Odd Semester (2023)

BINUS UNIVERSITY
BINUS INTERNATIONAL

Assignment Cover Letter

(Individual Work)

Student Information:	Surname	Given Names	Student ID Number
	Reynaldi	Raphael	2440071973

Course Code	: COMP6699	Course Name	: Object Orientated Programming
Class	: L2BC	Name of Lecturer(s)	: Jude Joseph Lamug Martinez

Major:

Computer Science

Title of Assignment (if any) : Jot Down Audio

Type of Assignment **: Final Project**

Submission Pattern

Due Date **: 22-06-2021**

Submission Date **: 22-06-2020**

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

Binus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept and consent to Binus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

(Name of Student)

Raphael Reynaldi

Table of Contents

Project Specification

When trying to come up with an idea for the project, I first thought about the common problems that I and many others face in our daily lives. I am a person who enjoys music to a huge extent from listening to different works to playing instruments for self expression. One of the major problems that I face whenever I pick up my guitar and start playing is that oftentimes I find myself with new musical ideas that I want to be able to capture and use in my projects later on. Usually, I would use my phone to record the idea that I wanted to keep. However, I need to make compromises to make it work. The program aims to reduce the amount of compromises through specific features. Users are able to take the audio input of the user and output it directly to their audio output of preference. This will allow users to check before recording in case there is a problem with their audio input. The program also is able to record audio input while monitoring the input and there is also an option where they could just record without monitoring their input.

Input into the program:

- Audio input line
- Audio output line
- Audio samples gathered from audio input
- User defined WAV file name

Output of the program:

- Live audio playback from user
- WAV file with user defined file name

Solution Design

When the user starts up the program, they will be greeted with a main menu in which they can use to navigate the features of the program. The user will need to input an integer to navigate the menu. The main menu is presented as follows:

|| Welcome to Jot Down Audio ||
||Audio Monitoring and Recording Program||
Enter an integer to navigate
1. Stream audio input
2. Stop audio stream
3. Record while monitoring audio
4. Record without monitoring audio
5. Stop recording audio
6. Exit program

Stream Audio Input

When the user selects this option the program will detect the user's input and output and initiate the two sources together to create an infinite stream where the user is able to monitor their input and hear it through their preferred output in real time. The user will not be able to initiate the second option which is to stop the audio stream without starting this one. Furthermore, the user will also be unable to select this option while the stream is already active.

Stop Audio Input

Selecting this option will allow the user to close the input/output stream and end the monitoring session. It will clear out all of the audio samples that are processed by the user's preferred audio input and then proceed to close the audio input and output of the user. The user will not be able to select this option without selecting the stream audio input option too. Furthermore, they will not be able to select the stop audio input option unless there is an audio stream that is currently active.

