Java Sound, Creating, Playing, and Saving Synthetic Sounds

Baldwin shows 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. Seven different sample sounds are provided and explained.

Published: June 17, 2003
By Richard G. Baldwin

Java Programming Notes # 2022

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, Writing More Robust Audio Programs.

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:

- 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.

Preview

What are synthetic sounds?

Synthetic sounds, *(as opposed to sounds that you record via a microphone)*, are sounds that you create by executing a mathematical algorithm.

How does this differ from MIDI sounds?

Generally speaking, *(but not entirely)*, MIDI sounds are designed to provide a computer-generated simulation of musical instruments. Even in those cases where MIDI sounds may not be intended to simulate real musical instruments, MIDI sounds are generally intended to somehow fit into the domain of making music.

Synthetic sounds, as discussed in this lesson, are more comparable to what you may consider to be *sound effects*. For example, synthetic sounds might be appropriate for including in a computer game, or for gaining someone's attention when they download your web page.

For example, the Windows operating system makes various sounds when the user performs certain operations with the mouse or keyboard *(Critical Stop, Default Beep, Exclamation, etc.)*. While some of those sounds may have been recorded via a microphone, many of those sounds appear to have been generated synthetically.

Why create synthetic sounds?

I'm publishing this lesson for several reasons. The first reason is simply that creating and listening to synthetic sounds can be lots of fun. It is fun to write a new algorithm that produces synthetic sounds, and then to listen and hear what it sounds like.

Reason 2: Creating synthetic sounds is easy

In addition to being fun, creating synthetic sounds can be relatively easy. For example, while it is also fun to create and view animated graphics, creating animated graphics requires a lot of work. Once you know how to do it, it is much easier to create interesting sounds than it is to create interesting animations.

The sample program that I will discuss in this lesson contains algorithms for creating seven different sounds. The code for each algorithm is very similar to the code for every other algorithm. None of the algorithms contain more than one page of code, and they all sound very different

Reason 3: Will need in the future

In the near future, I plan to write a tutorial lesson explaining the technical aspects of the different encoding schemes used with the different audio formats (ALAW, PCM_SIGNED, PCM_UNSIGNED, and ULAW for example).

To explain the different encoding schemes, it will be necessary to have audio data that is both deterministic and repeatable. I will apply different encoding schemes to the same deterministic audio samples, and will show numbers to explain the differences between the encoding schemes. Algorithms similar to those explained in this lesson will suffice for providing the audio data.

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.

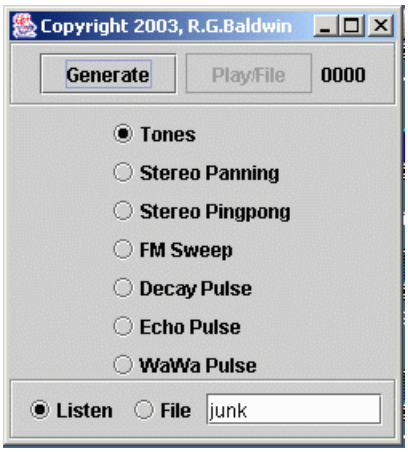

Figure 1 GUI for current version of the program

Seven different synthetic sounds

The center panel in the GUI contains radio buttons that allow the user to select from seven different synthetic sounds *(hopefully you will add many more):*

- Tones - This sound consists of a two-second monaural mixture of sinusoids at three frequencies.
- Stereo Panning - This one-second stereo sound begins in the left speaker and pans across to the right speaker with a shift in the frequency from high to low in the process
- Stereo Pingpong - This one-second stereo sound switches rapidly back and forth between the two speakers shifting frequency in the process.
- FM Sweep - This monaural sound starts at 100 Hz and does a linear frequency shift up to 1000 Hz during a two-second period.
- Decay Pulse - This two-second monaural sound is a pulse whose amplitude decays in a linear fashion from a maximum value at the beginning to zero at the end of one-second elapsed time. There is no sound during the second half of the period.
- Echo Pulse - This two-second monaural sound consists a primary pulse *(based on the Decay Pulse algorithm)* and several echoes that decrease in intensity over time.
- WaWa Pulse - This two-second monaural sound is similar to the Echo Pulse described above, except that two of the three echoes were added in with a 180-degree phase shift. This produces a decidedly different effect.

Listen or write to file

The bottom panel in the GUI contains two radio buttons that allow the user to specify whether she wants to listen to the sound immediately or to write it into an audio file of type AU. A text field is provided to allow for specifying a name for the file. *(The default file name is junk.au.)*

Generate and Play or File

The top panel in the GUI contains two buttons that allow the user to first generate a sound as specified by the radio buttons in the center panel, and then to either play the sound or write it into an audio file, depending on which radio button has been selected in the bottom panel.

Having generated a sound, the user can listen to it repeatedly and then write it into a file if desired.

The top panel also contains an elapsed-time meter that shows the length of the sound in milliseconds each time it is played.

Default case

As you can see from Figure 1, the default case on startup is to generate a *Tones* sound and *Listen* to the sound when the *Play/File* button is clicked.

Operating instructions

Here are the operating instructions for the program:

- Start the program.
- Select a sound from the center panel, or accept the *Tones* default.
- Select *Listen* or *File* in the bottom panel, or accept the *Listen* default.
- Click the *Generate* button in the top panel to generate the sound and store it in memory.
- Click the *Play/File* button in the top panel one or more times. If you previously selected *Listen* in the bottom panel, the file will be played each time you click the button. If you previously selected *File* in the bottom panel, an audio file of type AU will be written with the name showing in the text field.
- Play back the recorded audio file, if any. You should be able to play back the file using a media player such as the Windows Media Player, or a Java program such as the program named AudioPlayer02 that I discussed in an earlier lesson entitled **Java Sound, Playing Back Audio Files using Java**.

Will discuss the program in fragments

The sample program that I will discuss in this lesson is named **AudioSynth01**. As usual, I will discuss this program in fragments. A complete listing of the program is shown in Listing 49 near the end of the lesson.

Similar to previous programs

The program named **AudioSynth01** contains many elements that are similar to other programs that I have discussed in earlier lessons in this series. *(You are strongly encouraged to review those earlier lessons.)*

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 having to do with the creation, playback, and recording of synthetic sound.

The controlling class named AudioSynth01

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

```
public class AudioSynth01 extends
JFrame{

  AudioFormat audioFormat;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;

Listing 1
```

The code in Listing 1 includes the declaration of three instance variables used to create a **SourceDataLine** object that feeds data to the speakers on playback. I have discussed **SourceDataLine** objects in several previous lessons, so I won't discuss the instance variables further in this lesson.

Audio format parameters

The instance variables in Listing 2 are audio format parameters with their default values. Some of these values are modified later by the code in the algorithms that generate the sound. The values for each parameter allowed by Java SDK 1.4.1 are shown in comments following the declaration of each parameter.

```
  float sampleRate = 16000.0F;
  //Allowable
8000,11025,16000,22050,44100
  int sampleSizeInBits = 16;
  //Allowable 8,16
  int channels = 1;
  //Allowable 1,2
  boolean signed = true;
  //Allowable true,false
  boolean bigEndian = true;
  //Allowable true,false

Listing 2
```

Although I have used format parameters in several previous lessons, I haven't had much to say about them to this point. I will discuss the format parameters in the following paragraphs with respect to the impact that they have on the generation of synthetic sound.

Sample rate

I have discussed the sampling rate in general in my tutorial lesson entitled Digital Signal Processing (DSP) in Java, Sampled Time Series. Rather than to repeat that discussion, I will simply refer you to that earlier lesson.

> *(By the way, you will find an index to all of my tutorial lessons at www.DickBaldwin.com)*

The higher the sampling rate, the more samples are required for a fixed amount of time, the more memory is required, and the more computational demands are placed on the computer to be able to handle the audio data in real time.

For this lesson, I chose a sampling rate of 16000 samples per second as a reasonable compromise between the minimum allowable rate of 8000 samples per second and the highest allowable rate of 44,100 samples per second.

Sample size in bits

Java SDK 1.4.1 allows sample sizes of eight bits or 16 bits. Using signed PCM encoding, *(which I elected to use)*, an 8-bit sample can record a dynamic range of only 127 to 1. In other words, the loudest sound can only be 127 times as loud as the quietest sound, assuming that the range of sounds is perfectly balanced within the allowable range of the digitizer.

> *(In addition, Java type **short** is a natural fit for 16-bit signed PCM encoding with big-endian byte order.)*

I elected to use 16-bit signed samples *(based on type **short**)*, which provide a dynamic range of 32,767 to 1.

Number of channels

Java SDK 1.4.1 allows both monaural *(one channel)* and stereo *(two channel)* sound. I will show you how to use both in this lesson.

Signed or unsigned data

Java allows for the use of either signed or unsigned audio data. However, because Java does not support unsigned integer types *(as does C and C++)*, extra work is required to create synthetic sound for unsigned data. Therefore, I elected to use signed 16-bit data for all of the synthetic sound examples that I will discuss in this lesson.

Big-endian or little-endian

Java SDK 1.4.1 supports both big-endian and little-endian audio data. However, according to Roedy Green,

> *"Everything in Java binary format files is stored big-endian, MSB(Most Significant Byte) first. This is sometimes called network order. This is good news. This means if you use only Java, all files are done the same way on all platforms Mac, PC, Solaris, etc. You can freely exchange binary data electronically over the Internet or on CD/floppy without any concerns about endianness. The problem comes when you must exchange data files with some program not written in Java that uses little-endian order, most commonly C on the PC. Some platforms use big-endian order internally (Mac, IBM 390); some use little-endian order (Intel). Java hides that internal endianness from you. "*

Because Java inherently creates big-endian data, you must do a lot of extra work to create little-endian audio data in Java. Therefore, I elected to create all of the synthetic sounds in this lesson in big-endian order.

PCM, ALAW, or ULAW encoding

Of the available encoding schemes, linear PCM is not only the simplest, it is also the default for one of the constructors for the **AudioFormat** class. I used that constructor in this sample program. Therefore, I used linear PCM encoding for all the synthetic samples in this lesson.

I plan to publish a future lesson that will explain the differences between the different audio encoding schemes supported by Java.

An audio data buffer for synthetic data

Listing 3 shows the declaration and initialization of a **byte** array with a **length** of 64000 bytes.

```
  byte audioData[] = new
byte[16000*4];
```

Listing 3

Each of the synthetic sound data generators deposits the synthetic sound data in this array when it is invoked.

At 16-bits per sample and 16000 samples per second, this array can contain two seconds of monaural *(one-channel)* data or one second of stereo *(two-channel)* data.

For simplicity, all of the synthetic data generators in this sample program fill this array when called upon to generate synthetic sound data. Thus, the stereo samples are only half as long as the monaural samples.

> *(You can change the length of the audio data by changing the size of this array. However, for reasons that I will mention later, you should make the size of the array an even multiple of four.)*

The GUI components

I'm not going to spend much time discussing the GUI or its components. However, I will skim over the GUI code very lightly to establish the context.

The instance variables in Listing 4 hold references to components that appear in the top panel in Figure 1.

```
  final JButton generateBtn =
                          new
JButton("Generate");
  final JButton playOrFileBtn =
                         new
JButton("Play/File");
  final JLabel elapsedTimeMeter =
```

```
                                  new
JLabel("0000");

Listing 4
```

Radio buttons in the center of the GUI

The instance variables in Listing 5 hold references to radio buttons that appear in the center of the GUI in Figure 1.

```
  final JRadioButton tones =
                    new
JRadioButton("Tones",true);
  final JRadioButton stereoPanning =
              new JRadioButton("Stereo
Panning");
  final JRadioButton stereoPingpong =
             new JRadioButton("Stereo
Pingpong");
  final JRadioButton fmSweep =
                     new
JRadioButton("FM Sweep");
  final JRadioButton decayPulse =
                   new
JRadioButton("Decay Pulse");
  final JRadioButton echoPulse =
                   new
JRadioButton("Echo Pulse");
  final JRadioButton waWaPulse =
                   new
JRadioButton("WaWa Pulse");

Listing 5
```

If you update the program to add new synthetic sound data generators *(which I hope that you do)*, this is where you establish the radio buttons for the new generators.

Components in the bottom panel of the GUI

The instance variables in Listing 6 hold references to the two radio buttons and the text field that appear in the bottom panel of the GUI in Figure 1.

```
  final JRadioButton listen =
                   new
JRadioButton("Listen",true);
  final JRadioButton file =
                        new
JRadioButton("File");
  final JTextField fileName =
```

```
                          new
JTextField("junk",10);

Listing 6
```

The main method

The **main** method is shown in Listing 7. This method simply instantiates an object of the controlling class.

```
  public static void main(String
args[]){
    new AudioSynth01();
  }//end main

Listing 7
```

The constructor

The constructor begins in Listing 8.

```
  public AudioSynth01(){//constructor
    final JPanel controlButtonPanel =

new JPanel();
    controlButtonPanel.setBorder(

BorderFactory.createEtchedBorder());

Listing 8
```

The code in Listing 8 instantiates a **JPanel** object, which will contain the two buttons and the label in the top panel. Note the use of the **setBorder** method to create a border on the **JPanel** object as shown in Figure 1.

