Java Wav File IO

This page describes an easy to use Java class for handling the reading and writing of Wav files.

Download Files
WavFile.tar.gz

Overview

The `WavFile` class takes care of all the file IO for reading and writing audio data to and from wave files. The `WavFile` class provides methods for accessing wav file data using different java primitive types, these are `long`, `int` and `double`. It is up to the user to choose the correct type for the given wav file's sample resolution.

The standard wav file resolutions are shown in the table below. For all resolutions, up to 32 bit unsigned, handling samples using the `int` type is fine. For greater resolutions, the `long` type should be used. The `WavFile` class assumes all samples of resolution 8 bits and less are unsigned, all resolutions over 8 bits are signed (this is the seems to be the standard for wav files).

The `double` type can be used for any wav file data resolution. The `WavFile` class will automatically scale sample values to and from the -1.0 to 1.0 floating point range.

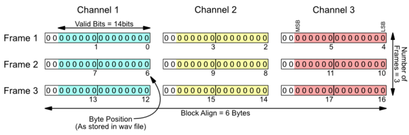
Resolution	Min Value		Max Value		Number of Bytes
8 bit Unsigned	0	(0x00)	255	(0xFF)	1
16 bit Signed	-32,768	(0x7FFF)	32,767	(0x8000)	2
24 bit Signed	-34,359,738,367	(0x7FFFFF)	34,359,738,368	(0x800000)	3

Floating Point Value Scaling

Due to the scaling process, reading wav files as doubles then writing them back out can result in changes in some sample values. This process is illustrated in the image below. Signed integers encoded using two's complement have different maximum magnitudes for positive and negative numbers. For example a signed 16 bit value has a range of -32,768 to 32,767. When scaling from integer to double, the maximum negative magnitude must be used to ensure that no sample has a value less than -1.0. When scaling from a double to an integer, the positive magnitude is used to ensure the integer range is not exceeded.

Wav File Parameters and Format

The image below illustrates the structure of the section storing audio data found within a wav file. In this case, there are 3 audio channels. The resolution is 14 bits so two bytes are required to store each sample. Therefore, each frame requires 6 bytes, this is termed the 'Block Align'.



The following links provide detailed information on the wav file format.

- http://www.sonicspot.com/guide/wavefiles.html
- http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html
- http://www.blitter.com/~russtopia/MIDI/~jglatt/tech/wave.htm

Reading Wav Files

The code below is an example of how to read a wav file. The The data is read in blocks of 100 frames, then each sample is checked to find the maximum and minimum sample value.

File: `ReadExample.java`

```
import java.io.*;

public class ReadExample
{
   public static void main(String[] args)
   {
      try
      {
         // Open the wav file specified as the first argument
         WavFile wavFile = WavFile.openWavFile(new File(args[0]));

         // Display information about the wav file
         wavFile.display();

         // Get the number of audio channels in the wav file
         int numChannels = wavFile.getNumChannels();

         // Create a buffer of 100 frames
         double[] buffer = new double[100 * numChannels];

         int framesRead;
         double min = Double.MAX_VALUE;
         double max = Double.MIN_VALUE;

         do
         {
            // Read frames into buffer
            framesRead = wavFile.readFrames(buffer, 100);

            // Loop through frames and look for minimum and maximum value
            for (int s=0 ; s<framesRead * numChannels ; s++)
            {
               if (buffer[s] > max) max = buffer[s];
               if (buffer[s] < min) min = buffer[s];
            }
         }
         while (framesRead != 0);

         // Close the wavFile
         wavFile.close();

         // Output the minimum and maximum value
         System.out.printf("Min: %f, Max: %f\n", min, max);
      }
      catch (Exception e)
      {
         System.err.println(e);
      }
   }
}
```

Writing Wav Files

The code below is an example of writing a wav file. The code creates a two channel wav file of 5 seconds duration. Each channel stores a single sinusoidal tone, one at 400Hz the other at at 500Hz.

