Java Labs 2021
Realtime audio processing in Java

Christophe Barès, Nicolas Papazoglou,
Sylvain Reynal, Antoine Tauvel, ENSEA

October 2021

Welcome to this lab. This lab is intended to be dealt with by a team of 4 students, but every student is assumed to be working on the project on their own: every team member is responsible for their own work but also for the work of fellow team members.

There are some rules to follow:

- As you are now at graduate entry level, we shall not correct missing semi-colons or similar stuff. Definitely not.
- Before the end of the first class session you should have a **working setup** and you should be able to **compile and execute** a simple "Hello World" program (see next section for more on this).
- Your backup, your problem.
- Please use a professional tool to monitor modifications made to your code. Git is the one you are looking for.

1 What this lab is about

We want to build an application that can record a sound from a microphone in real time, display the wave form and the spectrum on a chart, apply some effect (like an echo chamber), and play it back to the headphone plug.

The general app design could look like in Figure 1.

2 A simple User Interface with JavaFX

2.1 JavaFX vs Swing

This series of Labs is based on the JavaFx framework, a set of dedicated Java libraries that will allow you to build beautifully styled user interfaces (UI). JavaFX has become a neat, modern and efficient replacement to the Swing API

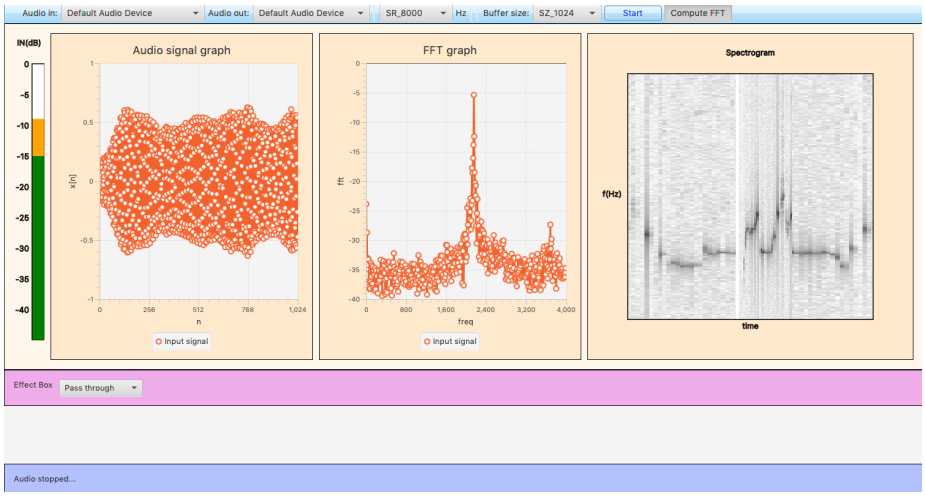

Figure 1: A possible app design.

(the JFrame, JButton, etc. components), an API that surfaced about more than twenty years ago as an attempt to create the first Java cross-platform widget[1] kit.

If you have been used to Swing components, you might find the philosophy behind JavaFX quite familiar at first, but this is merely the tip of the iceberg! JavaFX can do much more than Swing does, from fluid animations to CSS style sheet based styling to 3D shapes and transforms. Above all, JavaFX is hardware accelerated on most platform, either through DirectX on Windows, or through OpenGL on OS X or Linux. With all these features in its bag, it is thus quite easy to understand why JavaFX is slowly superseding Swing as the reference framework for modern UI's.

Building a User Interface in JavaFX revolves mostly around creating a component tree made of nodes (branches and leaves), where leaves can be widgets (buttons and co), shapes (lines, rectangle, etc), text (with tons of styling options), images, sound or videos, charts and tables, and branches are actually groups thereof (e.g., menu, toolbars, complex drawings), see Figure 2. There are also special nodes that support dynamic processing, like transforms (rotations, translations) and animation effects (e.g., you want to make a 3D box rotate for 3 seconds), and every visible node can have an event listener attached to it, just as was already the case with Swing. Finally, every component in the tree supports a CSS based styling scheme, with the usual "classes" and "id" parameters of CSS styling. This makes it possible to split appearance issues from functionality issues, and have someone specialized in graphic design take care of the UI appearance while developers are busy designing the interface itself.

