Audio Recording With Java

Record and Save

By
Ismael Mosquera Rivera

To Maxi, a good friend of mine.

Abstract

In this brief article, we describe the implementation of an audio recorder written in the Java programming language. The recorder was integrated in the Java IMR-LIB library; now it is part of the imr.media.audio package.

In this implementation, we used the facilities offered by the javax.sound.sampled package, which is part of the Java API in the JDK (Java Development Kit); we also used some classes in the java.io package; all of them will be explained in this paper.

Our audio recorder has support to record audio, play the recorded sound and, if we want, store it in a file; the allowed audio formats to save our recorded audio are:

- WAVE (*.wav).
- AIFF (*.aif).
- AU (*.au).

You can play your saved audio files using the iPlayer in the examples folder in the imrlib directory or with any comercial player.

We added another example in the examples folder of the imrlib directory to test that new feature; it was placed in the test_recorder folder; it is a command line application and you must have a microphone connected to your computer so that the program do not laumch a not audio input device found in your system exception.

We hope that this new feature can be useful for an interested reader.

Contents

1 Description **7**
2 Conclusions **12**
3 Appendix **13**

1 Description

We coded this audio recorder to be integrated in the Java IMR-LIB as a part of the imr.media.audio package, so that our library has been now extended having a new feature. First of all, we thought in the requirements for our recorder. Our audio recorder has the following functionality:

- We can record digital audio.
- We can play our recorded audio, so that after listening we can decided to save it or not.
- We can store the recorded audio in a file.
- We can stop recording or playing at any moment.

We also decided to allow instantiate an AudioRecorder object ppassing a value for the sample rate to a constructor as parameter; the number of channels is always 2 and the number of bits per sample is always 16. The allowed values for sample rate are 11025.0f, 22050.0f and 44100.0f. On the other hand, we must laumch the recorder and player as a thread, so that the system keeps responsive and we have freedom to stop it whenever we want.

To achieve the goal, we used some classes in the JDK API; packages java.io and javax.sound.sampled that will be explained in the rest of this short description for our audio recorder.

The code for our audio recorder was placed in the AudioRecorder.java file, so, in that archive a public class named AudioRecorder is defined. Our class where we implemented the recorder has two constructors, one without parameters and another one having one parameter for the sample rate; the first one simply calls the other with a value for sample rate equals to 44100. 0f, so, we can say that 44100.0f is the default value for sample rate.

```
package imr.media.audio;

...
...

public class AudioRecorder
{
public AudioRecorder()
{
this(44100.0f);
}

...
...

}
```

We also need some artifact to record digital audio and also a resource to play it, so that we can listen our recorded audio to decide whether we want to store it in a file. So, we have to instantiate an AudioFormat object and retrieve the needed resorces using the AudioSystem class. To record digital audio, what we need is a TargetDataLine

object, and to playback we can use a SourceDataLine object; both of them can be obtain through the AudioSystem class mentioned before.

In our second constructor, we take the sample rate value passed as parameter and make an instance of an AudioFormat object; using that AudioFormat instance, we retrieve an object to record and another to playback the recorded audio; some kind of pseudo-code can be as follows:

```
AudioFormat format = new AudioFormat(
encoding,
sampleRate,
bits_per_sample,
num_channels,
...
... );

TargetDataLine recorder = AudioSystem.getTargetDataLine(format);
SourceDataLine player = AudioSystem.getSourceDataLine(format);
```

and open them passing a buffer size value that was set before.

```
recorder.open(format, bufferSize);
player.open(format, bufferSize);
```

Now, we have an already open object to record digital audio and also another objecdt already open to perform playback; all of this is done inside our second constructor.

To record digital audio, there is the rec() method available; as we said before, we launch our recorder as a thread so, inside the rec() method we create and start such a thread.

```
(new Thread(new rec_thread_code())).start();
```

the rec_thread_code is a private inner class of the AudioRecorder one and implements the Runnable interface so, an instance of it can be passed as parameter to a Thread constructor. If there is some recorded audio, maybe we would like to listen it in order to decide to save it or not in an audio file; to perform playback of our recorded audio, we have the play() method which, as in the case of the rec() one, launches a thread to, in this case, play audio.

```
(new Thread(new play_thread_code())).start();
```

we can deduce that the play_thread_code is also a private inner class of the AudioRecorder one and implements the Runnable interface too. Following, we are going to explain these two private inner classes and illustrate their functionality with some kind of pseudo-code.

```
private class rec_thread_code implements Runnable
{

public void run()
{
start the recorder
```

```
instantiate vars
some flags
a byte array to record audio data
a ByteArrayOutputStream object to store the recorded data.
a counter to count the number of bytes recorded
while(!stopRequested)
{
record a byte data frame using our TargetDataLine object.
store it in our ByteArrayOutputStream object
}
if the recorder is active stop it.
}
}
```

The case for playback the recorded audio is similar; as we already mention before, the task is done by other private inner class.

```
private class play_thread_code implements Runnable
{

public void run()
{
start the player
instantiate vars
some flags
a byte array to read audio data
a ByteArrayInputStream object built with the recorded data in our
ByteArrayOutputStream object.
A counter to count the bytes read.
while(bytesRead < bytesRec && !stopRequested)
{
read a frame from the ByteArrayInputStream object.
Play it using our SourceDataLine object.
}
if the player is active stop it.
}
}
```

the variable to stop recording or playing is a private boolean of the AudioRecorder class, we use the stop() method to do it.

```
public void stop()
{
_stopRequested = true;
}
```

Now, the only method to complete our description for the AudioRecorder class is the one to store the recorded audio into an audio file.

