDK version 1.4.1. Therefore, I can't be certain that they will compile and run correctly with earlier versions of Java.

Summary

In this lesson, I showed you how to convert audio data from one audio file type to another. The essential steps involved in making such a conversion are:

- Get the names of the input and output files.
- Get an object of the class **AudioFileFormat.Type**, which defines the type of the output file.
- Get an **AudioInputStream** object on the input file.
- Invoke the method named **write** shown in Listing 9 passing the above information as parameters to the **write** method.

I also explained that this program could be made much more general either by basing the **AudioInputStream** object on an **InputStream** object other than a file, or by causing the output to be an **OutputStream** other than a file.

You should be able to play the output file produced by this program with any standard media player that can handle the file type, or with a program written in Java, such as the program named AudioPlayer02 that was developed in an earlier lesson.

What's Next?

In the next lesson, I will show you how to use mu-law encoding and decoding to compress and restore 16-bit linear PCM samples.

Complete Program Listing

Complete listings of the two programs discussed in this lesson are shown in Listing 11 and Listing 12.

```
/*File AudioFileConvert01.java
Copyright 2003, R.G.Baldwin

This program demonstrates the ability to write a
Java program to convert one audio file type to a
different audio file type.

Usage: java AudioFileConvert01
                              inputFile outputFile

Output file type depends on the output file name
extension, such as au, wav, or aif.

Input file type does not depend on input file
name or extension.  Actual type of input file is
determined by the program irrespective of name
or extension.

You should be able to play the output file with
any standard media player that can handle the
file type, or with a program written in Java,
such as the program named AudioPlayer02 that was
discussed in an earlier lesson.

The following are sample screen outputs for
different input and output file types.  Note that
line breaks were manually inserted to force the
material to fit in this narrow publication
format.

In this example, the valid input wav file was
forced to have an invalid file extension.  The
wav file was successfully converted to
an au file.

java AudioFileConvert01 ringout.txt junk.au
Input file: ringout.txt
Output file: junk.au
```

```
Output type: au
Output type is supported
Input file format:
WAVE (.wav) file, byte length: 5212,
data format: PCM_UNSIGNED, 11025.0 Hz,
8 bit, mono, audio data
Bytes written: 5191
Output file format:
AU (.au) file, byte length: 5191,
data format: PCM_SIGNED, 11025.0 Hz,
8 bit, mono, audio data, frame length: 5167

In this example, the input file was a stereo au
file produced by a sample program from an earlier
lesson.  The au file was successfully converted
to a wav file.

java AudioFileConvert01 junk3.au junk.wav
Input file: junk3.au
Output file: junk.wav
Output type: wav
Output type is supported
Input file format:
AU (.au) file, byte length: 64024,
data format: PCM_SIGNED, 16000.0 Hz, 16 bit,
stereo, big-endian, audio data,
frame length: 16000
Bytes written: 64044
Output file format:
WAVE (.wav) file, byte length: 64044,
data format: PCM_SIGNED, 16000.0 Hz, 16 bit,
stereo, little-endian, audio data

In this example, the input file was a standard
Windows wav file, which was successfully
converted to an aif file.

java AudioFileConvert01 ringout.wav junk.aif
Input file: ringout.wav
Output file: junk.aif
Output type: aif
Output type is supported
Input file format:
WAVE (.wav) file, byte length: 5212,
data format: PCM_UNSIGNED, 11025.0 Hz, 8 bit,
mono, audio data
Bytes written: 5221
Output file format:
AIFF (.aif) file, byte length: 5221,
data format: PCM_SIGNED, 11025.0 Hz, 8 bit,
mono, audio data, frame length: 5167

In this example, the output file was specified
with an unsupported type.  Thus, the program
aborted, providing a list of the output file
types that are supported.
```

```
java AudioFileConvert01 junk3.au junk.xyz
Input file: junk3.au
Output file: junk.xyz
Output type: xyz
Output type not supported.
Supported audio file types: au aif wav

In this example, although the input file claimed
to be a wav file, it was not a valid audio file.
Rather, it was a text file that was renamed to
impersonate a wav file.  This caused the program
to throw a runtime exception and abort.

java AudioFileConvert01 invalidFile.wav junk.au
Input file: invalidFile.wav
Output file: junk.au
Output type: au
Output type is supported
javax.sound.sampled.
UnsupportedAudioFileException: could not get
audio input stream from input stream
at javax.sound.sampled.AudioSystem.
getAudioInputStream(AudioSystem.java:756)
at AudioFileConvert01.
main(AudioFileConvert01.java:84)

In this example, the program was run with no
command-line parameters, causing the program to
provide usage information and abort.

java AudioFileConvert01
Usage: java AudioFileConvert01
                            inputFile outputFile


Tested using SDK 1.4.1 under WinXP
*********************************************/


import java.io.*;
import javax.sound.sampled.*;

public class AudioFileConvert01{

  public static void main(String[] args){
    if(args.length != 2){
      System.out.println(
                "Usage: java AudioFileConvert01 "
                      + "inputFile outputFile");
      System.exit(0);
    }//end if

    System.out.println("Input file: " + args[0]);
    System.out.println("Output file: "+ args[1]);
```

```
    //Output file type depends on output file
    // name extension.
    String outputTypeStr =
                        args[1].substring(args[1].
                            lastIndexOf(".") + 1);
    System.out.println("Output type: "
                                  + outputTypeStr);
    AudioFileFormat.Type outputType =
                     getTargetType(outputTypeStr);
    if(outputType != null){
      System.out.println(
                      "Output type is supported");
    }else{
      System.out.println(
                   "Output type not supported.");
      getTargetTypesSupported();
      System.exit(0);
    }//end else

    //Note that input file type does not depend
    // on file name or extension.
    File inputFileObj = new File(args[0]);
    AudioInputStream audioInputStream = null;
    try{
      audioInputStream = AudioSystem.
                getAudioInputStream(inputFileObj);
    }catch (Exception e){
      e.printStackTrace();
      System.exit(0);
    }//end catch

    System.out.println("Input file format:");
    showFileType(inputFileObj);

    int bytesWritten = 0;
    try{
      bytesWritten = AudioSystem.
                        write(audioInputStream,
                              outputType,
                              new File(args[1]));
    }catch (Exception e){
      e.printStackTrace();
      System.exit(0);
    }//end catch
    System.out.println("Bytes written: "
                                   + bytesWritten);
    System.out.println("Output file format:");
    showFileType(new File(args[1]));

  }//end main

  private static void getTargetTypesSupported(){
    AudioFileFormat.Type[] typesSupported =
                 AudioSystem.getAudioFileTypes();
    System.out.print(
                  "Supported audio file types:");
```

```
    for(int i = 0; i < typesSupported.length;
                                            i++){
      System.out.print(" " +
               typesSupported[i].getExtension());
    }//end for loop
    System.out.println();
  }//end getTargetTypesSupported

  private static AudioFileFormat.Type
                getTargetType(String extension){
    AudioFileFormat.Type[] typesSupported =
                AudioSystem.getAudioFileTypes();
    for(int i = 0; i < typesSupported.length;
                                            i++){
      if(typesSupported[i].getExtension().
                             equals(extension)){
        return typesSupported[i];
      }//end if
    }//end for loop
    return null;//no match
  }//end getTargetType

  private static void showFileType(File file){
    try{
      System.out.println(AudioSystem.
                        getAudioFileFormat(file));
    }catch(Exception e){
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end showFileFormat
}//end class
```