[1] A "widget" is a component of a user interface like a button, a menu or a toolbar.

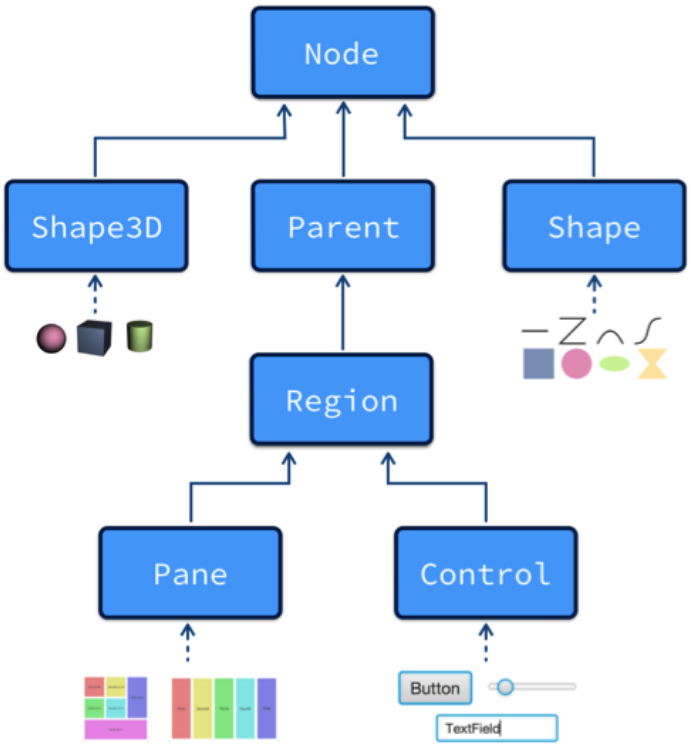

Figure 2: JavaFX component graph: the upper node is the root of the tree. It has three children, a Shape3D, a Shape2D, and a parent of a "Region", a group of nodes which itself aggregates together a Pane and a Control (e.g., a Button or a TextField). Applying a transform to a Region applies the transform to each of its children in turn, so that this is very efficient in terms of performance.

2.2 Software requirement / Homework

As of JavaFX, implementations are available on all major Operating Systems (Windows, Linux, MacOS)[2]. Screenshots provided below were obtained using Eclipse IDE, though you are free to use any IDE you want, from IntelliJ to Apache Netbeans (yet we strongly advise you against plain old Geany or any other IDE without refactoring or code completion ability as they swiftly prove useless when it comes to serious coding).

First thing we need is a basic setup to write code, compile and execute Java and JavaFX code[3]:

[2]The 3D extension of JavaFx is still not available for Unix / Linux users, yet we will not be using it

[3]Beware: the Java Development Kit (JDK) is not the same as the Java Runtime Exectution (JRE). JDK is somehow a JRE plus a compiler and a bunch of additional developer tools like a profiler, a decompiler, a tool that generates glue code when linking C code with Java, etc

- Download the last stable version of the Java Development Kit (aka JDK)
- Download the community edition of IntelliJ or any similar IDE (e.g. Eclipse or Netbeans are ok).
- Download the latest stable version of JavaFx from "openjfx.io". Unzip it in a folder which you will create on purpose, for example "javafx-libs" (it is good practice to pick a folder location inside the usual folder where you store your java projects so that you can quickly locate your JavaFX libraries when you will be asked during your project configuration ; another approach is to uncompress directly inside your eclipse workspace if you work with Eclipse). Write down the path to the JavaFx Library for later use.