The center panel in the GUI

The code in Listing 9 instantiates a **JPanel** object that will be used to create a physical grouping of the radio buttons in the center of the GUI. The code also instantiates a **ButtonGroup** object that will be used to group the radio buttons into a logical, mutually exclusive group.

```
    final JPanel synButtonPanel = new
JPanel();
    final ButtonGroup synButtonGroup =
                                  new
```

```
ButtonGroup();
    final JPanel centerPanel = new
JPanel();

Listing 9
```

In addition, the code in Listing 9 instantiates another **JPanel** object that will be used for cosmetic purposes, to cause the radio buttons to be centered horizontally in the center of the GUI in Figure 1.

JPanel and ButtonGroup for the bottom panel of the GUI

The code in Listing 10 instantiates a **JPanel** object with an etched border to hold the components in the bottom panel of the GUI.

```
    final JPanel outputButtonPanel =

new JPanel();
    outputButtonPanel.setBorder(

BorderFactory.createEtchedBorder());
    final ButtonGroup
outputButtonGroup =
                              new
ButtonGroup();

Listing 10
```

The code in Listing 10 also instantiates a **ButtonGroup** object that will be used to group the two radio buttons in the bottom panel into a logical, mutually exclusive group.

Don't play before generating a synthetic sound

It would not work for the user to attempt to play a synthetic sound before generating such a sound. The code in Listing 11 disables the **Play/File** button *(see Figure 1)* to prevent this from happening.

```
    playOrFileBtn.setEnabled(false);

Listing 11
```

As you will see later, this button is enabled after the first synthetic sound is generated by the user.

Register action listener on the Generate button

The code in listing 12 instantiates an anonymous action listener object and registers it for action events on the **Generate** button.

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

ActionEvent e){
          //Don't allow Play during
generation

playOrFileBtn.setEnabled(false);
          //Generate synthetic data
          new
SynGen().getSyntheticData(

audioData);
          //Now it is OK for the user
to listen
          // to or file the synthetic
audio data.

playOrFileBtn.setEnabled(true);
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

Listing 12
```

I have discussed code similar to this in several previous lessons.

For purposes of this lesson, the three most significant lines of code in Listing 12 are those highlighted in boldface. Two of those lines of code first disable, and later enable the **Play/File** button. The purpose is to prevent the user from attempting to play the synthetic sound or to store it in a file while it is being generated.

Generate the synthetic sound

The most important code in Listing 12 is the code that instantiates an object of the class **SynGen**, and invokes the **getSyntheticData** method on that object. This is the statement that actually causes the synthetic sound to be generated.

Note that the method invocation passes the byte array named **audioData** to the method. This is a 64000-byte array, which will be filled with synthetic sound data when the **getSyntheticData** method returns.

At this point, I am going to depart from my discussion of the constructor for the controlling class and discuss the synthetic sound generator class named **SynGen**. I will return to a discussion of the constructor later.

The SynGen class

Listing 13 shows the beginning of the **SynGen** class. Note that this is an inner class, defined within the controlling class named **AudioSynth01**. As a result, methods of objects instantiated from this class have direct access to the instance variables of the controlling class. This results in less parameter passing than would be the case if this were a top-level class.

```
class SynGen{
  ByteBuffer byteBuffer;
  ShortBuffer shortBuffer;
  int byteLength;

Listing 13
```

An object of the **SynGen** class can be used to generate a variety of different synthetic sound signals. Each time the **getSyntheticData** method is called on an object of this class, the method will fill the **audioData** array *(see Listing 3)* with the samples for a synthetic signal.

Type ByteBuffer

Listing 13 also shows the declaration of three instance variables. The types of the first two, **ByteBuffer** and **ShortBuffer**, are new to the **java.nio** package, which was released in Java SDK, version 1.4.

> *(Among other things this means that you must be using Java version 1.4 or later to successfully compile and execute this program.*
>
> *To learn more about the capabilities of the **java.nio** package, see my tutorial lessons beginning with number 1780, Understanding the Buffer class in Java. See my web site for an index to the other lessons in that series.)*

The new capabilities of the **java.nio** package make the task of translating back and forth between signed 16-bit **short** data and bytes somewhat easier than would otherwise be the case.

The getSyntheticData method

Listing 14 shows the beginning of the **getSyntheticData** method that was invoked earlier in Listing 12.

```
  void getSyntheticData(byte[]
synDataBuffer){

    byteBuffer =
ByteBuffer.wrap(synDataBuffer);
    shortBuffer =
byteBuffer.asShortBuffer();
```

```
    byteLength = synDataBuffer.length;

Listing 14
```

Note that this method receives an incoming parameter, which is a reference to the 64000-byte array named **audioData** discussed earlier in Listing 3. This is the array in which the synthetic data generators deposit the synthetic sound data for use by other parts of the program.

Preparing the arrays for use

The code in Listing 14 begins by wrapping the incoming **audioData** array in a **ByteBuffer** object. Then a **ShortBuffer** object is created as a **short** *view* of the **ByteBuffer** object.

This makes it possible to store **short** data directly into the **audioData** array by invoking the **put** method on the **ShortData** view of the array.

Getting the length of the audio data in bytes

The code in Listing 14 also gets and saves the required length of the synthetic sound data in bytes. This value will be used in the algorithms to be discussed later.

Choose a synthetic data algorithm

The code in Listing 15 decides which synthetic data generator method to invoke based on which radio button the user has selected in the center of the GUI in Figure 1.

> *(If you add more methods for other synthetic sound types, you need to add corresponding radio buttons to the GUI and add statements here to test the new radio buttons.)*

```
    if(tones.isSelected()) tones();
    if(stereoPanning.isSelected())

stereoPanning();
    if(stereoPingpong.isSelected())

stereoPingpong();
    if(fmSweep.isSelected())
fmSweep();
    if(decayPulse.isSelected())
decayPulse();
    if(echoPulse.isSelected())
echoPulse();
    if(waWaPulse.isSelected())
waWaPulse();

  }//end getSyntheticData method
```

```
Listing 15
```

Synthetic data generator method names

The names of the synthetic data generator methods that correspond to each of the buttons are highlighted in boldface in Listing 15. I will discuss each of those methods in the paragraphs that follow.

Listing 15 also signals the end of the **getSyntheticData** method.

The tones method

Listing 16 shows the beginning of the method named **tones**, which corresponds to the radio button labeled **Tones** in Figure 1. This method generates a monaural tone, two seconds in length, consisting of the sum of three sinusoids at different frequencies.

This is a relatively simple synthetic sound. One of my main reasons for including it in this lesson is to introduce you to the concept of using sinusoids as sources for synthetic sound.

I have also discussed sinusoids in detail in my tutorial lesson entitled Periodic Motion and Sinusoids. Rather than to repeat that discussion, I will simply refer you to that earlier lesson.

```
  void tones(){
    channels = 1;//Java allows 1 or 2
    int bytesPerSamp = 2;
    sampleRate = 16000.0F;
    // Allowable
8000,11025,16000,22050,44100
    int sampLength =
byteLength/bytesPerSamp;

Listing 16
```

Monaural versus stereo data

Before getting into the use of sinusoids, however, there is some general housekeeping that I need to take care of.

To begin with, the code in Listing 16 sets the audio format instance variable named **channels** *(see Listing 2)* to a value of 1. This causes the audio data produced by this method to be later interpreted as monaural data instead of stereo data.

The number of bytes per sample

Each channel requires two 8-bit bytes per 16-bit sample. Since this method produces one-channel *(monaural)* data, Listing 16 sets the value of **bytesPerSamp** to 2. *(for stereo data, the number of bytes per sample would be 4).*

The sampling rate

As mentioned earlier, Listing 16 sets the sampling rate to 16000 samples per second as a reasonable compromise between the lowest and highest allowable sampling rates.

The length of the audio data in samples

Finally, the code in Listing 16 computes and saves the required length of the synthetic sound data in samples by dividing the length of the **audioData** array ***(byteLength**, see Listing 14)* by the number of bytes per sample ***(bytesPerSamp)**.*

The required number of samples is saved in the variable named **sampLength**.

Generate the synthetic data

Listing 17 contains a **for** loop. Each iteration of the loop:

- Generates a data sample as type **double**.
- Casts that data to type **short**.
- Invokes the **put** method on the **ShortBuffer** object to store the sample in the **byte** array named **audioData** *(see Listing 3 and Listing 12).*

```
    for(int cnt = 0; cnt < sampLength;
cnt++){
      double time = cnt/sampleRate;
      double freq = 950.0;//arbitrary
frequency
      double sinValue =
        (Math.sin(2*Math.PI*freq*time) +

Math.sin(2*Math.PI*(freq/1.8)*time) +

Math.sin(2*Math.PI*(freq/1.5)*time))/3.0;

shortBuffer.put((short)(16000*sinValue));
    }//end for loop
  }//end method tones