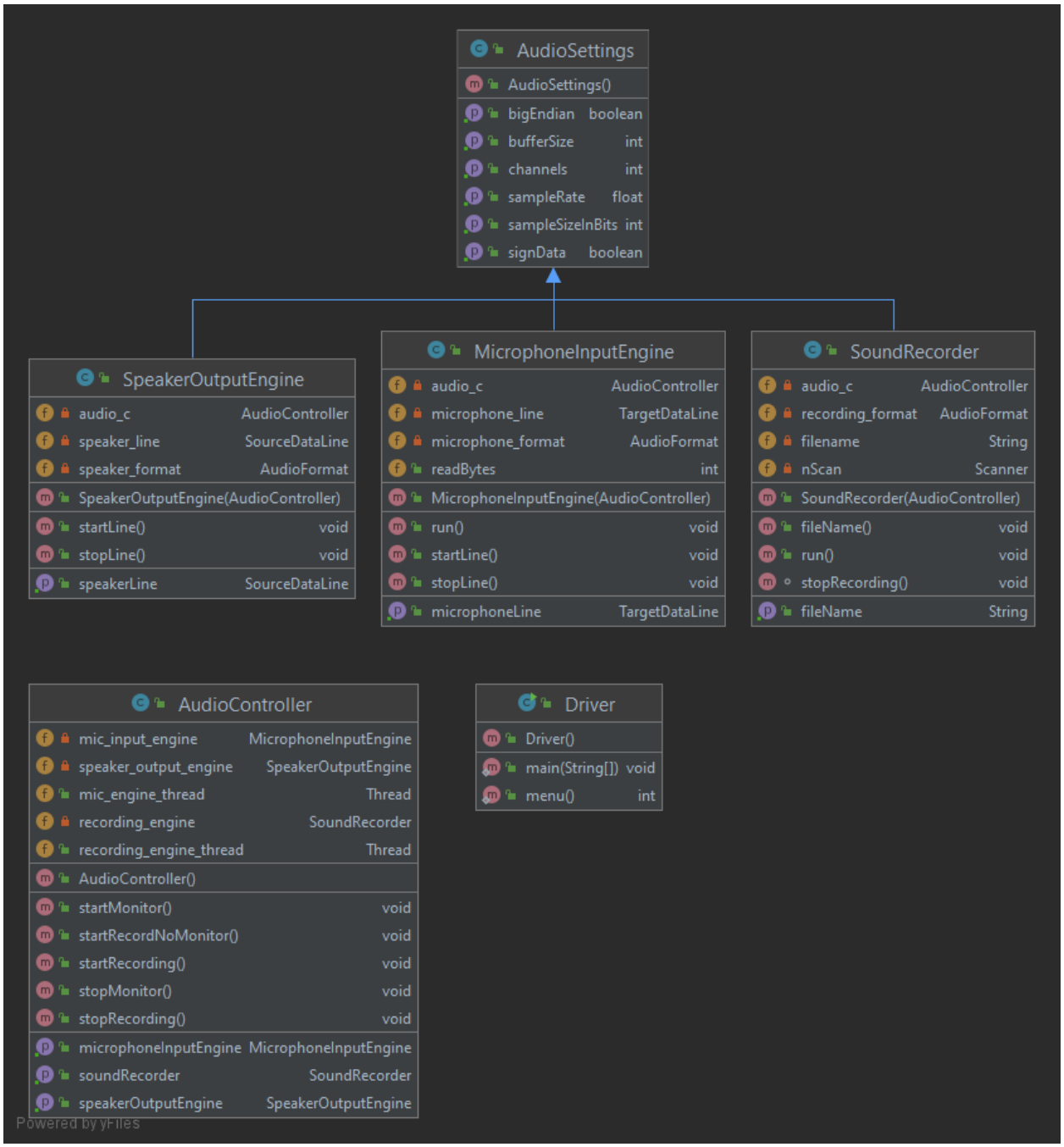
Record While Monitoring Audio

This option allows the user to record their audio while monitoring their audio input. The user is able to specify the name of their file and once they select the stop recording audio option, it will produce a WAV file with the file name they typed in. The program does this by taking the user's audio samples and writing them into a WAV file. This option is geared more towards users with instruments that require amplification such as an electric guitar. Unlike other instruments, for an electric guitar to be heard, the user must be able to have a device that amplifies their signal that is being produced by the pickups. If the user doesn't have an amplifier at their disposal, this option will be helpful for them.

Record Without Monitoring Audio
Similarly to the option of recording audio while monitoring their input, everything is the same. However this time, the user is unable to hear themselves through their preferred output. This option is geared towards instruments that don't require a source of amplification such as acoustic guitars or vocals. It can get distracting for some users to hear themselves twice and thus this option is provided to the users that would prefer to not be able to monitor their audio.

Stop Recording Audio
This option is tied to the user recording their audio. This option is used to stop receiving audio samples from the user’s audio input and write them into a WAV file.

UML Diagram


AudioSettings
AudioSettings()
bigEndian boolean
bufferSize int
channels int
sampleRate float
sampleSizeInBits int
signData boolean
SpeakerOutputEngine
audio_c AudioController
speaker_line SourceDataLine
speaker_format AudioFormat
SpeakerOutputEngine(AudioController)
startLine() void
stopLine() void
speakerLine SourceDataLine
MicrophoneInputEngine
audio_c AudioController
microphone_line TargetDataLine
microphone_format AudioFormat
readBytes int
MicrophoneInputEngine(AudioController)
run() void
startLine() void
stopLine() void
microphoneLine TargetDataLine
SoundRecorder
audio_c AudioController
recording_format AudioFormat
filename String
nScan Scanner
SoundRecorder(AudioController)
fileName() void
run() void
stopRecording() void
fileName String
AudioController
mic_input_engine MicrophoneInputEngine
speaker_output_engine SpeakerOutputEngine
mic_engine_thread Thread
recording_engine SoundRecorder
recording_engine_thread Thread
AudioController()
startMonitor() void
startRecordNoMonitor() void
startRecording() void
stopMonitor() void
stopRecording() void
microphoneInputEngine MicrophoneInputEngine
soundRecorder SoundRecorder
speakerOutputEngine SpeakerOutputEngine
Driver
Driver()
main(String[]) void
menu() int
Powered by yFiles