Listing 11

```
/*File AudioFileConvert02.java
Copyright 2003, R.G.Baldwin

This program demonstrates the ability to write a
Java program to convert one audio file type to a
different audio file type.  This is an updated
version of AudioFileConvert01 in which all
unnecessary code has been removed.

Usage: java AudioFileConvert02
                            inputFile outputFile

Output file type depends on the output file name
extension, such as au, wav, or aif.

Input file type does not depend on input file
name or extension.  Actual type of input file is
determined by the program irrespective of name
```

```
or extension.

You should be able to play the output file with
any standard media player that can handle the
file type, or with a program written in Java,
such as the program named AudioPlayer02 that was
discussed in an earlier lesson.


Tested using SDK 1.4.1 under WinXP
**********************************************/


import java.io.*;
import javax.sound.sampled.*;

public class AudioFileConvert02{

  public static void main(String[] args){
    if(args.length != 2){
      System.out.println(
                "Usage: java AudioFileConvert02 "
                       + "inputFile outputFile");
      System.exit(0);
    }//end if

    AudioFileFormat.Type outputType =
         getTargetType(args[1].substring(args[1].
                          lastIndexOf(".") + 1));

    if(outputType == null){
      System.out.println(
                   "Output type not supported.");
      System.exit(0);
    }//end else

    File inputFileObj = new File(args[0]);
    AudioInputStream audioInputStream = null;
    try{
      audioInputStream = AudioSystem.
                getAudioInputStream(inputFileObj);

      AudioSystem.write(audioInputStream,
                                outputType,
                                new File(args[1]));
    }catch (Exception e){
      e.printStackTrace();
      System.exit(0);
    }//end catch

  }//end main
  //-----------------------------------------//

  private static AudioFileFormat.Type
                  getTargetType(String extension){
    AudioFileFormat.Type[] typesSupported =
```

```
                AudioSystem.getAudioFileTypes();
    for(int i = 0; i < typesSupported.length;
                                              i++){
      if(typesSupported[i].getExtension().
                               equals(extension)){
        return typesSupported[i];
      }//end if
    }//end for loop
    return null;//no match
  }//end getTargetType
  //---------------------------------------//
}//end class
```

Listing 12

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-