Listing 17
```

The loop iterates once for each required data sample, producing the required number of synthetic data samples before terminating.

(A similar loop structure is used for all of the synthetic data generator methods. Generally, it is the code within the body of the loop that determines the nature of the synthetic sound that is produced.)

Arguments to the Math.sin method

During each iteration of the **for** loop, the code in Listing 17 calculates the **time** in seconds, by dividing the sample number by the number of samples per second.

*(As discussed earlier, for this monaural case, the time will range from zero to two seconds before the **for** loop terminates.)*

This **time** value is multiplied by three different frequency values, a factor of 2, and the constant **PI** to produce three different values in radians to be used as arguments to the **Math.sin** method.

(If this terminology isn't familiar to you, please review Periodic Motion and Sinusoids before going further.)

The three values in radians are passed to three separate invocations of the **Math.sin** method to produce the sum of three separate sine values as type **double**. This sum is divided by 3 to produce the numeric average of the three sine values.

The numeric average of the three sine values is multiplied by the constant 16000, cast to type **short**, and **put** into the output array.

Why was type short used?

The type **short** is inherently a signed 16-bit type with big-endian byte order in Java, which is exactly what we need for the audio data format that I elected to use.

Why was the constant value of 16000 used?

I wanted the sound to be loud enough to hear easily. I also wanted to make certain that I didn't overflow the maximum value that can be contained in a value of type **short**.

The maximum value produced by the **Math.sin** method is 1.0. Thus, the maximum possible value in the average of the three sine values is also 1.0. The constant value of 16000 was chosen because it is approximately half the maximum value that can be contained in a value of type **short**. Thus, the maximum value that this algorithm can produce is approximately half the maximum value that can be contained in type **short**.

(These are very important considerations, because integer arithmetic overflow can destroy what might otherwise be a good synthetic sound algorithm.)

Why was a frequency 950 Hz used?

The frequency of 950 Hz was chosen because it is well within the spectral hearing range of most people, and it is within the spectral reproduction range of most computer speakers. However, the choice was arbitrary. Any other frequency that meets the above requirements should work just as well.

Play and/or modify the sound

If you generate and play the sound, you should hear a monaural tone, two seconds in length.

You can change the sound by modifying the frequency *(950)* in Listing 17, and by changing the factors used to specify different frequencies *(1.8 and 1.5)* in Listing 17.

You can change the length of the sound by changing the size of the array referred to by **audioData** in Listing 3.

> *(Increase the length of the array to make the sound longer than two seconds, and decrease the length of the array to make the sound shorter than two seconds. For reasons having to do with audio frame size, you should make certain that the size of the array is evenly divisible by four.)*

The quality of the playback

Despite everything that I have done in my attempts to improve the quality of the playback, I hear extraneous clicking noises when the sound is played back by this program, and by other Java programs written by other people. This may indicate that my computer is too slow to provide the audio data to the speakers in real time, although I'm not certain that is the cause of the problem.

In any event, I get much better playback quality by saving the synthetic sound in an audio file and using a media player such as the Windows Media Player, or the RealOne media player to play back the synthetic sound.

A visual analysis

When creating synthetic sounds, it is often useful to perform a visual analysis of the sound's waveform, and to measure the spectral content of the sound to confirm that your algorithm is performing as expected.

> *(For example, your algorithm may have experienced integer overflow without you having realized it.)*

Numerous audio tools are available for downloading that you can use for this purpose. *(If you are ambitious, you can even write your own.)* Some are free, some are not free, some are free for an evaluation period only, some are free for certain features and are not free for other features, and some are free for some other combination of the above.

AudioSuite 4.20.3

I am going to show you some pictures that were produced with the unregistered evaluation version of AudioSuite 4.20.3, which can be downloaded free of charge. This is an extremely powerful set of audio tools.

> *(Some of the features have a timeout period in the unregistered version. Fortunately, many of the features, such as waveform plotting, do not expire in the unregistered version.)*

The raw waveform for the tones method

The raw waveform of the complete two-second audio signal produced by the **tones** method is shown in Figure 2. This is a plot of signal amplitude on the vertical versus time on the horizontal.

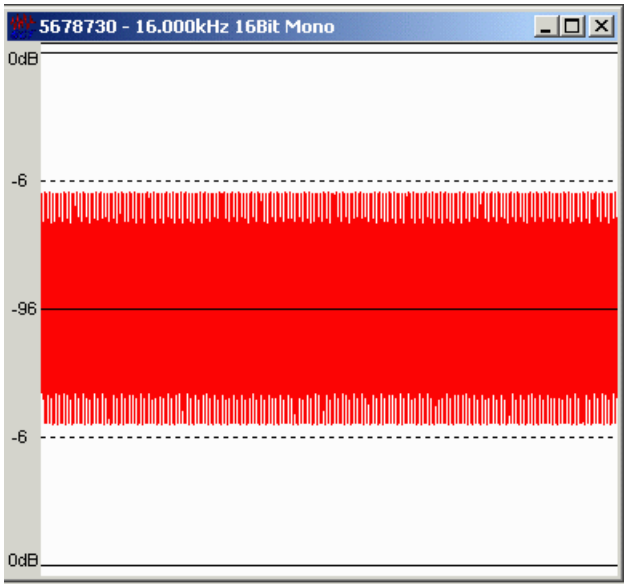

Figure 2 Raw waveform for **tones** method

Because of the horizontal compression that was required to include the entire waveform in this narrow format, the waveform shown in Figure 2 isn't very enlightening.

A more enlightening waveform

Figure 3 shows a very small portion of the beginning of the waveform greatly expanded along the horizontal *(time)* axis.

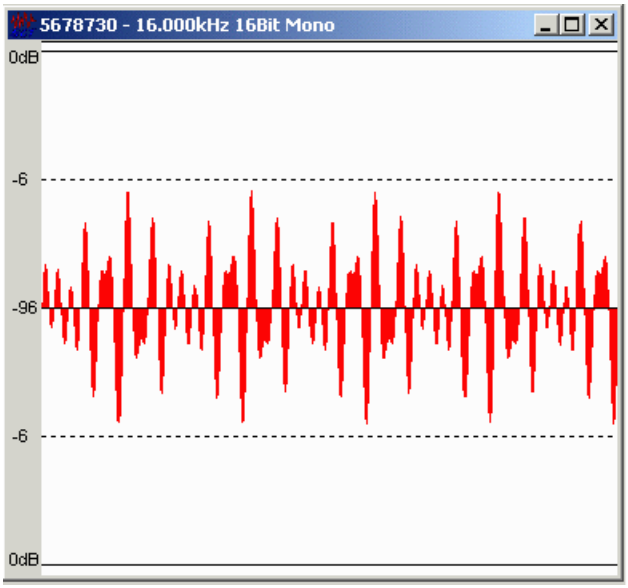

Figure 3 Time-expanded waveform for **tones** method

This representation of the waveform is much more enlightening. If you plot the sum of the three sinusoids that were added together in Listing 17 to produce the synthetic sound, this is what you should see.

> *(Note that the waveform shown in Figure 3 is periodic, with almost five periods of the waveform showing. It also exhibits an odd (as opposed to even) symmetry within each period.)*

Spectrum analysis

The synthetic sound produced by the code in Listing 17 consists of the sum of three sinusoids at frequencies of 950 Hz, 527 Hz, and 633 Hz.

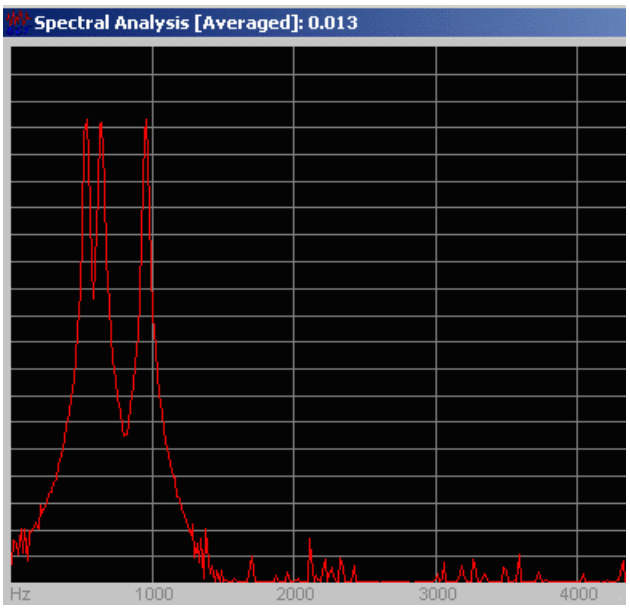

Figure 4 Spectrum analysis

Figure 4 shows the result of performing a spectrum analysis on a portion of the two-second synthetic sound signal. *(Figure 4 plots energy on the vertical axis versus frequency on the horizontal axis.)* Note the peaks in the spectrum at 950 Hz, 527 Hz, and 633 Hz.

> *(This is just what I would expect, which confirms that my algorithm behaves as I intended for it to behave.)*

The stereoPanning method

The **stereoPanning** method generates a one-second stereo speaker sweep, starting with a relatively high frequency tone on the left speaker and moving across to a lower frequency tone on the right speaker. Among other things, this method will teach you how to generate synthetic sound data that will later be interpreted as two-channel or stereo data.

The beginning of the **stereoPanning** method is shown in Listing 18.

```
  void stereoPanning(){
    channels = 2;
    int bytesPerSamp = 4;
    sampleRate = 16000.0F;
    // Allowable
8000,11025,16000,22050,44100
    int sampLength =
byteLength/bytesPerSamp;

Listing 18
```

Similar to the tones method so far

This method begins just like the **tones** method discussed earlier, except:

- The code in Listing 18 sets the value of **channels** to 2 instead of 1. As a result, the synthetic data produced by this method will be interpreted as stereo data by the playback code later.
- The code in Listing 18 sets the value of **bytesPerSamp** to 4 instead of 2. One sample is considered to contain the data for both channels *(this may more properly be referred to as a frame).* One sample for each channel requires two bytes. Thus, the sample *(frame)* for both channels requires four bytes.

An important note

*(There is nothing in the synthetic data produced by these generator methods that indicates the number of channels. These methods simply produce byte data and store that data in an array object of type **byte**. The synthetic data must be constructed by these generator methods in such a way that it will be correct for the number of channels specified in the audio format when the data is either played back or written to an audio file later. The purpose of setting **channels** in these methods is to properly set the audio format for the playback loop to be executed later in the program.)*

Creating a stereo sweep

The for loop that is typical of these generator methods begins in Listing 19.

```
    for(int cnt = 0; cnt < sampLength;
cnt++){

      double rightGain =
16000.0*cnt/sampLength;
      double leftGain = 16000.0 -
rightGain;
```

Listing 19

This method generates two channels of data. One channel will ultimately be supplied to each speaker at playback. The apparent sweep from the left speaker to the right speaker is accomplished by:

- Causing the strength of the signal applied to the left speaker to decrease from a maximum value to zero over the *(one second)* time span of the signal.
- Causing the strength of the signal applied to the right speaker to increase from zero to the maximum value over the time span of the signal.

Time-varying gains

The code in Listing 19 computes the time-varying gain to be applied to the data for each channel during each iteration of the **for** loop. This code is straightforward and shouldn't be difficult to understand. The gain for the left channel varies from 16000 to zero while the gain for the right channel varies from zero to 16000.

Time and frequency

The code in Listing 20 calculates the time and sets the frequency to be used in the arguments for the **Math.sin** methods later.

```
      double time = cnt/sampleRate;
      double freq = 600;//An arbitrary
frequency

Listing 20
```

> *(It occurred to me during the writing of this lesson that because the frequency doesn't vary with time, it would have been more logical to set the frequency value prior to entering the **for** loop. However, by the time I had that epiphany I was too far down the road to go back and change everything.)*

Generate data for the left speaker

The required format of the byte data for stereo sound signals consists of alternating left speaker and right speaker samples, beginning with the data for the left speaker. *(Again, the set of combined samples for both channels is often referred to as a frame.)* The bytes in a single frame are interpreted to be one sample for each of the two channels that occur at the same point in time.

Thus, the code in Listing 21:

- Generates a **double** sine value for the left speaker at the correct frequency for the left speaker.
- Multiplies that sine value by the time-varying **leftGain** value for the left speaker.
- Casts the **double** value to type **short**.
- Puts the two bytes that constitute the sample for the left speaker into the output array.

```
      double sinValue =

Math.sin(2*Math.PI*(freq)*time);
      shortBuffer.put(

(short)(leftGain*sinValue));

Listing 21
```

(This will be followed by putting two bytes that constitute the corresponding sample for the right speaker into the next two bytes in the output array.)

Generate data for the right speaker

The code in Listing 22:

- Generates a **double** sine value for the right speaker at the correct frequency for the right speaker *(0.8 times the frequency of the left speaker).*
- Multiplies that sine value by the time-varying **rightGain** value for the right speaker.
- Casts the **double** value to type **short**.
- Puts the two bytes that constitute the sample for the right speaker in the output array, immediately following the two bytes that were put there by the code in Listing 21.

```
      sinValue =

Math.sin(2*Math.PI*(freq*0.8)*time);
      shortBuffer.put(

(short)(rightGain*sinValue));
    }//end for loop
  }//end method stereoPanning

Listing 22
```

The code in Listing 22 also signals the end of the method named **stereoPanning**.

The waveform

Recall that this is a one-second, two-channel stereo sound. At the beginning, all of the sound comes from the left speaker, and at the end of one second, all the sound comes from the right speaker.

Between the beginning and the end, the sound coming from the left speaker decreases from maximum to zero in a liner fashion. During that same period, the sound coming from the right speaker increases from zero to maximum in a linear fashion.

Also, the pitch of the sound from the right speaker is lower than the pitch of the sound from the left speaker, because the signal for the right speaker has a lower frequency. This causes the sound to appear to sweep from the left to the right, changing pitch in the process.

This is shown by the waveforms in Figure 5 and Figure 6.

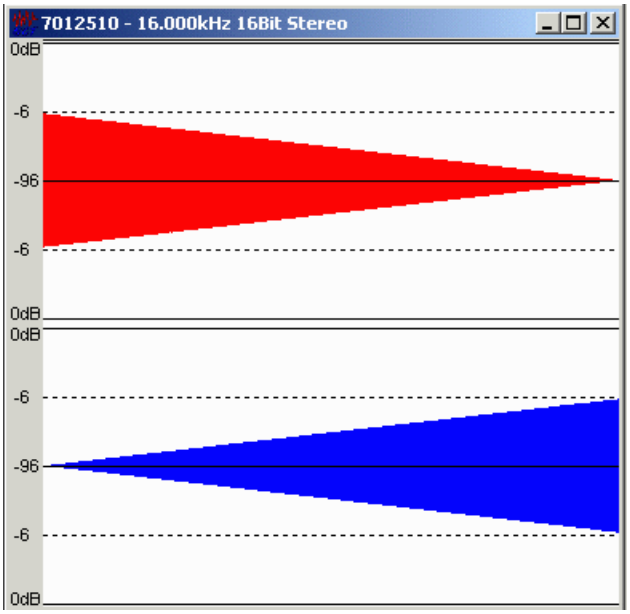

Figure 5 Raw waveforms from the stereoPanning method

Figure 5 shows the waveform of the left-channel signal in red and shows the waveform of the right-channel signal in blue. This representation of the waveforms clearly shows the change in sound level for each channel during the one-second period. However, because of the horizontal compression, Figure 5 doesn't show anything about the frequency or pitch of the sound from the two channels.

Expanded waveform

Figure 6 shows a small slice in time from both waveforms near the one-half second point, with a greatly expanded time scale.

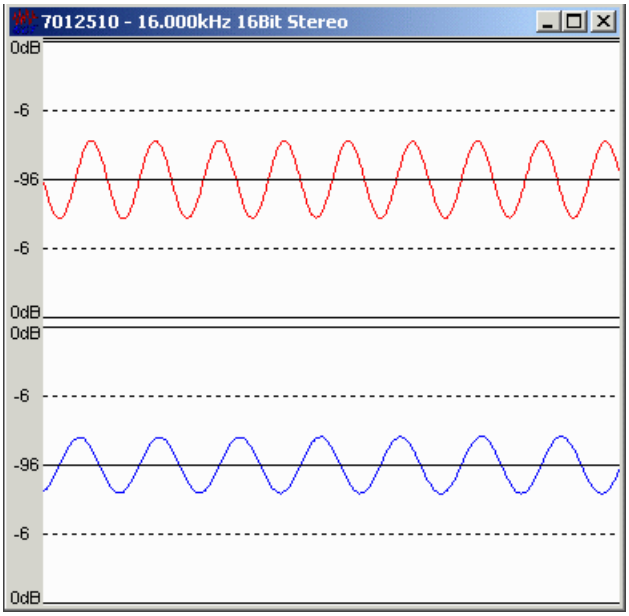

Figure 6 Expanded waveforms from the stereoPanning method

Figure 6 confirms that each of the two sounds is a simple sinusoid, as shown in Listing 21 and Listing 22. Also, Figure 6 confirms that the frequency of the sinusoid on the right channel is approximately eighty-percent of the frequency of the sinusoid on the left channel as indicated by Listing 22.

> *(There are nine positive peaks in the waveform for the left channel in Figure 6 and only 7 positive peaks in the waveform for the right channel in the same time period.)*

The stereoPingpong method

The **stereoPingpong** method uses stereo to switch a sound back and forth between the left and right speakers at a rate of about eight switches per second. On my system, this is a much better demonstration of the sound separation between the two speakers than is the demonstration produced by the **stereoPanning** method.

The sounds produced are at different frequencies. As a result, the sounds produced are similar to that of U.S. emergency vehicles.

Following discussions will be more abbreviated

Now that you understand the fundamental structure of these generator methods, the discussion of the remaining methods should go more quickly than the discussion of the first two methods.

The beginning of the **stereoPingpong** method is shown in Listing 23.