Code Explanation

AudioSettings Class

```java
public class AudioSettings {

    // Default values for a common WAV file and required values for SourceDataLine and
    // TargetDataLine parameters.

    // The amount of samples per second measured in khz
    public float sampleRate = 44100.0f;
    // The amount bits of data per audio sample
    public int SampleSizeInBits = 16;
    // The amount of audio channels
    public int channels = 1;
    // Determines how the data will be stored (signed means that the data can be expressed in negatives and positives
    // whereas if it isn't signed then the values will be stored from 0 and above)
    public boolean signData = true;
    // Determines if the audio samples will be stored in big endian or small endian
    public boolean bigEndian = false;
    // The amount of time it takes for the computer to process incoming audio
    // Set to the default value of buffer size
    public int bufferSize = 4096;
```

This class initializes the audio format that will be used in the SourceDataLine and TargetDataLine later on. These values represent the standard values for CD quality which means that the quality of audio that is seen on album discs are based on these values. Sample rate determines the amount of samples per second that is captured. The higher the value, the higher the quality. Sample size in bits is the amount of bits of data that is being stored per audio sample. Similarly, the higher the value the more data is being stored which means it will have an increase in quality. Channels is the amount of channels that the mixer will have. The value is set to 1 which means that it will be a mono channel. This allows a uniform amount of audio to be recorded and the user can alter it to pan left or right if they choose to do so. signData signifies if the data is signed or not. For the context of this project, there is no difference between the two. However, signed data means that negative and positive values represent vectors and zero represents no value while if the data is not signed then the values being expressed are represented as ones and zeros. Similarly, endianness is also a way for the program to store data and in the context of this project, it doesn't matter too much. Big endian is storing the most significant byte at the smallest memory and the least significant byte at the largest memory. Small endian is vice versa. BufferSize is the amount of time it takes for the computer to process incoming audio.

```
    public float getSampleRate() {
        return sampleRate;
    }

    public int getSampleSizeInBits() {
        return SampleSizeInBits;
    }

    public int getChannels() {
        return channels;
    }

    public boolean isSignData() {
        return signData;
    }

    public boolean isBigEndian() {
        return bigEndian;
    }

    public int getBufferSize() {
        return bufferSize;
    }

}
```

These are the getters for each of the variables mentioned before which will be extended to the MicrophoneInputEngine class and SpeakerOutputEngine class.

MicrophoneInputEngine

```java
public class MicrophoneInputEngine extends AudioSettings implements Runnable {

    private AudioController audio_c;
    private TargetDataLine microphoneLine;
    private AudioFormat microphoneFormat;

    public int readBytes;

    // MicrophoneInputEngine constructor which allows access of this class and others through one java class
    public MicrophoneInputEngine(AudioController audio_c) {
        this.audio_c = audio_c;

        // Try/ Catch statement to open up the user's TargetDataLine
        try {
            // Specified audio format from the AudioSettings class
            microphoneFormat = new AudioFormat(getSampleRate(), getSampleSizeInBits(), getChannels(), isSignData(), isBigEndian());

            // Receives the format and appends to the TargetDataLine
            microphoneLine = AudioSystem.getTargetDataLine(microphoneFormat);

            // Opens the TargetDataLine based on the format and the buffer size specified in AudioSettings class
            microphoneLine.open(microphoneFormat, getBufferSize());
            System.out.println("Microphone Line Opened.");

        } catch (LineUnavailableException e) {
            System.out.println("Microphone Line Unavailable!");
        }
    }
```