To save our recorded audio in an audio file there is a method suitable to do it in the javax.sound.sampled.AudioSystem class; it takes three parameters; the first one is a javax.sound.sampled.AudioInputStream object, the second is a javax.sound.sampled.AudioFileFormat.Type and the third is a java.io.File object. In our

save(...) method, what we do first is check the file name extension so that we can get the correct AudioFileFormat.Type object; to get it, we have a private method:

```
AudioFileFormat.Type getFileType(String audioFile)
```

if after checking the extension of the file name passed as parameter it does not match any supported ones, this method returns null. Once we get a valid AudioFileFormat.Type, we can make an instance for an AudioInputStream object using the constructor:

```
AudioInputStream(InputStream stream, AudioFormat format, long length)
```

Constructs an audio input stream that has the requested format and length in sample frames, using audio data from the specified input stream.

the first parameter can be obtained from our ByteArrayOutputStream object, which has the recorded audio data:

```
ByteArrayInputStream inputData = new ByteArrayInputStream(recBuffer.toByteArray());
```

we get the second parameter from our TargetDataLine object, since it has the audio format as a member:

```
AudioFormat format = recorder.getFormat();
```

And the third parameter can be computed as follows:

```
long len = recBuffer.toByteArray().length / recorder.getFormat().getFrameSize();
```

so, some pseudo-code for our save method can be as follows:

```
public void save(String filename) throws UnsupportedAudioFileException
{
get the AudioFileFormat.Type.
AudioFileFormat.Type fileType = getFileType(filename);
if it is null throw an UnsupportedAudioFileException
 make an instance of an AudioInputStream object:
AudioInputStream audio = new AudioInputStream(inputData, format, len);
And store it in an audio file using the write method in the AudioSystem class:
Try
{
AudioSystem.write(audio, fileType, new File(filename));
}
catch(IOException e)
{
System.out.println(e);
}
}
```

now, we have finished our description for this new feature in the imr/media/audio/AudioRecorder.java. You can pick up inside the code, which is available in the src folder of the imrlib directory. Remember that all the code is fully

documented, and it was placed in the doc.zip file in the imrlib directory; you can uncompress that file to see a doc folder with all the API documentation that was generated using the ‘javadoc’ utility. Following, we recommend some readings:

Java I/O; O’Reilly.
Java Sound API Programmer’s Guide; Sun Microsystems, Inc.

2 Conclusions

We successfully implemented an audio recorder, and it also has been integrated in the imr-lib library as part of the imr.media.audio package. We tested all the functionallity offered by the AudioRecorder class which code was placed in the AudioRecorder.java file and we can say that it works smoothly.

We know that our recorder has support to save audio files in the formats:

- WAVE *.wav
- AIFF *.aif
- AU *.au

so, one improvement could be to make support for other audio file formats, as can be compressed formats like mp3, oog, etc.

On the other hand, further work could be to extend our imr-lib library adding new functionallity in the already existing packages and even add more packages to it. Any kind of contribution is welcome.

3 Appendix

In this appendix, we just comment how to execute the example for our audio recorder and, finally, we write some notes about the author. The example was placed in the examples directory, in the test_recorder folder.

The example is a command line application already compiled and you can execute the run.bat file if you are under a MSWindows command line or the run.sh script if you are under a Linux like shell. When you run the example, you will see a menu with several options:

```
1: show menu.
2: rec.
3: play.
4: stop.
5: save.
6: exit.

option:
```

as you can see, the example covers all the functionallity in the AudioRecorder class. Remember to have a microphone connected to your computer, so that the system does not launch a not found audio input device exception.

About The Author

Ismael Mosquera Rivera has a degree in Computer Science by Pompeu Fabra University http://www.upf.edu, Barcelona – Spain, 2004. He also was involved in the Clam project, http://www.clam-project.org (C++ Library for Audio and Music); unfortunately, he is affected by a severe eye disease (Retinitis Pigmenthosa), which caused him to get totally blind a number of years ago. Anyway, he finally decided to get back to the computer world, so, he installed a screen reader in his old computer. The one he installed is only available for Windows OS, but he got the MinGW compiler which has also a command line shell from where you can have a little Linux in your Windows system. So, all the textual information in a computer can be read, and he began to do research. Among the things he has done until that, the development of this audio recorder encouraged him to write a paper explaining how it was done, and this is the result. He apologizes the reader to the lack of graphics, diagrams and identation in the source code. Anyway, his expectation is that an interested reader can take benefit from this article.