Practical 4 – Sound

This practical looks at the facilities available in Java to manipulate sound. We shall start by playing a simple sound sample. We will then look at splitting up the relevant sound components and applying a filter. Finally, we will look at writing our own sound filters.

Before we start, download the file, SoundCode.zip, from the Canvas Practicals page, to a suitable folder on your H drive. You will need to create two IDE projects: *SimpleSound* and *SoundFilters* with the relevant files found in each of the respective directories in the zip file. You will also need to copy the wav files in the *Samples* sub directory to a similarly named directory that is at the same directory level as your above two projects.

1. Our first step will be to look at simply playing a WAV sound file. The Java code required to do this is contained within the file SoundPlay.java within SimpleSound. Set SoundPlay.java to be the main file then put on your headphones and compile and run your project. This will play the WAV file 'groovey.wav' which is located within the 'Samples' subdirectory of the main Sounds directory. We are going to keep all our original sound files in this directory since they can get quite large and we do not want to have many identical copies of the same file.

2. Within the 'Samples' directory there are a few other sound samples. Try changing the sound file played by the SoundPlay code to one of these other files (make sure you use the headphones before you do this).

3. The *SoundPlay* code plays the sound without using threads. Compare this code to your assignment code for playing a sound and note how the code shown in *SoundPlay* has been converted to work as a thread.

We are now going to look at applying sound filters to our sound samples – this will require us to add an additional class which implements the particular filter that we wish to apply.

1. The files in *SoundFilters* contains the relevant elements that you will need to begin using and creating sound filters. Using the *SoundFilters* project you created previously, set the PlayFilter.java file to be your main file and compile the project. With your headphones still plugged in, run this program.

   By default this application will fade out a sound sample that is passed to it so you should hear the sound sample gradually get quieter. Try changing the sound sample that is used in the PlayFilter.java file and see how it works for different sounds in the Samples directory.

   The class that is doing all the work is called FadeFilterStream.java. Open this file and look for the `read` method. This method receives a data stream that it can then apply a filter to. For our WAV files, this data stream is in a 2 bytes per sample, little endian format.

   In order to grab 2 bytes at a time and reverse their order to make them big endian, we can use the `getSample` method provided in the FadeFilterStream class (later on we can also use the `setSample` method to insert values back into the data stream in little endian format). The useful thing about the `getSample` and `setSample` methods is that they provide the amplitude values as a short integer (16 bit instead of the normal 32 bit integer that you are familiar with). This means we can manipulate the amplitude as an integer number. If we make it bigger, the sound will be louder, if we make it smaller, the sound will be quieter. You will have to be careful not to make it too loud or the short value will overflow and give you a low number instead of a high one (you will hear a lot of crackling if you have done this).

Practical 4 – Sound

The read method of FadeFilterStream steadily reduces the amplitude of the sound sample as we progress through it. It does this by multiplying the sound amplitude we receive by a number that starts off at 1 (i.e. make no change) and decreases towards 0 (i.e. make everything silent). The read method then overwrites the old sample value at our current position `p` with our new value at reduced volume.

2. Try adjusting the FadeFilterStream class to make the volume of the sample become 1.1 times the original sound volume instead of a steadily decreasing volume.

3. Now try altering the FadeFilterStream class to make the sound sample start off being silent and then get louder (in other words operate in reverse to the original fade filter).

4. We are now going to create a filter that reverses the current sample such that it is played backwards. Effectively what we want to do here is reverse the `sample` data stream array in the `read` method such that what we hear at the beginning is the last value in the original array and what we hear at the end is the value that was originally the first value in the array.

   An easy (but not very efficient) way to do this is make a copy of the entire sound array and then overwrite the original array with a reversed version of the copy. An alternative method is to use a couple of buffer variables to read values from each end, reverse them around and put them back in the original array. We must remember to stop reversing once we get to the middle of the array otherwise we will just reverse our previous reversal and end up back where we started.

   The file ReverseFilterStream.java contains some starting elements of code in the read method that will enable you to try the first approach. Try to complete this code and then replace the references to the FadeFilterStream class in PlayFilter.java with ReverseFilterStream. Make sure ReverseFilterStream.java is part of your project, then compile your program and listen to check if you have succeeded in reversing the sound signal. If you succeed, try out your filter on the different sound samples in the Samples directory.

5. For a bigger challenge, we are now going to look at creating an echo filter. In order to create an echo, we need to simulate the sound we are currently playing bouncing back to us a little later on with a lower volume. We can implement this with a filter by taking the sound we are processing now and adding it to the sound we will hear a little later but with a reduced volume for the current sound. Since our data array `sample` contains the entire sample, we can access the sound that will be heard in the future by getting and setting values further down the array. This allows us to read the volume now and the volume a bit further down the sample, then add a quieter version of the current sound to the volume we read from further down the array and set this value back into the array at the later time.

   The file EchoFilterStream.java contains some initial code in the read method to point you in the direction of a solution but you will need to implement some of the code yourself. Make sure this file is part of your project, then replace the references to ReverseFilterStream in your PlayFilter.java file with the word EchoFilterStream from above with ReverseFilterStream. When you have made an attempt at solving this problem by editing the EchoFilterStream.java file, compile the project and check to see if you can hear an echo. Don't worry if this seems a little difficult at first, the solution is not as hard as you might think. When you get it working, try your new EchoFilterStream program on the different sound samples in the Samples directory.