I created an audio controller object in this class so that I will be able to access the OutputSpeakerEngine class when there is an input stream ongoing. The TargetDataLine is a type of dataline which can be used to read audio data. This is what will be used in order to capture the audio samples which will then be read by this class. The AudioFormat class is used to be able to append the parameters set in the AudioSettings class into the AudioFormat class which is renamed to MicrophoneFormat. My constructor consists of the AudioController class as the parameters and setting it. Then I included a try and catch statement to open the user's target line by first creating a new audio format. Then, I append the format into the TargetDataLine and call the variable microphoneLine. After that, I opened the microphoneLine by giving the required parameters. I specified the format and the buffer size to allow the TargetDataLine to be able to be opened. The catch statement is there to signal the user if their microphone isn't able to get picked up by the program.

```java
// A Thread that functions to continuously receive audio samples from user
@Override
public void run() {

    // Creates a new Byte array with the buffer size specified from AudioSettings class
    // Byte array is used because it is the required parameter for a SourceDataLine to write audio into the mixer
    byte[] data = new byte[getBufferSize()];

    // A while loop is used to be able to read the audio input buffer and then write it to the user's mixer
    while(true) {
        // Read bytes from the microphone buffer
        readBytes = microphoneLine.read(data,  off: 0, data.length);
        audio_c.getSpeakerOutputEngine().getSpeakerLine().write(data,  off: 0, readBytes);

    }
}
```

After that, I have a method that will continuously run when the class is inside a thread. This method is the one that is used to be able to receive audio samples which are then written to the SourceDataLine. I create a new byte array which will be used to read the input buffer once I start the read method. After that, there is a while loop which includes the TargetDataLine read method and the SourceDataLine read method that I called through the AudioController class. The TargetDataLine read method reads the byte array data which contains the requested input data when the method is able to successfully run. The second parameter is the offset which is set to 0 to try to best combat any latency. Since the bytes being read must be an integral number of the sample frame so that it equals to zero, I specified the length as data.length to ensure that it equals to zero. While I did declare that the sample size in bits is 16-bits, the requirements of the read method asks for a byte array and thus, I initiated a byte array rather than a Short array. I then call in the SourceDataLine write method so that it will be able to write the audio samples in the byte array into the user's mixer which they would be able to hear the audio itself. Similarly, the bytes written must be an integral of its frame size such that it equals to zero and so I declared the SourceDataLine into a variable called readBytes which is then used in the parameters of the write method.

```java
    // Method to start the microphone line and allows to receive audio samples
    public void startLine() {
        microphoneLine.start();
        System.out.println("Microphone Started.");
    }

    // Method to stop the microphone line from inputting audio samples into the buffer
    public void stopLine() {
        microphoneLine.stop();
        System.out.println("Microphone Closed.");
    }

    // Getter to allow other classes to access the methods in this class
    public TargetDataLine getMicrophoneLine() {
        return microphoneLine;
    }

}
```

These methods are used to start and close the TargetDataLine when needed. There is a getter so that other classes can access the methods in this class which will be explained later on.

SpeakerOutputEngine Class

```java
public class SpeakerOutputEngine extends AudioSettings{

    private AudioController audio_c;
    private SourceDataLine speakerLine;
    private AudioFormat speakerFormat;

    // SpeakerOutputEngine constructor which allows access of this class and others through one java class
    public SpeakerOutputEngine(AudioController audio_c) {
        this.audio_c = audio_c;

        // Try/ Catch statement to open up the user's SourceDataLine
        try {
            // Specified audio format from the AudioSettings class
            speakerFormat = new AudioFormat(getSampleRate(), getSampleSizeInBits(), getChannels(), isSignData(), isBigEndian());

            // Receives the format and appends to the SourceDataLine
            speakerLine = AudioSystem.getSourceDataLine(speakerFormat);

            // Opens the SourceDataLine based on the format and the buffer size specified in AudioSettings class
            speakerLine.open(speakerFormat, getBufferSize());
            System.out.println("Speaker Line Opened.");

        } catch (LineUnavailableException e) {
            System.out.println("Speaker Line Unavailable!");
        }
    }
```

Similarly like the MicrophoneInputEngine class, the objects being declared are the AudioController class, the SourceDataLine class, and the AudioFormat class. Like mentioned before, the AudioController class is defined so that the methods in this class can be used elsewhere. The SourceDataLine will write the audio samples that are defined in a byte array and then append them into the user's mixer. The logic is the same as shown in the MicrophoneInputEngine class where the constructor consists of the AudioController as the parameter and try/catch statements to initiate the user's speaker output. A new audio format is initiated when the parameters being the methods declared in the AudioSettings class. Then, the program appends the format into a SourceDataLine called speakerLine. After that, it opens using the format and the getBufferSize() method to fully start the user's audio output. The catch statement will notify the user if the program isn't able to get their system's output.

```
    // Method to start and allow the user to hear the audio input
    public void startLine() {
        speakerLine.start();
        System.out.println("Speakers Started.");
    }


    // Method to stop the user from hearing their audio input by emptying out the speaker buffer and then stopping it
    public void stopLine() {
        speakerLine.drain();
        speakerLine.stop();
        System.out.println("Speakers Closed.");
    }

    // getter to allow other classes to access the methods in this class
    public SourceDataLine getSpeakerLine() {
        return speakerLine;
    }

}
```