File: `WriteExample.java`

```
import java.io.*;

public class WriteExample
{
   public static void main(String[] args)
   {
      try
      {
         int sampleRate = 44100;    // Samples per second
         double duration = 5.0;     // Seconds

         // Calculate the number of frames required for specified duration
         long numFrames = (long)(duration * sampleRate);

         // Create a wav file with the name specified as the first argument
         WavFile wavFile = WavFile.newWavFile(new File(args[0]), 2, numFrames, 16, sampleRate);

         // Create a buffer of 100 frames
         double[][] buffer = new double[2][100];

         // Initialise a local frame counter
         long frameCounter = 0;

         // Loop until all frames written
         while (frameCounter < numFrames)
         {
            // Determine how many frames to write, up to a maximum of the buffer size
            long remaining = wavFile.getFramesRemaining();
            int toWrite = (remaining > 100) ? 100 : (int) remaining;

            // Fill the buffer, one tone per channel
            for (int s=0 ; s<toWrite ; s++, frameCounter++)
            {
               buffer[0][s] = Math.sin(2.0 * Math.PI * 400 * frameCounter / sampleRate);
               buffer[1][s] = Math.sin(2.0 * Math.PI * 500 * frameCounter / sampleRate);
            }

            // Write the buffer
            wavFile.writeFrames(buffer, toWrite);
         }

         // Close the wavFile
         wavFile.close();
      }
      catch (Exception e)
      {
         System.err.println(e);
      }
   }
}
```

WavFile Methods

The tables below document the public methods of the `WavFile` class.

General Methods

`static WavFile`	`newWavFile(File file, int numChannels, long numFrames, int validBits, long sampleRate) throws IOException, WavFileException` This method creates a new wav file for writing to. It writes header information to the wav file. Once this method has been called and a `WavFile` instance retrieved, the `writeFrames` methods can be used to write samples to the wav file.
`static WavFile`	`openWavFile(File file) throws IOException, WavFileException` This method opens a wav file ready for reading. It retrieves wav details from the file header. Once this method has been called and a `WavFile` instance retrieved, the `readFrames` methods can be used for accessing the samples.
`void`	`close() throws IOException` This method closes the open file handlers. When reading a wav file, it can be called at any point, but once closed no more samples can be read. When writing, this **must** be called once all samples have been written using the `writeFrames` methods. If not called, the wav file may be incomplete.
`int`	`getNumChannels()` Returns the number of channels.
`long`	`getNumFrames()` Returns the total number of frames stored in the wav file.
`long`	`getFramesRemaining()` Returns the remaining number of frames available for reading or writing.
`long`	`getSampleRate()` Returns the sample rate.
`int`	`getValidBits()` Returns the number of valid bits used for storing a single sample. This is the sample resolution.
`void`	`display()` Prints parameters for this wav file to `System.out`.
`void`	`display(PrintStream out)` Prints parameters for this wav file to the specified `PrintStream`

Read and Write Using The `int` Type

The `int` type should only be used when dealing with wav files using 32 bit signed values or 31 bit or less signed or unsigned values.