```
  void stereoPingpong(){
    channels = 2;//Java allows 1 or 2
    int bytesPerSamp = 4;//Based on
channels
    sampleRate = 16000.0F;
    // Allowable
8000,11025,16000,22050,44100
    int sampLength =
byteLength/bytesPerSamp;

    double leftGain = 0.0;
    double rightGain = 16000.0;

Listing 23
```

Time varying gains

Much of the code in Listing 23 is similar or identical to the code that you have seen in the previous generator methods.

As was the case with the method named **stereoPanning** this method generates two channels of data. Each channel will ultimately be supplied to each speaker at playback. The apparent switch from one speaker to the other speaker is accomplished by causing the strength of the signal applied to one speaker to go to zero at the same time that the strength of the signal applied to the other speaker goes to its maximum value.

The code in Listing 23 declares and initializes two variables named **leftGain** and **rightGain**, which are used for this purpose. Note that the left gain value is initialized to 0.0, while the right gain value is initialized to 16000. These values will be periodically swapped between the two channels in the **for** loop that follows.

The for loop

The typical **for** loop begins in Listing 24. During each iteration of this loop, one data sample is produced for each channel, and the samples are **put** into successive bytes in the output array.

```
    for(int cnt = 0; cnt < sampLength;
cnt++){

      if(cnt % (sampLength/8) == 0){
        //swap gain values
        double temp = leftGain;
        leftGain = rightGain;
        rightGain = temp;
      }//end if

Listing 24
```

Computing the time-varying gains

The code in Listing 24 computes the gain for each channel during each iteration of the **for** loop.

This code uses the modulus operator to swap the gain values between the left and right channels each time the iteration counter value is an even multiple of one-eighth of the sample length. For the **audioData** array of 64000 bytes, this amounts to one swap of the gain values every 2000 samples, or eight times during the one-second elapsed time of the sound.

Remainder of the for loop

The remainder of the **for** loop is shown in Listing 25.

```
      double time = cnt/sampleRate;
      double freq = 600;//An arbitrary
frequency
      //Generate data for left speaker
      double sinValue =

Math.sin(2*Math.PI*(freq)*time);
      shortBuffer.put(

(short)(leftGain*sinValue));
      //Generate data for right
speaker
      sinValue =

Math.sin(2*Math.PI*(freq*0.8)*time);
      shortBuffer.put(

(short)(rightGain*sinValue));
    }//end for loop
  }//end stereoPingpong method

Listing 25
```

The code in Listing 25 is essentially the same as code that I discussed in conjunction with an earlier generator method. Therefore, I won't discuss it further.

Waveforms

Figure 7 shows the waveforms for the left *(red)* and right *(blue)* channels of the synthetic sound produced by the method named **stereoPingpong**. In Figure 7, you can see the signals for each of the channels being switched on and off in an alternating manner.

(When the left channel is on, the right channel is off, and vice versa.)

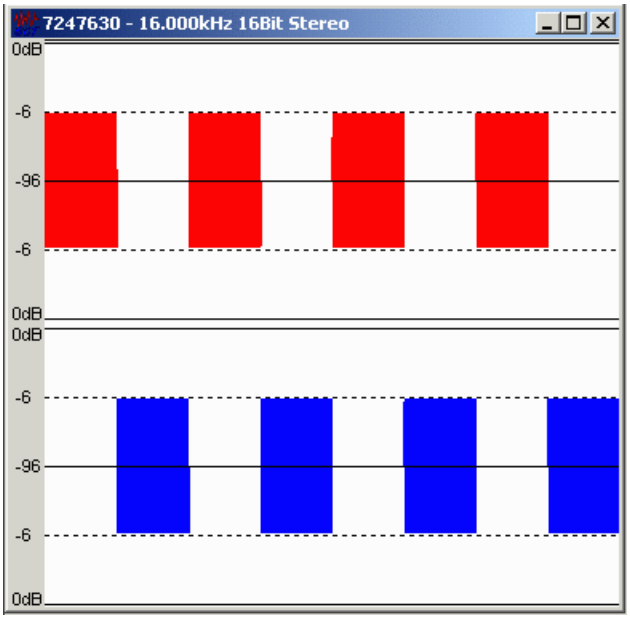

Figure 7 Waveforms from **stereoPingpong** method

Because of the horizontal compression, you can't tell anything about the frequencies involved in Figure 7.

Figure 8 shows a time-expanded waveform display of a very small time slice taken at one of the transition points where the left channel is being turned off and the right channel is being turned on.

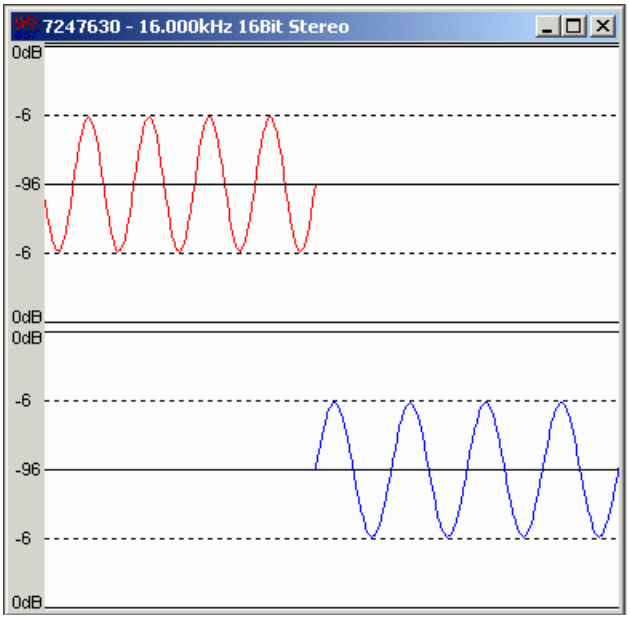

Figure 8 Time-expanded waveforms from **stereoPingpong** method

If you measure the time between the peaks on the two signals, you can confirm that the frequency of the right channel is lower than the frequency of the left channel, as indicated in Listing 25.

Some background on the fmSweep method

I have spent a good portion of my career doing digital signal processing *(DSP).* During part of that time, I worked in the submarine sonar business.

There are fundamentally two types of sonar systems, active and passive. Active sonar systems are the ones that you usually see in the movies, where a ship transmits a *ping* into the water and listens for an echo that comes back from other objects in the water, such as submarines.

Passive sonar is not frequently shown in the movies because it doesn't appear to do anything. With a passive sonar, the system, *(including the human operator),* simply listens for sounds in the water, and tries to identify those sounds as friendly or unfriendly.

Typically surface ships use active sonar and submarines use passive sonar.

Different types of pings

The actual sound pulse that is put into the water by an active sonar can take on many different waveforms. One waveform that is fairly popular is a linear FM sweep. This is a waveform where a carrier frequency undergoes frequency modulation from a low frequency to a higher frequency, or vice versa. This particular waveform has a number of desirable characteristics having to do with underwater physics, digital signal processing, Doppler effects, etc.

> *(By the way, the sound produced by an active sonar is a good example of synthetic sound. The sound is not produced by someone banging on a piece of steel with a hammer and recording the resulting sound through a microphone. Rather, the sound is produced by evaluating some sort of algorithm using some sort of electronic device, and then converting the results of that evaluation into sound pressure waves in the water.)*

The fmSweep method

This method generates a monaural linear frequency sweep that begins at 100 Hz and changes linearly up to 1000 Hz during the two-second elapsed time period of the sound.

The **fmSweep** method begins in Listing 26.

```
  void fmSweep(){
    channels = 1;//Java allows 1 or 2
    int bytesPerSamp = 2;//Based on
channels
    sampleRate = 16000.0F;
```

```
    // Allowable
8000,11025,16000,22050,44100
    int sampLength =
byteLength/bytesPerSamp;

    double lowFreq = 100.0;
    double highFreq = 1000.0;

Listing 26
```

Listing 26 initializes the typical variables. In addition, Listing 26 declares and initializes variables containing the low and high frequency values that will be used in the **for** loop that follows.

Generate the synthetic sound

The synthetic sound data is generated by the **for** loop shown in Listing 27.

```
    for(int cnt = 0; cnt < sampLength;
cnt++){
      double time = cnt/sampleRate;

      double freq = lowFreq +
                cnt*(highFreq-
lowFreq)/sampLength;
      double sinValue =

Math.sin(2*Math.PI*freq*time);

shortBuffer.put((short)(16000*sinValue));
    }//end for loop
  }//end method fmSweep

Listing 27
```

The thing that is new and different in Listing 27 is the statement that is highlighted in boldface. This statement computes a new frequency value to be used during each iteration. The frequency value changes linearly from low to high during the two-second elapsed time interval for the sound.

Waveform

Figure 9 shows a time-expanded waveform for the beginning of the signal produced by the **fmSweep** method.

> *(Note that in this format, instead of drawing lines, the graphics program fills in the entire area under the curve.)*

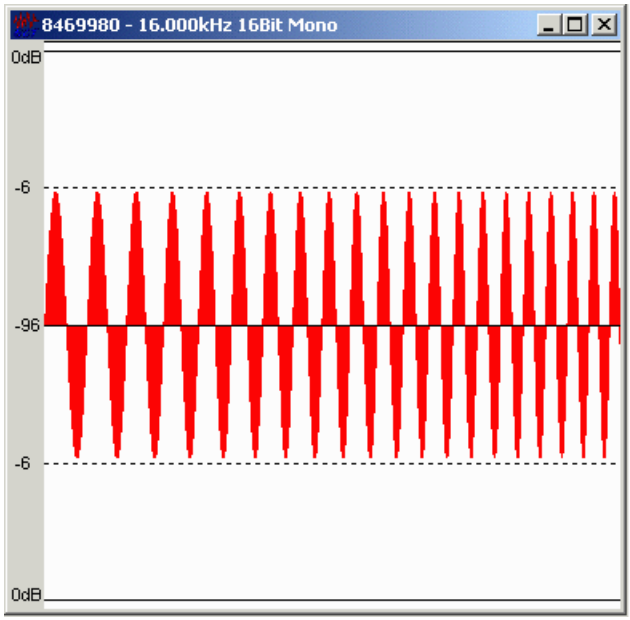

Figure 9 Time-expanded waveform from **fmSweep** method

By observing the distance between the peaks in Figure 9, this waveform confirms the implementation of the algorithm in Listing 27. The frequency of the sine wave increases with time.

The decayPulse method

The sound produced by this method is significantly different from the sounds produced by the previous methods. The previous methods have produced sounds, which had essentially constant intensity during the entire elapsed time period of the sound *(although that intensity may have been allocated between two different channels).*

The **decayPulse** method generates a monaural pulse for which the intensity decays over time. The decay function is linear with respect to time. The pulse begins with a maximum amplitude. The amplitude of the pulse decreases linearly with time and goes to zero at the end of one second.

The pulse is made up of the sum of three sinusoids at different frequencies.

The **decayPulse** method begins in Listing 28, by initializing the typical values.

```
  void decayPulse(){
    channels = 1;//Java allows 1 or 2
    int bytesPerSamp = 2;//Based on
channels
    sampleRate = 16000.0F;
    // Allowable
```

```
8000,11025,16000,22050,44100
    int sampLength =
byteLength/bytesPerSamp;

Listing 28
```

Generate the synthetic sound

The **for** loop, which is used to generate the sound, begins in Listing 29.