These three methods are created to be able to control the audio input and output of the user. The drain method drains queued data from the line by continuing the I/O until the line's internal buffer has been emptied. The stop method is to turn off the user's mixer and output stream. The getter is to be able to access methods from this later on from the AudioController class.

SoundRecorder Class

```java
public class SoundRecorder extends AudioSettings implements Runnable {

    private AudioController audio_c;
    private AudioFormat recording_format;
    private String filename;
    private Scanner nScan = new Scanner(System.in);

    // SoundRecorder constructor which allows access of this class and others through one java class
    public SoundRecorder(AudioController audio_c) { this.audio_c = audio_c; }

    // Receives the user's input from a scanner and applies it to the getter for the filename
    public void fileName() { filename = nScan.nextLine(); }

    // A Thread that functions to record audio based on user specified input of file name and default parameters of a WAV file
    public void run() {
        try {
            // Receives the audio format from AudioSettings class
            recording_format = new AudioFormat( getSampleRate(), getSampleSizeInBits(), getChannels(), isSignData(), isBigEndian());

            // Creates a new file to allow the user to write their audio input into
            File wavFile = new File( pathname: getFileName() + ".wav");

            // Specify the audio file to be a WAV file
            AudioFileFormat.Type fileType = AudioFileFormat.Type.WAVE;

            // Creates a new AudioInputStream to be used in appending the audio samples from the input line in the
            // MicrophoneInputEngine class
            AudioInputStream ais = new AudioInputStream(audio_c.getMicrophoneInputEngine().getMicrophoneLine());

            // Starts recording
            AudioSystem.write(ais, fileType, wavFile);

            // Catches an exception if the user inputs an invalid character
        } catch (IOException e) {
            audio_c.stopMonitor();
            System.out.println("A filename cannot contain any of the following characters: \\ / : * ? \" < > | ");
        }
    }
```

The SoundRecorder class is similar to the MicrophoneInputEngine in the sense that it also inherits the variables and methods in the AudioSettings class and has an interface for threading. The objects initialized are the AudioController class, the AudioFormat class, a string titled filename, and a scanner for user input titled nScan. The constructor has the parameters AudioController which will allow the controller class to access the methods in this class. The filename method is a void method that takes the user inputted string and stores it into the variable filename. This variable will be used later on in the class to specify user inputted file names. After that, there is a run method which will be used when threading gets initialized. The method contains a try/catch statement which declares the format for the WAV file. After that, the program creates a new file with the filename specified by the user which gets stored into a variable called wavFile. After that, it specifies the file type to be a WAV file. A new AudioInputsStream is then initialized to fully complete the parameters needed for the AudioSystem class to write all of this data into. The catch statement is met when the user inputs invalid characters and stops the stream from continuing.

```
    // Stops the recording by invoking the stop line from the controller to the MicrophoneInputEngine class
    void stopRecording() {
        audio_c.getMicrophoneInputEngine().getMicrophoneLine().stop();
        System.out.println("Finished recording");
    }

    public String getFileName() { return filename; }

}
```

The stopRecording method is called when the user wants to stop recording audio and the getter is used to be able to receive user inputted strings for the title of the file.

AudioController Class

```
public class AudioController {

    // Declaring each of the classes
    private MicrophoneInputEngine mic_input_engine;
    private SpeakerOutputEngine speaker_output_engine;
    public Thread mic_engine_thread;
    private SoundRecorder recording_engine;
    public Thread recording_engine_thread;

    // Initializing to use all of the classes specified
    public AudioController() {
        System.out.println("Opening Audio Interfaces.");
        mic_input_engine = new MicrophoneInputEngine( audio_c: this);
        speaker_output_engine = new SpeakerOutputEngine( audio_c: this);
        recording_engine = new SoundRecorder( audio_c: this);
    }
```