2.3 Writing a basic JavaFx application

Writing a JavaFX application begins with overriding the `start()` method of the `javafx.application.Application` class. The argument of the `start()` method is a `Stage` object which represents the primary stage of the UI. You can have several stages: these simply correspond to distinct windows. Every `Stage` has a title (which is displayed in the window upper bar) and a `Scene` object attached to it, where the `Scene` is somehow the "top" of the component UI tree. A `Scene` has a dimension, a location on screen, and contains a root component (a `BorderPane` in the example given in Fig. 1) which can itself have children if it is a branch node (containers are always branch nodes so this is ok). Finally, a `Scene` can have one or several CSS style sheets attached to it.

Write and execute the simple JavaFX application of listing 1.

You can easily add text and 2D or 3D geometrical shapes, images, sounds and event videos to your interface by creating the appropriate object (e.g., a `Line` or an `Image`) and adding it to a parent container. This container may either be a :

- `Group` (in which case it just acts as a way to maintain a link between every node belonging to it, for example, if you add a `Transform` to a `Group`, then every node in the group undergoes this transform)
- Or a sophisticated container, like e.g. `BorderPane`, that can also layout its children neatly on the screen, in particular because it takes care of how its dimension influences the layout.

The package `javafx.scene.shape` contains a lot of 2D shapes, from lines to circles to Bezier splines. It also includes classes dedicated to 3D graphics (boxes, spheres, cylinders) which, when associated with transforms (see package javax`.scene.transform`), makes it possible to build complicated 3D worlds. More information is available in the API document of javafx, `http://docs.oracle.com/javase/8/javafx/api/toc.htm`.

2.4 Running a JavaFX app from the command line

Open a shell (or a "Command" or a "Terminal", depending on your OS), and go to where your code is located. Change to the folder where all compiled classes are located (e.g. "bin" when working with Eclipse), and then run the following command (everything on a single line):

```
java --module-path path-to-javafx-libs --add-modules ALL-MODULE-PATH
application.Main
```

where "path-to-javafx-libs" has to replaced by the path to your javafx libraries (remember? you add to write down somewhere where this folder is located...)

Hint: you may want to put the whole command above in a file, save this file as — for example — *run.bat* (or *run.sh* if you are working on Linux or MacOS), and simply execute your application by typing "run.bat" from the command line. This will save you some time!

3 Audio processing

3.1 Code architecture

You will now write your code inside java packages according to the following guidelines:

- the "ui" package is responsible for everything that has to be displayed on the screen as well as mouse event handling: Main (main JavaFX class), SignalView (signal visualization), VuMeter (dB level monitoring), Spectrogram, etc.
- the "audio" package is dedicated to audio processing classes, e.g., AudioSignal (signal container), AudioProcessor (audio I/O processor), and AudioIO (hub for I/O resources, e.g., microphones, headphones, etc).
- the "audio.effect" subpackage contained specialized audio effects, e.g., the Echo class, the PassThrough class, etc.
- the "math" package comprises classes that handle math operations, e.g., Complex numbers and Fast Fourier Transform.

Create the four packages above, and move your JavaFX code to the "ui" package. Check that it still works.

3.2 Basic audio processing with Java

There are two java packages dedicated to audio operations:

- the `javax.sound.sampled` package, for audio I/O
- the `javax.sound.midi` package, for MIDI[4] related development

We will stick to the `javax.sound.sampled` package from now on. Among the numerous classes inside this package, only a couple is really useful to scrutinize into:

- `SourceDataLine`: an audio rendering device (an audio line to which data may be written, e.g., a headphone)
- `TargetDataLine`: an audio capture device (a line from which audio data can be read, like a microphone)
- `Mixer` : an audio device with one or more (input or output) lines. You can think of a Mixer as a part of a soundcard. Usually there is one Mixer for audio outputs (headphones), and another for audio inputs (microphones).
- `Mixer.Info and Line.Info` : represent information about an audio mixer or an audio line, including the product's name, version, and vendor (for example : "USB Audio Device" or "External Headphones")
- `AudioFormat` : specifies the sample rate, sample size in bits, number of channels, etc
- `AudioSystem`: acts as the entry point to obtain audio resources. It is where most things start... since with no resource you cannot do anything.