```
    for(int cnt = 0; cnt < sampLength;
cnt++){
      double scale = 2*cnt;
      if(scale > sampLength) scale =
sampLength;

Listing 29
```

The scale variable

The code in Listing 29 declares and initializes a variable named **scale**. The value of **scale** determines the rate of decay of the resulting pulse. Large values of **scale** cause the pulse to decay rapidly, while small values of **scale** cause the pulse to decay less rapidly.

> *(For example, to increase the rate of decay, change the literal constant 2 to a larger value. To decrease the rate of decay, change the literal constant 2 to a smaller value.)*

By virtue of the manner in which **scale** is used later in the algorithm, it is necessary to clip the value of **scale** at a maximum value of **sampLength**. This is also accomplished by the code in Listing 29.

Time-varying gain

As with some of the previous methods, this method also uses a time-varying gain value. In this case, the time-varying gain describes a decay function that decays in a linear fashion with respect to time. The statement that computes the time-varying gain value is shown in Listing 30.

> *(This value varies with time because it is based on the value of **scale**, which varies with time as shown in Listing 29.)*

```
      double gain =
             16000*(sampLength-
scale)/sampLength;

Listing 30
```

The remaining code

The remaining code in the **for** loop is shown in Listing 31. This code is very similar to what you have seen previously, and should not require further explanation.

```
      double time = cnt/sampleRate;
      double freq = 499.0;//an arbitrary
freq
      double sinValue =
        (Math.sin(2*Math.PI*freq*time) +

Math.sin(2*Math.PI*(freq/1.8)*time) +

Math.sin(2*Math.PI*(freq/1.5)*time))/3.0;

shortBuffer.put((short)(gain*sinValue));
    }//end for loop
  }//end method decayPulse

Listing 31
```

Waveform

Figure 10 shows the single-channel, two-second waveform produced by the **decayPulse** method.

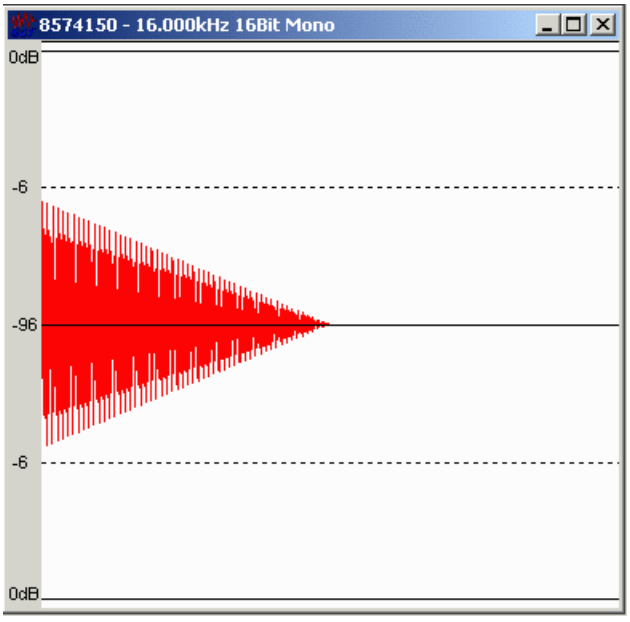

Figure 10 Waveform from **decayPulse** method

As you can see in Figure 10, the amplitude of the signals goes from maximum to zero in a linear fashion during the first one-second of the two-second interval covered by the synthetic sound. This confirms the algorithm defined in Listings 29, 30, and 31.

*(This synthetic sound makes use of sinusoidal functions that are similar to those used in the **tones** method, except that the base frequency is lower. If I were to show a time-expanded view of the waveform, it would look similar to that shown in Figure 3)*

The echoPulse method

The **echoPulse** method generates a monaural triple-frequency pulse that decays in a linear fashion with time. The synthetic sound data generated by this method begins the same as the sound data produced by the previously discussed method named **decayPulse**. However, with this method, three echoes can be heard over time with the amplitude of the echoes decreasing with time.

The beginning of the **echoPulse** method, including the initialization of the typical variables, is shown in Listing 32.

```
  void echoPulse(){
    channels = 1;//Java allows 1 or 2
    int bytesPerSamp = 2;//Based on
channels
    sampleRate = 16000.0F;
    // Allowable
8000,11025,16000,22050,44100
    int sampLength =
byteLength/bytesPerSamp;
```

Listing 32

The time-delay factors

The sound data produced by this method consists of the sum of four pulses occurring at different times. The first pulse occurs at zero time. The remaining three pulses are delayed relative to the previous pulse, and have a lower amplitude than the previous pulse.

The delays *(in samples)* that are applied to the three delayed pulses are controlled by the three variables that are declared and initialized in Listing 33.

```
    int cnt2 = -8000;
    int cnt3 = -16000;
    int cnt4 = -24000;
```

Listing 33

The first pulse begins at the beginning of the sound data. The second pulse begins at sample number 8000. The third pulse begins at sample number 16000, and the fourth pulse begins at sample number 24000. All three pulses have the same waveform with decreased amplitude.

The echoPulseHelper Method

Because this method is required to generate four separate pulses instead of just one, I elected to break a portion of the code out and put it into a helper method named **echoPulseHelper**.

The entire **echoPulseHelper Method** is shown in Listing 34.

```
  double echoPulseHelper(int cnt,int
sampLength){
    //The value of scale controls the
rate of
    // decay - large scale, fast decay.
    double scale = 2*cnt;
    if(scale > sampLength) scale =
sampLength;
    double gain =
             16000*(sampLength-
scale)/sampLength;
    double time = cnt/sampleRate;
    double freq = 499.0;//an arbitrary
freq
    double sinValue =
      (Math.sin(2*Math.PI*freq*time) +
      Math.sin(2*Math.PI*(freq/1.8)*time)
+

Math.sin(2*Math.PI*(freq/1.5)*time))/3.0;
    return(short)(gain*sinValue);
  }//end echoPulseHelper

Listing 34
```

This code in this method is identical to code in the **decayPulse** method and should not require an explanation.

Now back to the echoPulse method

Returning to the discussion of the **echoPulse** method, the code in Listing 35 shows the beginning of the **for** loop that is used to generate the synthetic sound data.

```
    for(int cnt1 = 0; cnt1 <
sampLength;

cnt1++,cnt2++,cnt3++,cnt4++){
      double val = echoPulseHelper(

cnt1,sampLength);
Listing 35
```

Update all counter values

Two things are worth noting in Listing 35. The first is that all four counter values are incremented in the update clause of the **for** loop. This not only includes the counter named **cnt1** that is used in the conditional clause of the **for** loop, it also includes the other three counters that were declared and initialized in Listing 33.

Call the echoPulseHelper method

The second thing that is worthy of note is that the code in Listing 35 calls the **echoPulseHelper** method, passing **cnt1** as a parameter, to get a value for each iteration of the **for** loop. Each call to the **echoPulseHelper** method passes a value of the counter. The value returned by the method is the correct value for that particular iteration of the **for** loop.

Get and add value for first delayed pulse

For positive values of **cnt2**, the code in Listing 36 calls the **echoPulseHelper** method to get a value for the first delayed pulse, and adds that value to the value produced earlier by the code in Listing 35.

```
      if(cnt2 > 0){
        val += 0.7 * echoPulseHelper(

cnt2,sampLength);
      }//end if

Listing 36
```

A time delay is implemented

Recall that the vale of **cnt2** was initialized to -8000 in Listing 33, and that the value of **cnt2** is incremented at the end of each iteration of the **for** loop in Listing 35.

Because of the conditional clause in the **if** statement in Listing 36, the code in Listing 36 contributes nothing to the synthetic sound data until the **for** loop has gone through the required number of iterations to cause the value of **cnt2** to go positive. At that point in time, the code in Listing 36 begins calling the **echoPulseHelper** method to get the values for another pulse. These values are scaled down by 0.7 and added to the values produced by the code in Listing 35. The result is that a second, attenuated pulse is generated and added to the first pulse at that point in time.

Add two more time-delayed pulses

The code in Listing 37 causes two more time-delayed pulses to be generated and added to the synthetic sound data beginning around sample number 16000 and sample number 24000. These

pulses are scaled by attenuation factors of 0.49 and 0.34 respectively.

```
      if(cnt3 > 0){
        val += 0.49 * echoPulseHelper(

cnt3,sampLength);
      }//end if
      if(cnt4 > 0){
        val += 0.34 * echoPulseHelper(

cnt4,sampLength);
      }//end if

      shortBuffer.put((short)val);
    }//end for loop
  }//end method echoPulse

Listing 37
```

Listing 37 also contains the requisite call to the **put** method to cause the synthetic sound data to be deposited in the **audioData** array during each iteration of the **for** loop.

Waveform

Figure 11 shows the waveform produced by the **echoPulse** method.

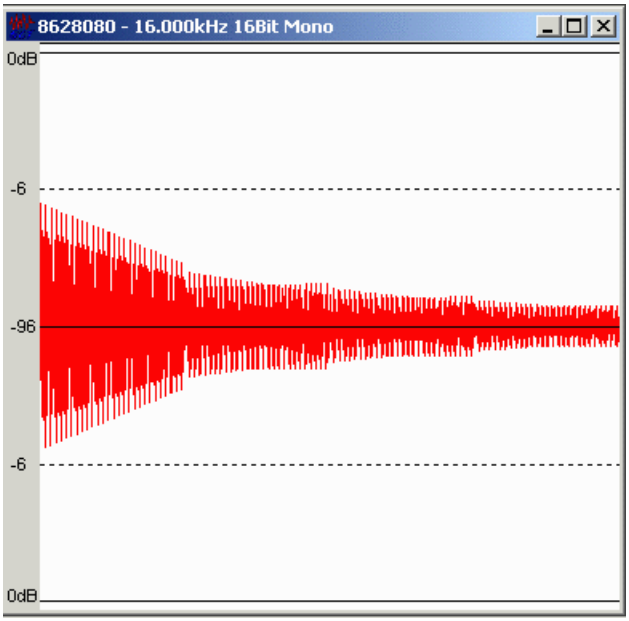

Figure 11 Waveform from **echoPulse** method

Is this what you expected?

This may not be what you expected to see for this method. You may have expected the three pulses that were added in after a time delay to be more obvious. However, this is one of the reasons that a visual analysis of the synthetic signal produced by your algorithm is very valuable.

Why aren't the pulses more obvious?

Remember that the output produced by the **echoPulseHelper** method *(Listing 34)* is simply a sequence of positive and negative values. Four versions of the output from the **echoPulseHelper** method, three with time delays, were added together.

Remember also that the underlying waveform for each of the sequences produced by the **echoPulseHelper** method is a pseudo-periodic function with an odd symmetry *(a periodic function with an amplitude that decreases linearly with time).*

Is cancellation possibility?

Were it not for the decreasing amplitude, there are certain time delays where the registration between two of the sequences would be such that the positive and negative values belonging to one sequence would exactly cancel the positive and negative values belonging to the other sequence.

Therefore, in the absence of a decreasing amplitude, when two of the sequences are added together, one with a time delay and the other without a time delay, the sum could be:

- All zero values
- Values that are exactly double the original values
- Values in between the two extremes, depending on the exact amount of time delay involved

Low side of the range

For the time delays used in this method, the values resulting from adding the sequences seem to be on the low side of that allowable range.

> *(The result will be different for the waveform for the **waWaPulse** method to be discussed later.)*

Again, this is a reason that the ability to examine the waveform is very valuable when creating synthetic sound signals. Although it would be possible to determine the result analytically by hand, that would require a very tedious effort.

The waWaPulse method

The **waWaPulse** method is identical to the method named **echoPulse**, except that the algebraic sign was switched on the amplitude of two of the echoes before adding them to the composite synthetic signal. This resulted in some differences in the synthetic sound data.

The entire **waWaPulse** method can be viewed in Listing 49 near the end of the lesson. Listing 38 shows only the **for** loop portion of the method, with the code that is different from the **echoPulse** method highlighted in boldface.