I declare all of the classes mentioned before and two threads for the respective run methods mentioned before. The AudioController class has no parameters but initializes all of the classes mentioned before. I do this so that I won't have long lines of code and that it will be easier for me to manage specific things in my driver file.

```java
// Starts a monitoring stream using the startLine methods of the TargetDataLine and SourceDataLine
public void startMonitor() {
    mic_input_engine.startLine();
    speaker_output_engine.startLine();

    // Creates a new thread to continuously receive new audio samples to the input buffer
    mic_engine_thread = new Thread(mic_input_engine);
    mic_engine_thread.start();

}


// Stops the monitoring stream using the stopLine methods of the TargetDataLine and SourceDataLine
public void stopMonitor() {
    mic_input_engine.stopLine();
    speaker_output_engine.stopLine();

    // To terminate the microphone thread
    // Tried to use interrupt but the stream would still continue
    mic_engine_thread.stop();

}
```

The startMonitor method contains the two start methods of both the speaker output engine and the speaker output engine. Furthermore, a thread gets called which contains the MicrophoneInputEngine class. This allows the audio input to continuously get buffers of new audio samples which will get written into the user's mixer. The stopMonitor method is called when the user wants to stop monitoring their audio input. The method contains the method to close both the microphone input engine and the speaker output engine. Furthermore, I tried to turn off the threading by calling the interrupt method but the audio stream keeps going even when called. Thus, I used the stop method for threading which completely terminates the thread altogether.

```java
// Starts recording with the user being able to monitor the audio
public void startRecording() {
    mic_input_engine.startLine();
    speaker_output_engine.startLine();
    mic_engine_thread = new Thread(mic_input_engine);

    // Create a new thread for appending the audio input buffers to the WAV file
    recording_engine_thread = new Thread(recording_engine);
    mic_engine_thread.start();
    recording_engine_thread.start();

}

// Stops recording and turns off all threads and DataLines
public void stopRecording() {
    recording_engine.stopRecording();
    mic_input_engine.stopLine();
    speaker_output_engine.stopLine();

    mic_engine_thread.stop();
    recording_engine_thread.stop();

}

// Similar to the startRecording method but without starting the SourceDataLine
public void startRecordNoMonitor() {
    mic_input_engine.startLine();
    mic_engine_thread = new Thread(mic_input_engine);

    recording_engine_thread = new Thread(recording_engine);
    mic_engine_thread.start();
    recording_engine_thread.start();
}
```

The startRecording method's main function is to capture the input of the user and stream it to their mixer while simultaneously recording it. The startLine methods of the user's input and output gets called and the thread to continuously run the run methods in the respective classes gets called. After that, the method starts the two threads. The stopRecording method stops all the threads and all of the data lines that are present during the recording. The startRecordNoMonitor is the same as the starRecording method without the startLine method for the user's output.

```
// Getter to allow access from class to class
public SpeakerOutputEngine getSpeakerOutputEngine() {
    return speaker_output_engine;
}

public MicrophoneInputEngine getMicrophoneInputEngine() {
    return mic_input_engine;
}

public SoundRecorder getSoundRecorder() {
    return recording_engine;
}
```

These are the getters which allows the classes to access each other's methods.

Driver Class

```java
public class Driver {
    // method for the menu for the users to navigate
    public static int menu() {

        int selection = 0;
        Scanner sc = new Scanner(System.in);
        System.out.println("||      Welcome to Jot Down Audio!      ||");
        System.out.println("||Audio Monitoring and Recording Program||");
        System.out.println("Enter an integer to navigate");
        System.out.println("1. Stream audio input");
        System.out.println("2. Stop audio stream");
        System.out.println("3. Record while monitoring audio");
        System.out.println("4. Record without monitoring audio");
        System.out.println("5. Stop recording audio");
        System.out.println("6. Exit program");

        // Tru statement to make sure that the user inputs an integer
        try{
            int selection2;
            selection2 = sc.nextInt();
            selection = selection2;
        } catch (Exception e) {
            System.out.println("Invalid input. Please enter an integer");
        }

        return selection;
    }
```

The driver class has a menu where the user can navigate by inputting integers into the command line. The menu method will be the menu that the user will see. After that, there will be a try and catch statement where it takes the user's input and stores it into the variable selection2. After that, selection2 will be set to equal to selection which then gets returned.

```java
public static void main(String[] args) {

    AudioController audioHub = new AudioController();
    int userChoice;


    // do while loop for the menu
    do {
        // Have the menu() method inside the do while loop and have switch cases for the integer the user inputs
        userChoice = menu();
        switch (userChoice) {
```