`int`	`readFrames(int[] sampleBuffer, int numFramesToRead) throws IOException, WavFileException` Read `numFramesToRead` frames from the wav file and place into `sampleBuffer` starting from index 0. The user must make sure `sampleBuffer` contains enough space for all frames to be read (required length is the number of channels multiplied by number of frames to be read). Samples from each channel are interlaced into the buffer. This method returns the number of frames read. When no frames are left to read this method returns 0.
`int`	`readFrames(int[] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException` Same as above, but `sampleBuffer` is filled starting at index `offset`.
`int`	`readFrames(int[][] sampleBuffer, int numFramesToRead) throws IOException, WavFileException` Read `numFramesToRead` frames from the wav file and place into `sampleBuffer` starting from indices 0,0. The `sampleBuffer` indices are 'Channel Number', 'Frame Number'. For example, `sampleBuffer[2][23]` is the channel 3 sample from frame 24. The user must make sure `sampleBuffer` is the correct dimensions for all channels and frames to be read. This method returns the number of frames read. When no frames are left to read this method returns 0.
`int`	`readFrames(int[][] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException` Same as above, but `sampleBuffer` is filled starting at frame index `offset`.
`int`	`writeFrames(int[] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException` Write `numFramesToWrite` frames from `sampleBuffer` into the wav file starting from index 0. The user must make sure `sampleBuffer` is large enough for all frames to be retrieved for writing (required length is the number of channels multiplied by number of frames to be written). Samples from each channel are retrieved in an interlaced order. This method returns the number of frames written. The maximum number of frames that can be written is specified when the `WavFile` is created using `newWavFile`. When all frames have been written this method returns 0.
`int`	`writeFrames(int[] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException` Same as above, but `sampleBuffer` is read starting at index `offset`.
`int`	`writeFrames(int[][] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException` Write `numFramesToRead` frames from `sampleBuffer` into the wav file starting from indices 0,0. The `sampleBuffer` indices are 'Channel Number', 'Frame Number'. For example, `sampleBuffer[2][23]` is the channel 3 sample from frame 24. The user must make sure `sampleBuffer` is the correct dimensions for all channels and frames to be retrieved for writing. This method returns the number of frames written. The maximum number of frames that can be written is specified when the `WavFile` is created using `newWavFile`. When all frames have been written this method returns 0.
`int`	`writeFrames(int[][] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException` Same as above, but `sampleBuffer` is read starting at frame index `offset`.

Read and Write Using The `long` Type

The `long` type should be used when dealing with wav files using 64 bit signed values or 63 bit or less signed or unsigned values.

`int`	`readFrames(long[] sampleBuffer, int numFramesToRead) throws IOException, WavFileException` Same as `int` version, but supports samples up to 64 bits unsigned.
`int`	`readFrames(long[] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException` Same as `int` version, but supports samples up to 64 bits unsigned.
`int`	`readFrames(long[][] sampleBuffer, int numFramesToRead) throws IOException, WavFileException` Same as `int` version, but supports samples up to 64 bits unsigned.
`int`	`readFrames(long[][] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException` Same as `int` version, but supports samples up to 64 bits unsigned.
`int`	`writeFrames(long[] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException` Same as `int` version, but supports samples up to 64 bits unsigned.
`int`	`writeFrames(long[] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException` Same as `int` version, but supports samples up to 64 bits unsigned.
`int`	`writeFrames(long[][] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException` Same as `int` version, but supports samples up to 64 bits unsigned.
`int`	`writeFrames(long[][] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException` Same as `int` version, but supports samples up to 64 bits unsigned.

Read and Write Using The `double` Type

`int`	`readFrames(double[] sampleBuffer, int numFramesToRead) throws IOException, WavFileException` Same as `int` version, but samples are scaled to range -1.0 to 1.0. See the note on data scaling.
`int`	`readFrames(double[] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException` Same as `int` version, but samples are scaled to range -1.0 to 1.0. See the note on data scaling.
`int`	`readFrames(double[][] sampleBuffer, int numFramesToRead) throws IOException, WavFileException` Same as `int` version, but samples are scaled to range -1.0 to 1.0. See the note on data scaling.
`int`	`readFrames(double[][] sampleBuffer, int offset, int numFramesToRead) throws IOException, WavFileException` Same as `int` version, but samples are scaled to range -1.0 to 1.0. See the note on data scaling.
`int`	`writeFrames(double[] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException` Same as `int` version, but samples are scaled from range -1.0 to 1.0. See the note on data scaling.
`int`	`writeFrames(double[] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException` Same as `int` version, but samples are scaled from range -1.0 to 1.0. See the note on data scaling.
`int`	`writeFrames(double[][] sampleBuffer, int numFramesToWrite) throws IOException, WavFileException` Same as `int` version, but samples are scaled from range -1.0 to 1.0. See the note on data scaling.
`int`	`writeFrames(double[][] sampleBuffer, int offset, int numFramesToWrite) throws IOException, WavFileException` Same as `int` version, but samples are scaled from range -1.0 to 1.0. See the note on data scaling.