```
    for(int cnt1 = 0; cnt1 <
sampLength;

cnt1++,cnt2++,cnt3++,cnt4++){
      double val = waWaPulseHelper(

cnt1,sampLength);
      if(cnt2 > 0){
        val += -0.7 * waWaPulseHelper(

cnt2,sampLength);
      }//end if
      if(cnt3 > 0){
        val += 0.49 * waWaPulseHelper(

cnt3,sampLength);
      }//end if
      if(cnt4 > 0){
        val += -0.34 *
waWaPulseHelper(

cnt4,sampLength);
      }//end if

      shortBuffer.put((short)val);
    }//end for loop
```

Listing 38

Waveform

The waveform produced by the **waWaPulse** method is shown in Figure 12. Compare this to the waveform produced by the **echoPulse** method in Figure 11, and you should notice a striking difference between the two.

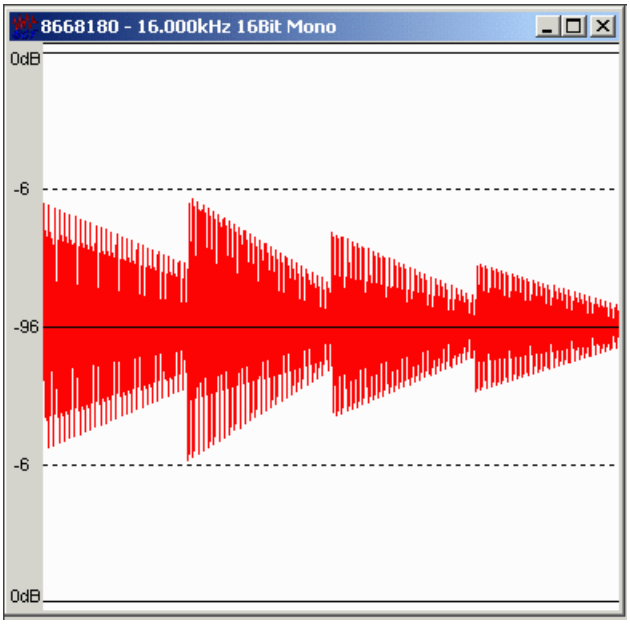

Figure 12 Waveform from **waWaPulse** method

It appears that in this case, the time delays used, in combination with the algebraic signs on the scale factors caused the waveforms to add constructively, whereas the waveforms in Figure 11 seem to have added destructively.

You should also be able to notice that difference when listing to the synthetic sounds produced by the **echoPulse** and **waWaPulse** methods.

> *(Because of constructive and destructive addition when adding delayed waveforms together, sometimes seemingly small changes can make big differences is a synthetic sound.)*

Creating your own synthetic sounds

Once again, I encourage you to use this program as a framework to create and experiment with synthetic sounds of your own design.

You may find some ideas for synthetic sound algorithms on the Audio Effects web page.

> *(Note that the Wah-Wah effect described there has no relationship to my method named **waWaPulse**. I used that simply as a unique method name. The author of the Audio Effects page used it as the actual name of an audio effect.)*

Now back to the constructor for the controlling class

That concludes the explanation of the methods that are used to generate the different kinds of synthetic sound data. Now it is time to return to the discussion of the constructor where I left off with Listing 12.

At this point, you understand how the following statement in Listing 12 causes the array named **audioData** to be filled with synthetic sound data according to the radio button that is selected in the center of Figure 1:

```
new SynGen().getSyntheticData(audioData);
```

Play or file the synthetic sound data

The code in Listing 39 instantiates an **ActionListener** object and registers it on the **Play/File** button shown in Figure 1. If you have been studying the previous lessons in this series, you will understand this style of programming using anonymous objects instantiated from anonymous classes.

In any event, in the context of this lesson, the most important code in Listing 39 is the statement that is highlighted in boldface.

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

ActionEvent e){
          //Play or file the data
synthetic data
          playOrFileData();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

Listing 39
```

The boldface statement in Listing 39 invokes the **playOrFileData** method each time the user clicks the **Play/File** button in Figure 1.

The playOrFileData method

Once again, I'm going to depart from a purely sequential explanation of the program code and discuss the method named **playOrFileData**. I will return to a discussion of the constructor later.

The **playOrFileData** method plays or files the synthetic sound data that has been generated and saved in an array in memory. If a decision is made to file the data, it is written to an audio file of type **AU**.

> *(Much of the material that follows has been discussed in previous lessons, so I will discuss it only briefly here.)*

The beginning of the **playOrFileData** method is shown in Listing 40.

```
  private void playOrFileData() {
    try{
      InputStream byteArrayInputStream
=
                          new
ByteArrayInputStream(

audioData);

Listing 40
```

The code in Listing 40 gets a **ByteArrayInputStream** object based on the synthetic sound data previously generated and stored in the array referred to by **audioData**.

Establish the audio format

The code in Listing 41 instantiates an **AudioFormat** object, based on the values stored in the audio format variables of Listing 2. This is the audio format that will be used when playing back the synthetic sound data, or when writing that data into an audio file.

```
      audioFormat = new AudioFormat(

sampleRate,

sampleSizeInBits,

channels,

signed,

bigEndian);

Listing 41
```

Recall that the values stored in some of these variables may have been modified by the code in the synthetic sound data generator methods. For example, those methods that generate monaural sound data will have set the value of **channels** to 1, while the methods that generate stereo data will have set the value of **channels** to 2.

Thus, the format values set by the generator methods will be used to either play the audio data back, or to write that data into an audio file.

Instantiate an audio input stream

The code in Listing 42 instantiates a required **AudioInputStream** object based on the data in the **ByteArrayInputStream**, and the audio format established from the default values in Listing 2 and the modified values that were set by the synthetic sound data generator program that

produced the data.

```
      audioInputStream = new
AudioInputStream(

byteArrayInputStream,
                    audioFormat,

audioData.length/audioFormat.

getFrameSize());
```

Listing 42

Get a SourceDataLine object

A **SourceDataLine** object handles the actual real-time delivery of the data to the speakers. The code in Listing 43 gets a **SourceDataLine** object.

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

SourceDataLine.class,

audioFormat);

      //Geta SourceDataLine object
      sourceDataLine =
(SourceDataLine)

AudioSystem.getLine(

dataLineInfo);
```

Listing 43

I have discussed code similar to that in Listing 43 in several previous lessons, so I won't discuss it further here.

Play the data, or write it into an audio file

The code in Listing 44 examines the radio buttons at the bottom of the GUI in Figure 1 to decide whether to play the synthetic sound data back immediately, or to write that data into an audio file of type **AU**.

```
      if(listen.isSelected()){
```

```
        new ListenThread().start();
      }else{
        //Write the synthetic data to
an audio
        // file of type AU.

Listing 44
```

If the user has selected the **Listen** button on the bottom of the GUI, the code in Listing 44 instantiates a **ListenThread** object and starts it running to play back the synthetic sound data immediately.

Otherwise, the code in Listing 44 writes the synthetic sound data to an audio file of type **AU**.

The ListenThread class

The code in the **ListenThread** method is so similar to code used to play back audio data in previous lessons that there is no point in discussing it here. You can view a complete listing of the **ListenThread** class in Listing 49 near the end of the lesson. If you don't understand that code, please go back and review the previous lessons in this series.

Playback quality

Some comments regarding playback quality are in order. One of the aspects of playing back synthetic sound data *(as opposed to microphone data)* is that you can know exactly what it should sound like, and you can play the same data back repeatedly and listen to it more than once. This makes it possible to identify playback quality problems that might not be as obvious when playing back microphone data.

My computer is several years old, and is not very fast. Whenever I use this program to play back the synthetic data, I hear clicks in the playback that are not in the data.

> *(I can confirm that the clicks are not in the data by saving the data into a file and playing it back using a media player such as the Windows Media Player, or the RealOne Player.)*

Although I'm not certain, this may indicate that my computer is incapable of delivering the audio data to the speakers in real time using the playback loop in the **ListenThread** class.

Furthermore, I have experimented with similar playback loops written by others, including code snippets that are available on the Sun site, and am unable to eliminate this problem.

I mention this here so that you can be on the lookout for similar problems when you compile and execute this program on your system.

An audio output file of type AU

Now, going back and picking up with the **else** clause in Listing 44, I will explain the code that writes the synthetic sound data into an audio file of type **AU**.

The code in Listing 45 disables both of the buttons at the top of the GUI in Figure 1 to prevent them from firing action events while the data is being written to the disk file.

```
        generateBtn.setEnabled(false);

playOrFileBtn.setEnabled(false);
```

Listing 45

Write the file

The code that writes the data into the audio file is shown in Listing 46. This code is very simple.

```
        try{
          AudioSystem.write(
                    audioInputStream,

AudioFileFormat.Type.AU,
                    new
File(fileName.getText() +

".au"));
        }catch (Exception e) {
          e.printStackTrace();
          System.exit(0);
        }//end catch
```

Listing 46

This code makes use of the static **write** method of the **AudioSystem** class to transfer the data from the **AudioInputStream** object *(provided as the first parameter)* to the audio file.

> *(The **AudioInputStream** object was instantiated in Listing 42.)*

The type of the audio file is specified as the second parameter to the **write** method.

> *(This code writes an audio file of type AU by default. If your system doesn't support that file type, you can easily write a different file type by modifying the second parameter.)*

The name of the audio file is extracted from the text field at the bottom of the GUI in Figure 1, and provided as the third parameter to the **write** method.

Enable the Generate and Play/File buttons

After the file has been written, the code in Listing 47 enables both of the buttons at the top of the GUI in Figure 1 to get the system ready for another operation.

```
        generateBtn.setEnabled(true);

playOrFileBtn.setEnabled(true);
      }//end else

Listing 47
```

Except for a **catch** block that you can view in Listing 49 near the end of the lesson, the code in Listing 47 signals the end of the **playOrFile** method.

Return to the constructor again

Returning once again to the place in the constructor where I left off in Listing 39, the code in Listing 48:

- Adds two buttons and a text field to a panel, which will appear at the top of the GUI.
- Adds seven radio buttons to a mutually exclusive group, which will appear in the center of the GUI. If you add a new generator method to the program, you will need to create a new radio button and add it to this group.
- Adds the seven radio buttons to a panel, which will appear centered horizontally in the center of the GUI. You will also need to make an addition here if you add a new generator method to the program.
- Adds two radio buttons to a mutually exclusive group, which will appear at the bottom of the GUI.
- Adds the two radio buttons and a text field to a panel, which will appear at the bottom of the GUI.
- Adds the three panels to the content pane at the North, Center, and South locations in the JFrame object that constitutes the GUI.
- Takes care of a few more odds and ends necessary to make the GUI appear on the screen with the correct title, correct size, etc.

The code to accomplish these tasks is straightforward, so I won't discuss it in detail here.

```
//Continue discussion of the constructor
here

    //Add two buttons and a text field to
a
    // panel in the North of the GUI.
    controlButtonPanel.add(generateBtn);

controlButtonPanel.add(playOrFileBtn);
```

```
controlButtonPanel.add(elapsedTimeMeter);

    //Add radio buttons to a mutually
exclusive
    // group in the Center of the GUI.
Make
    // additions here if you add new
synthetic
    // generator methods.
    synButtonGroup.add(tones);
    synButtonGroup.add(stereoPanning);
    synButtonGroup.add(stereoPingpong);
    synButtonGroup.add(fmSweep);
    synButtonGroup.add(decayPulse);
    synButtonGroup.add(echoPulse);
    synButtonGroup.add(waWaPulse);

    //Add radio buttons to a panel and
    // center it in the Center of the
GUI. Make
    // additions here if you add new
synthetic
    // generator methods.
    synButtonPanel.setLayout(
                              new
GridLayout(0,1));
    synButtonPanel.add(tones);
    synButtonPanel.add(stereoPanning);
    synButtonPanel.add(stereoPingpong);
    synButtonPanel.add(fmSweep);
    synButtonPanel.add(decayPulse);
    synButtonPanel.add(echoPulse);
    synButtonPanel.add(waWaPulse);

    //Note that the centerPanel has
center
    // alignment by default.
    centerPanel.add(synButtonPanel);

    //Add radio buttons to a mutually
exclusive
    // group in the South of the GUI.
    outputButtonGroup.add(listen);
    outputButtonGroup.add(file);

    //Add radio buttons to a panel in
    // the South of the GUI.
    outputButtonPanel.add(listen);
    outputButtonPanel.add(file);
    outputButtonPanel.add(fileName);

    //Add the panels containing
components to the
    // content pane of the GUI in the
appropriate
    // positions.
```

```
    getContentPane().add(

controlButtonPanel,BorderLayout.NORTH);
    getContentPane().add(centerPanel,

BorderLayout.CENTER);

getContentPane().add(outputButtonPanel,

BorderLayout.SOUTH);