The main args of the driver file contains the initiation of the AudioController class which inherently initiates all of the other classes. After that, there is a variable which gets initiated titled userChoice in which it will append the integer into the switch case expression. The menu is able to continuously run because it is constantly getting called inside a do while loop where the variable userChoice is equal to the selection variable in the menu method.

```java
            case 1:
                // Checks if the user has the mic engine thread on or not to ensure they can't continuously
                // have multiple threads open
                if (audioHub.mic_engine_thread == null) {
                    audioHub.startMonitor();
                } else {
                    System.out.println("Please turn off the Stream first!");
                }
                break;
```

Each of the cases represent the options that were present in the menu shown before. The first case contains an if statement to check if the user has already started an audio stream. If not, it starts the startMonitor method of the AudioController class and then breaks. By calling in the break statement it stops from iterating through each of the cases and goes back to the do while loop which contains the menu method which will get invoked.

```java
            case 2:
                // Checks if the user has turned on the monitor stream or not to be able to use this option
                try {
                    audioHub.stopMonitor();
                    audioHub.mic_engine_thread = null;
                    break;
                } catch (Exception e) {
                    System.out.println("Please turn on the Stream first!");
                    break;
                }
```

The second case allows the user to turn off the monitor stream. There is a try and catch statement to check if the user has the stream turned on to begin with. If not, then the program will notify the user to turn on the audio stream before being able to use this option.

```
case 3:
    // If statement to check if the recording thread is on or not to make sure they dont create
    // multiple recordings at once
    if (audioHub.recording_engine_thread == null) {
        try {
            System.out.println("Enter the name of the file: ");
            // uses the method in the soundRecorder class to get the file name
            audioHub.getSoundRecorder().fileName();
            // Set the file name
            String filename = audioHub.getSoundRecorder().getFileName();
            // Alter the sample rate of the input and output
            audioHub.startRecording();
            System.out.println("recording started\n");

            break;
        } catch (Exception e) {
            System.out.println("Unable to record");
            break;
        }
    } else {
        System.out.println("Audio recording in progress!\n");
        break;
    }
```

The third case has a if statement which contains a try catch statement inside it. It checks if the thread is turned off or not. If it is turned off it will continue to try and turn on the thread. If not, it goes to the else statement and notifies the user that they are already recording. The try/catch statement contains a printed message which asks the user to input the name for their file. After that, it stores the string into the fileName method in the SoundRecorder class and then sets the WAV file to the user input string. After that, it invokes the startRecording method.

```
            case 4:
                if (audioHub.recording_engine_thread == null) {
                    try {
                        System.out.println("Enter the name of the file: ");
                        // uses the method in the soundRecorder class to get the file name
                        audioHub.getSoundRecorder().fileName();
                        // Set the file name
                        String filename = audioHub.getSoundRecorder().getFileName();
                        // Alter the sample rate of the input and output
                        audioHub.startRecordNoMonitor();
                        System.out.println("recording started\n");

                        break;
                    } catch (Exception e) {
                        System.out.println("Unable to record");
                        break;
                    }
                } else {
                    System.out.println("Audio recording in progress!\n");
                    break;
                }
```

The 4th case is the same logic and structure as the 3rd case however, instead of invoking the startRecording method, it invokes the startRecordingNoMonitor method instead.

```
                case 5:
                    try {
                        audioHub.stopRecording();
                        audioHub.recording_engine_thread = null;
                        audioHub.mic_engine_thread = null;
                        break;
                    } catch (Exception e) {
                        System.out.println("Audio recording isn't in progress!\n");
                        break;
                    }
                case 6:
                    System.exit( status: 0);

                default:
                    System.out.println("Enter enter a correct integer\n");
            }

        } while (userChoice != 6);
    }
```

Case 5 is a try/catch statement in order to check if the user has an ongoing recording or not. If it does, it will try to stop the recording. Case 6 is to allow the user to quit the program and the default message is to notify the user to enter a correct integer.

Evidence of Working Program

```
Opening Audio Interfaces.
Microphone Line Opened.
Speaker Line Opened.
||      Welcome to Jot Down Audio!      ||
||Audio Monitoring and Recording Program||
Enter an integer to navigate
1. Stream audio input
2. Stop audio stream
3. Record while monitoring audio
4. Record without monitoring audio
5. Stop recording audio
6. Exit program
```

Main menu

```
1
Microphone Started.
Speakers Started.
||      Welcome to Jot Down Audio!      ||
||Audio Monitoring and Recording Program||
Enter an integer to navigate
1. Stream audio input
2. Stop audio stream
3. Record while monitoring audio
4. Record without monitoring audio
5. Stop recording audio
6. Exit program
```

After entering 1, the user is able to monitor themselves.

```
2
Microphone Closed.
Speakers Closed.
||      Welcome to Jot Down Audio!      ||
||Audio Monitoring and Recording Program||
Enter an integer to navigate
1. Stream audio input
2. Stop audio stream
3. Record while monitoring audio
4. Record without monitoring audio
5. Stop recording audio
6. Exit program
```

After entering 2, the user is able to turn of the audio monitoring

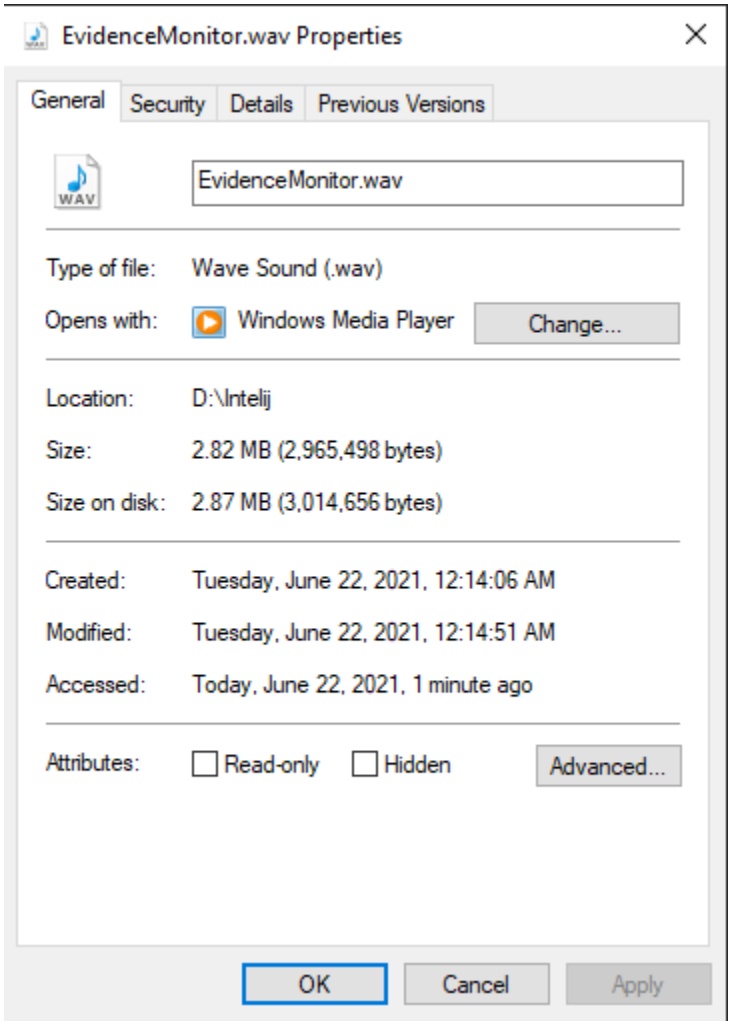
```
Enter the name of the file:
EvidenceMonitor
Microphone Started.
Speakers Started.
recording started

||      Welcome to Jot Down Audio!      ||
||Audio Monitoring and Recording Program||
Enter an integer to navigate
1. Stream audio input
2. Stop audio stream
3. Record while monitoring audio
4. Record without monitoring audio
5. Stop recording audio
6. Exit program
```

After entering 3, the user is able to record while monitoring their audio

```
Finished recording
Microphone Closed.
Speakers Closed.
||      Welcome to Jot Down Audio!      ||
||Audio Monitoring and Recording Program||
Enter an integer to navigate
1. Stream audio input
2. Stop audio stream
3. Record while monitoring audio
4. Record without monitoring audio
5. Stop recording audio
6. Exit program
```

After entering 5, the user is able to save the recording they initiated in 3 or 4



EvidenceMonitor.wav is created through the 3rd option