    //Finish the GUI.  If you add more
radio
    // buttons in the center, you may
need to
    // modify the call to setSize to
increase
    // the vertical component of the GUI
size.
    setTitle("Copyright 2003,
R.G.Baldwin");

setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(250,275);
    setVisible(true);
  }//end constructor
  //--------------------------------------
------//
}//end outer class AudioSynth01.java
```

Listing 48

The code in Listing 48 also signals the end of the constructor and the end of the program.

Run the Program

At this point, you may find it useful to compile and run the program shown in Listing 49 near the end of the lesson. Operating instructions were provided earlier in the section entitled **Operating instructions**.

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 this program makes use of Java features in the **java.nio** package, which was first released in Java version 1.4. Therefore, you must be running version 1.4 or later to successfully compile and run this program.

Summary

In this lesson, I showed you how to create synthetic sound data and how to play it back immediately, or to save it in an audio file of type **AU**.

Because this lesson is somewhat long and complex, I will recap the essence of creating, playing, and filing synthetic sound in this summary section.

First you need an algorithm to create the data

To create synthetic sound, you must write an algorithm that will place bytes of synthetic sound data into an array of type **byte**. The values of the bytes must represent the synthetic sound samples.

Keep the audio format in mind

When you create the bytes that represent the synthetic sound samples, you must keep in mind the audio format that will be used to play back the data, or to save the data into an audio file. You must arrange the bytes in the byte array in a manner that is consistent with that format.

Audio format attributes

An audio format consists of the following attributes. *(The choices supported by Java version 1.4 are listed.)*

- Encoding scheme, ALAW, PCM_SIGNED, PCM_UNSIGNED, OR ULAW
- Sample rate, 8000, 11025, 16000, 22050, or 44100 samples per second.
- Sample size, 8 bits or 16 bits
- Number of channels, 1 or 2
- Signed or unsigned for PCM encoding.
- Big-endian or little-endian byte order

Some formats are easy

Some formats are much easier to handle in Java than others. This is particularly true if your algorithm requires the use of arithmetic operations.

For example, Java data of type **short** is naturally compatible with PCM_SIGNED, 16-bit, big-endian format.

> *(Due to its ease of use, this is the format that was used for all the samples in this lesson.)*

Java data of type **byte** is naturally compatible with PCM_SIGNED, 8-bit, big-endian format.

> *(This format is also relatively easy to use, but it has very limited dynamic range. Integer overflow is a constant potential problem when doing 8-bit*

arithmetic. For that reason, this format was not used for any of the samples in this lesson.)

Some formats are more difficult

If you want to use ALAW or ULAW encoding, PCM_UNSIGNED, or little-endian byte order, you are going to have to expend some extra programming effort to convert the data that is naturally produced by Java arithmetic operations to the other format parameters.

Generate the synthetic sound data

Having defined an algorithm, and having chosen an audio format, you must generate the synthetic sound data samples and store them in the **byte** array with an arrangement that matches the chosen format parameters.

The java.nio package can be very useful

If you are creating 16-bit audio data samples as type **short**, you can use the capabilities of the **java.nio** package to help you with the translation from 16-bit data to bytes in the array. Without the **java.nio** capabilities, you would probably need to perform bitwise operations to handle that translation.

Arrangement for monaural and stereo data samples

For single-channel *(monaural)* data, the audio data samples follow one another in the **byte** array.

For two-channel *(stereo)* data, the data in the byte array must consist of alternating data samples from each of the two channels, beginning with a sample from the left channel.

Make certain that the size of the **byte** array is correct for an integer number of samples for the number of channels specified in the format.

(A byte array size that is a multiple of four bytes should handle both monaural and stereo data for either 8-bit or 16-bit samples.)

Playback or file writing

To playback the synthetic sound data, or to write it into an audio file, you will need to:

- Instantiate an **AudioFormat** object using the format parameters that you used to arrange your data in the **byte** array.
- Instantiate a **ByteArrayInputStream** object based on the **byte** array that contains your data samples.
- Instantiate an **AudioInputStream** object based on your **ByteArrayInputStream** object and your **AudioFormat** object.

File writing only

To write the data to an audio file, invoke the **write** method of the **AudioSystem** class, passing the following as parameters:

- Your **AudioInputStream** object.
- The audio file type as a constant defined in the **AudioFileFormat.Type** class.
- A **File** object that supplies the name and extension for your file.

Playback of synthetic sound data

Having instantiated your **AudioFormat** object and your **AudioInputStream** object from above, to play back the data from within the same program:

- Instantiate a **DataLine.Info** object that describes a **SourceDataLine** according to the **AudioFormat** object.
- Get and save a **SourceDataLine** object by invoking the **getLine** method of the **AudioSystem** class, passing your **DataLine.Info** object as a parameter.
- Spawn a thread that uses a playback loop to transfer the data from the **AudioInputStream** object to the **SourceDataLine** object in real time. An example of such a thread has been discussed in several previous lessons, and is also provided in the class definition for the **ListenThread** class in Listing 49 near the end of the lesson.

Complete Program Listing

A complete listing of the program is shown in Listing 49.

```
/*File AudioSynth01.java
Copyright 2003, R.G.Baldwin

This program demonstrates the ability to create
synthetic audio data, and to play it back
immediately, or to store it in an AU file for
later playback.

A GUI appears on the screen containing the
following components in the North position:

Generate button
Play/File button
Elapsed time meter (JTextField)

Several radio buttons appear in the Center
position of the GUI.  Each radio button selects
a different format for synthetic audio data.

The South position of the GUI contains the
following components:
```

```
Listen radio button
File radio button
File Name text field

Select a radio button from the Center and click
the Generate button.  A short segment of
synthetic audio data will be generated and saved
in memory.  The segment length is two seconds
for monaural data and one second for stereo data,
at 16000 samp/sec and 16 bits per sample.

To listen to the audio data, select the Listen
radio button in the South position and click the
Play/File button.  You can listen to the data
repeatedly if you so choose.  In addition to
listening to the data, you can also save it in
an audio file.

To save the audio data in an audio file of type
AU, enter a file name (without extension) in the
text field in the South position, select the
File radio button in the South position, and
click the Play/File button.

You should be able to play the audio file back
with any standard media player that can handle
the AU 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 1.4.0 under Win2000
**********************************************/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.sound.sampled.*;
import java.io.*;
import java.nio.channels.*;
import java.nio.*;
import java.util.*;

public class AudioSynth01 extends JFrame{

  //The following are general instance variables
  // used to create a SourceDataLine object.
  AudioFormat audioFormat;
  AudioInputStream audioInputStream;
  SourceDataLine sourceDataLine;

  //The following are audio format parameters.
  // They may be modified by the signal generator
  // at runtime.  Values allowed by Java
  // SDK 1.4.1 are shown in comments.
  float sampleRate = 16000.0F;
  //Allowable 8000,11025,16000,22050,44100
```

```
  int sampleSizeInBits = 16;
  //Allowable 8,16
  int channels = 1;
  //Allowable 1,2
  boolean signed = true;
  //Allowable true,false
  boolean bigEndian = true;
  //Allowable true,false

  //A buffer to hold two seconds monaural and one
  // second stereo data at 16000 samp/sec for
  // 16-bit samples
  byte audioData[] = new byte[16000*4];

  //Following components appear in the North
  // position of the GUI.
  final JButton generateBtn =
                          new JButton("Generate");
  final JButton playOrFileBtn =
                        new JButton("Play/File");
  final JLabel elapsedTimeMeter =
                               new JLabel("0000");

  //Following radio buttons select a synthetic
  // data type.  Add more buttons if you add
  // more synthetic data types.  They appear in
  // the center position of the GUI.
  final JRadioButton tones =
                  new JRadioButton("Tones",true);
  final JRadioButton stereoPanning =
              new JRadioButton("Stereo Panning");
  final JRadioButton stereoPingpong =
            new JRadioButton("Stereo Pingpong");
  final JRadioButton fmSweep =
                    new JRadioButton("FM Sweep");
  final JRadioButton decayPulse =
                 new JRadioButton("Decay Pulse");
  final JRadioButton echoPulse =
                 new JRadioButton("Echo Pulse");
  final JRadioButton waWaPulse =
                 new JRadioButton("WaWa Pulse");

  //Following components appear in the South
  // position of the GUI.
  final JRadioButton listen =
                 new JRadioButton("Listen",true);
  final JRadioButton file =
                        new JRadioButton("File");
  final JTextField fileName =
                       new JTextField("junk",10);

  //----------------------------------------//
  public static void main(
                        String args[]){
    new AudioSynth01();
  }//end main
```

```
  //---------------------------------------//

  public AudioSynth01(){//constructor
    //A panel for the North position.  Note the
    // etched border.
    final JPanel controlButtonPanel =
                                      new JPanel();
    controlButtonPanel.setBorder(
             BorderFactory.createEtchedBorder());

    //A panel and button group for the radio
    // buttons in the Center position.
    final JPanel synButtonPanel = new JPanel();
    final ButtonGroup synButtonGroup =
                                new ButtonGroup();
    //This panel is used for cosmetic purposes
    // only, to cause the radio buttons to be
    // centered horizontally in the Center
    // position.
    final JPanel centerPanel = new JPanel();

    //A panel for the South position.  Note the
    // etched border.
    final JPanel outputButtonPanel =
                                      new JPanel();
    outputButtonPanel.setBorder(
             BorderFactory.createEtchedBorder());
    final ButtonGroup outputButtonGroup =
                                new ButtonGroup();

    //Disable the Play button initially to force
    // the user to generate some data before
    // trying to listen to it or write it to a
    // file.
    playOrFileBtn.setEnabled(false);

    //Register anonymous listeners on the
    // Generate button and the Play/File button.
    generateBtn.addActionListener(
      new ActionListener(){
        public void actionPerformed(
                                    ActionEvent e){
          //Don't allow Play during generation
          playOrFileBtn.setEnabled(false);
          //Generate synthetic data
          new SynGen().getSyntheticData(
                                       audioData);
          //Now it is OK for the user to listen
          // to or file the synthetic audio data.
          playOrFileBtn.setEnabled(true);
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

    playOrFileBtn.addActionListener(
      new ActionListener(){
```

```
        public void actionPerformed(
                                    ActionEvent e){
          //Play or file the data synthetic data
          playOrFileData();
        }//end actionPerformed
      }//end ActionListener
    );//end addActionListener()

    //Add two buttons and a text field to a
    // physical group in the North of the GUI.
    controlButtonPanel.add(generateBtn);
    controlButtonPanel.add(playOrFileBtn);
    controlButtonPanel.add(elapsedTimeMeter);

    //Add radio buttons to a mutually exclusive
    // group in the Center of the GUI.  Make
    // additions here if you add new synthetic
    // generator methods.
    synButtonGroup.add(tones);
    synButtonGroup.add(stereoPanning);
    synButtonGroup.add(stereoPingpong);
    synButtonGroup.add(fmSweep);
    synButtonGroup.add(decayPulse);
    synButtonGroup.add(echoPulse);
    synButtonGroup.add(waWaPulse);

    //Add radio buttons to a physical group and
    // center it in the Center of the GUI. Make
    // additions here if you add new synthetic
    // generator methods.
    synButtonPanel.setLayout(
                            new GridLayout(0,1));
    synButtonPanel.add(tones);
    synButtonPanel.add(stereoPanning);
    synButtonPanel.add(stereoPingpong);
    synButtonPanel.add(fmSweep);
    synButtonPanel.add(decayPulse);
    synButtonPanel.add(echoPulse);
    synButtonPanel.add(waWaPulse);

    //Note that the centerPanel has center
    // alignment by default.
    centerPanel.add(synButtonPanel);

    //Add radio buttons to a mutually exclusive
    // group in the South of the GUI.
    outputButtonGroup.add(listen);
    outputButtonGroup.add(file);

    //Add radio buttons to a physical group in
    // the South of the GUI.
    outputButtonPanel.add(listen);
    outputButtonPanel.add(file);
    outputButtonPanel.add(fileName);

    //Add the panels containing components to the
```

```
    // content pane of the GUI in the appropriate
    // positions.
    getContentPane().add(
          controlButtonPanel,BorderLayout.NORTH);
    getContentPane().add(centerPanel,
                         BorderLayout.CENTER);
    getContentPane().add(outputButtonPanel,
                          BorderLayout.SOUTH);

    //Finish the GUI.  If you add more radio
    // buttons in the center, you may need to
    // modify the call to setSize to increase
    // the vertical component of the GUI size.
    setTitle("Copyright 2003, R.G.Baldwin");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(250,275);
    setVisible(true);
  }//end constructor
  //--------------------------------------------//

  //This method plays or files the synthetic
  // audio data that has been generated and saved
  // in an array in memory.
  private void playOrFileData() {
    try{
      //Get an input stream on the byte array
      // containing the data
      InputStream byteArrayInputStream =
                        new ByteArrayInputStream(
                                      audioData);

      //Get the required audio format
      audioFormat = new AudioFormat(
                                 sampleRate,
                                 sampleSizeInBits,
                                 channels,
                                 signed,
                                 bigEndian);

      //Get an audio input stream from the
      // ByteArrayInputStream
      audioInputStream = new AudioInputStream(
                    byteArrayInputStream,
                    audioFormat,
                    audioData.length/audioFormat.
                                   getFrameSize());

      //Get info on the required data line
      DataLine.Info dataLineInfo =
                          new DataLine.Info(
                            SourceDataLine.class,
                                    audioFormat);

      //Get a SourceDataLine object
      sourceDataLine = (SourceDataLine)
                             AudioSystem.getLine(
```

```
                                      dataLineInfo);
      //Decide whether to play the synthetic
      // data immediately, or to write it into
      // an audio file, based on the user
      // selection of the radio buttons in the
      // South of the GUI..
      if(listen.isSelected()){
      //Create a thread to play back the data and
      // start it running.  It will run until all
      // the data has been played back
        new ListenThread().start();
      }else{
        //Disable buttons until existing data
        // is written to the file.
        generateBtn.setEnabled(false);
        playOrFileBtn.setEnabled(false);

        //Write the data to an output file with
        // the name provided by the text field
        // in the South of the GUI.
        try{
          AudioSystem.write(
                    audioInputStream,
                    AudioFileFormat.Type.AU,
                    new File(fileName.getText() +
                                          ".au"));
        }catch (Exception e) {
          e.printStackTrace();
          System.exit(0);
        }//end catch
        //Enable buttons for another operation
        generateBtn.setEnabled(true);
        playOrFileBtn.setEnabled(true);
      }//end else
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch
  }//end playOrFileData
//===============================================//

//Inner class to play back the data that was
// saved.
class ListenThread extends Thread{
  //This is a working buffer used to transfer
  // the data between the AudioInputStream and
  // the SourceDataLine.  The size is rather
  // arbitrary.
  byte playBuffer[] = new byte[16384];

  public void run(){
    try{
      //Disable buttons while data is being
      // played.
      generateBtn.setEnabled(false);
      playOrFileBtn.setEnabled(false);
```

```
      //Open and start the SourceDataLine
      sourceDataLine.open(audioFormat);
      sourceDataLine.start();

      int cnt;
      //Get beginning of elapsed time for
      // playback
      long startTime = new Date().getTime();

      //Transfer the audio data to the speakers
      while((cnt = audioInputStream.read(
                              playBuffer, 0,
                              playBuffer.length))
                                        != -1){
        //Keep looping until the input read
        // method returns -1 for empty stream.
        if(cnt > 0){
          //Write data to the internal buffer of
          // the data line where it will be
          // delivered to the speakers in real
          // time
          sourceDataLine.write(
                             playBuffer, 0, cnt);
        }//end if
      }//end while

      //Block and wait for internal buffer of the
      // SourceDataLine to become empty.
      sourceDataLine.drain();


      //Get and display the elapsed time for
      // the previous playback.
      int elapsedTime =
         (int)(new Date().getTime() - startTime);
      elapsedTimeMeter.setText("" + elapsedTime);

      //Finish with the SourceDataLine
      sourceDataLine.stop();
      sourceDataLine.close();

      //Re-enable buttons for another operation
      generateBtn.setEnabled(true);
      playOrFileBtn.setEnabled(true);
    }catch (Exception e) {
      e.printStackTrace();
      System.exit(0);
    }//end catch

  }//end run
}//end inner class ListenThread
//===========================================//

//Inner signal generator class.
```

```
//An object of this class can be used to
// generate a variety of different synthetic
// audio signals.  Each time the getSyntheticData
// method is called on an object of this class,
// the method will fill the incoming array with
// the samples for a synthetic signal.
class SynGen{
  //Note:  Because this class uses a ByteBuffer
  // asShortBuffer to handle the data, it can
  // only be used to generate signed 16-bit
  // data.
  ByteBuffer byteBuffer;
  ShortBuffer shortBuffer;
  int byteLength;

  void getSyntheticData(byte[] synDataBuffer){
    //Prepare the ByteBuffer and the shortBuffer
    // for use
    byteBuffer = ByteBuffer.wrap(synDataBuffer);
    shortBuffer = byteBuffer.asShortBuffer();

    byteLength = synDataBuffer.length;

    //Decide which synthetic data generator
    // method to invoke based on which radio
    // button the user selected in the Center of
    // the GUI.  If you add more methods for
    // other synthetic data types, you need to
    // add corresponding radio buttons to the
    // GUI and add statements here to test the
    // new radio buttons.  Make additions here
    // if you add new synthetic generator
    // methods.

    if(tones.isSelected()) tones();
    if(stereoPanning.isSelected())
                                  stereoPanning();
    if(stereoPingpong.isSelected())
                                 stereoPingpong();
    if(fmSweep.isSelected()) fmSweep();
    if(decayPulse.isSelected()) decayPulse();
    if(echoPulse.isSelected()) echoPulse();
    if(waWaPulse.isSelected()) waWaPulse();

  }//end getSyntheticData method
  //-----------------------------------------//

  //This method generates a monaural tone
  // consisting of the sum of three sinusoids.
  void tones(){
    channels = 1;//Java allows 1 or 2
    //Each channel requires two 8-bit bytes per
    // 16-bit sample.
    int bytesPerSamp = 2;
    sampleRate = 16000.0F;
    // Allowable 8000,11025,16000,22050,44100
```

```
    int sampLength = byteLength/bytesPerSamp;
    for(int cnt = 0; cnt < sampLength; cnt++){
      double time = cnt/sampleRate;
      double freq = 950.0;//arbitrary frequency
      double sinValue =
        (Math.sin(2*Math.PI*freq*time) +
        Math.sin(2*Math.PI*(freq/1.8)*time) +
        Math.sin(2*Math.PI*(freq/1.5)*time))/3.0;
      shortBuffer.put((short)(16000*sinValue));
    }//end for loop
  }//end method tones
  //-----------------------------------------//

  //This method generates a stereo speaker sweep,
  // starting with a relatively high frequency
  // tone on the left speaker and moving across
  // to a lower frequency tone on the right
  // speaker.
  void stereoPanning(){
    channels = 2;//Java allows 1 or 2
    int bytesPerSamp = 4;//Based on channels
    sampleRate = 16000.0F;
    // Allowable 8000,11025,16000,22050,44100
    int sampLength = byteLength/bytesPerSamp;
    for(int cnt = 0; cnt < sampLength; cnt++){
      //Calculate time-varying gain for each
      // speaker
      double rightGain = 16000.0*cnt/sampLength;
      double leftGain = 16000.0 - rightGain;

      double time = cnt/sampleRate;
      double freq = 600;//An arbitrary frequency
      //Generate data for left speaker
      double sinValue =
                 Math.sin(2*Math.PI*(freq)*time);
      shortBuffer.put(
                      (short)(leftGain*sinValue));
      //Generate data for right speaker
      sinValue =
             Math.sin(2*Math.PI*(freq*0.8)*time);
      shortBuffer.put(
                    (short)(rightGain*sinValue));
    }//end for loop
  }//end method stereoPanning
  //-----------------------------------------//

  //This method uses stereo to switch a sound
  // back and forth between the left and right
  // speakers at a rate of about eight switches
  // per second.  On my system, this is a much
  // better demonstration of the sound separation
  // between the two speakers than is the
  // demonstration produced by the stereoPanning
  // method.  Note also that because the sounds
  // are at different frequencies, the sound
  // produced is similar to that of U.S.
```

```
// emergency vehicles.

void stereoPingpong(){
  channels = 2;//Java allows 1 or 2
  int bytesPerSamp = 4;//Based on channels
  sampleRate = 16000.0F;
  // Allowable 8000,11025,16000,22050,44100
  int sampLength = byteLength/bytesPerSamp;
  double leftGain = 0.0;
  double rightGain = 16000.0;
  for(int cnt = 0; cnt < sampLength; cnt++){
    //Calculate time-varying gain for each
    // speaker
    if(cnt % (sampLength/8) == 0){
      //swap gain values
      double temp = leftGain;
      leftGain = rightGain;
      rightGain = temp;
    }//end if

    double time = cnt/sampleRate;
    double freq = 600;//An arbitrary frequency
    //Generate data for left speaker
    double sinValue =
                Math.sin(2*Math.PI*(freq)*time);
    shortBuffer.put(
                    (short)(leftGain*sinValue));
    //Generate data for right speaker
    sinValue =
           Math.sin(2*Math.PI*(freq*0.8)*time);
    shortBuffer.put(
                   (short)(rightGain*sinValue));
  }//end for loop
}//end stereoPingpong method
//-----------------------------------------//

//This method generates a monaural linear
// frequency sweep from 100 Hz to 1000Hz.
void fmSweep(){
  channels = 1;//Java allows 1 or 2
  int bytesPerSamp = 2;//Based on channels
  sampleRate = 16000.0F;
  // Allowable 8000,11025,16000,22050,44100
  int sampLength = byteLength/bytesPerSamp;
  double lowFreq = 100.0;
  double highFreq = 1000.0;

  for(int cnt = 0; cnt < sampLength; cnt++){
    double time = cnt/sampleRate;

    double freq = lowFreq +
             cnt*(highFreq-lowFreq)/sampLength;
    double sinValue =
                 Math.sin(2*Math.PI*freq*time);
    shortBuffer.put((short)(16000*sinValue));
  }//end for loop
```

```
  }//end method fmSweep
  //-----------------------------------------//

  //This method generates a monaural triple-
  // frequency pulse that decays in a linear
  // fashion with time.
  void decayPulse(){
    channels = 1;//Java allows 1 or 2
    int bytesPerSamp = 2;//Based on channels
    sampleRate = 16000.0F;
    // Allowable 8000,11025,16000,22050,44100
    int sampLength = byteLength/bytesPerSamp;
    for(int cnt = 0; cnt < sampLength; cnt++){
      //The value of scale controls the rate of
      // decay - large scale, fast decay.
      double scale = 2*cnt;
      if(scale > sampLength) scale = sampLength;
      double gain =
             16000*(sampLength-scale)/sampLength;
      double time = cnt/sampleRate;
      double freq = 499.0;//an arbitrary freq
      double sinValue =
        (Math.sin(2*Math.PI*freq*time) +
        Math.sin(2*Math.PI*(freq/1.8)*time) +
        Math.sin(2*Math.PI*(freq/1.5)*time))/3.0;
      shortBuffer.put((short)(gain*sinValue));
    }//end for loop
  }//end method decayPulse
  //-----------------------------------------//

  //This method generates a monaural triple-
  // frequency pulse that decays in a linear
  // fashion with time.  However, three echoes
  // can be heard over time with the amplitude
  // of the echoes also decreasing with time.
  void echoPulse(){
    channels = 1;//Java allows 1 or 2
    int bytesPerSamp = 2;//Based on channels
    sampleRate = 16000.0F;
    // Allowable 8000,11025,16000,22050,44100
    int sampLength = byteLength/bytesPerSamp;
    int cnt2 = -8000;
    int cnt3 = -16000;
    int cnt4 = -24000;
    for(int cnt1 = 0; cnt1 < sampLength;
                    cnt1++,cnt2++,cnt3++,cnt4++){
      double val = echoPulseHelper(
                                cnt1,sampLength);
      if(cnt2 > 0){
        val += 0.7 * echoPulseHelper(
                                cnt2,sampLength);
      }//end if
      if(cnt3 > 0){
        val += 0.49 * echoPulseHelper(
                                cnt3,sampLength);
      }//end if
```

```
      if(cnt4 > 0){
        val += 0.34 * echoPulseHelper(
                                cnt4,sampLength);
      }//end if

      shortBuffer.put((short)val);
    }//end for loop
  }//end method echoPulse
  //-----------------------------------------//

  double echoPulseHelper(int cnt,int sampLength){
    //The value of scale controls the rate of
    // decay - large scale, fast decay.
    double scale = 2*cnt;
    if(scale > sampLength) scale = sampLength;
    double gain =
             16000*(sampLength-scale)/sampLength;
    double time = cnt/sampleRate;
    double freq = 499.0;//an arbitrary freq
    double sinValue =
      (Math.sin(2*Math.PI*freq*time) +
      Math.sin(2*Math.PI*(freq/1.8)*time) +
      Math.sin(2*Math.PI*(freq/1.5)*time))/3.0;
    return(short)(gain*sinValue);
  }//end echoPulseHelper

  //-----------------------------------------//

  //This method generates a monaural triple-
  // frequency pulse that decays in a linear
  // fashion with time.  However, three echoes
  // can be heard over time with the amplitude
  // of the echoes also decreasing with time.
  //Note that this method is identical to the
  // method named echoPulse, except that the
  // algebraic sign was switched on the amplitude
  // of two of the echoes before adding them to
  // the composite synthetic signal.  This
  // resulted in a difference in the
  // sound.
  void waWaPulse(){
    channels = 1;//Java allows 1 or 2
    int bytesPerSamp = 2;//Based on channels
    sampleRate = 16000.0F;
    // Allowable 8000,11025,16000,22050,44100
    int sampLength = byteLength/bytesPerSamp;
    int cnt2 = -8000;
    int cnt3 = -16000;
    int cnt4 = -24000;
    for(int cnt1 = 0; cnt1 < sampLength;
                    cnt1++,cnt2++,cnt3++,cnt4++){
      double val = waWaPulseHelper(
                                cnt1,sampLength);
      if(cnt2 > 0){
        val += -0.7 * waWaPulseHelper(
                                cnt2,sampLength);
```

```
      }//end if
      if(cnt3 > 0){
        val += 0.49 * waWaPulseHelper(
                                cnt3,sampLength);
      }//end if
      if(cnt4 > 0){
        val += -0.34 * waWaPulseHelper(
                                cnt4,sampLength);
      }//end if

      shortBuffer.put((short)val);
    }//end for loop
  }//end method waWaPulse
  //---------------------------------------//

  double waWaPulseHelper(int cnt,int sampLength){
    //The value of scale controls the rate of
    // decay - large scale, fast decay.
      double scale = 2*cnt;
      if(scale > sampLength) scale = sampLength;
      double gain =
             16000*(sampLength-scale)/sampLength;
    double time = cnt/sampleRate;
    double freq = 499.0;//an arbitrary freq
    double sinValue =
      (Math.sin(2*Math.PI*freq*time) +
      Math.sin(2*Math.PI*(freq/1.8)*time) +
      Math.sin(2*Math.PI*(freq/1.5)*time))/3.0;
    return(short)(gain*sinValue);
  }//end waWaPulseHelper

  //---------------------------------------//
}//end SynGen class
//=============================================//

}//end outer class AudioSynth01.java
```

Listing 49

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-