You can do almost everything with these six classes, so take some time to go through their online documentation (googling `javax.sound.sampled` should be fine here although any serious java programmer should always have the complete java API documentation available in a web browser)

The basic process when one wants to play a sound is more or less the following one:

[4] MIDI is a serial communication bus for communication between electronic music instruments.

1. Obtain a `SourceDataLine`, either the **default** one (usually connected to the headphone plug), or for a specific mixer if your laptop is equipped with more than one soundcard or has multiple audio outputs ; to begin with, you may want to simply use:

   `AudioSystem.getTargetDataLine(AudioFormat)`

   where `AudioFormat` may be initialized with a sample rate of 8000Hz, 8 bits per sample, and a "monophonic" (one channel) configuration as in:

   `new AudioFormat(8000, 8, 1, true, true)`

   This will give you the **default** audio output, and if you are lucky, this corresponds to the embedded loudspeaker, or the default headphone plug :) If you cannot here anything, do not panic, this just means your default audio output is not the one you wanted and you will have to write a few more lines of code to correct this (see below, AudioIO class).

2. call `open()` and then `start()` on the `SourceDataLine` you just obtained.

3. Once you have a working `SourceDataLine`, it is enough to (periodically) fill a buffer of bytes with a sound wave, and then write this buffer to the audio output by means of the `SourceDataLine.write()` method.

The same process in the reverse way allows you to record sound from a microphone using a `TargetDataLine`.

3.3 Populating the audio package

We have to write three classes for our application to properly process audio:

- `AudioSignal`: this is a generic container for audio samples represented as double's ; if offers methods to modify sample values, get the value of particular sample, compute the level of the signal in dB.
- `AudioProcessor` : the main audio processor that offers methods to record audio signals from a microphone, play signals to a headphone, and compute the FFT of the input signal ; it also offers the ability to run concurrently as a Thread (question: why is this crucial when doing audio inside a graphic application?)
- `AudioIO` : a "hub" offering methods to retrieve audio lines (microphones, headphones, etc), and which knows how to start the whole audio processing.

Create this three (so far empty) audio classes inside the audio package.

3.3.1 The AudioSignal class

A skeleton for AudioSignal is shown in Listing 2. Audio samples are stored in double precision in an array that can be filled by the `recordFrom()` method, or played back with the `playTo()` method. The proposed implementation corresponds to 16 bit wide samples, but you can stick to 8 bit sampling (just make sure you use the appropriate parameters for the `AudioFormat`).

Implement the empty methods in the AudioSignal class skeleton. How are you going to test it? You may want to write a small piece of test code inside a main() method located inside the AudioSignal class and run your test using the command `"java audio.AudioSignal"` until you are sure the class does what you want.

3.3.2 The AudioIO class

The audio.AudioIO class is a collection of **static** functions[5] related to the audio system. A skeleton is shown in Listing 3.

A few comments:

- `static void printAudioMixers()` : Print a list of every audio mixer available on the current system. Hint: instead of using a for() loop, we show you how using streams yield a much more compact code. With `Arrays.stream()` (from `java.util.Arrays`), it is easy to turn an array into a stream, and then call `forEach()` on it... Kind of Python programming in Java!

- `static Mixer.Info getMixerInfo(String mixerName)` : Return a Mixer.Info whose name matches the given string. This will prove useful later on, when selecting a mixer from a JavaFX ComboBox where only strings are stored.

- `static TargetDataLine obtainAudioInput(String mixerName, SampleRate sampleRate)` : Return a line that is appropriate for recording sound from a microphone. Example of use:

  ```
  TargetDataLine line = obtainInputLine("USB Audio Device", 8000).
  ```

  Hint: you may need the `getMixerInfo()` method defined above. Once you have a `Mixer.Info`, you can obtain a `Mixer` with `AudioSystem.getMixer()`, then call `getTargetLineInfo()` on this mixer to obtain a list of specific lines supported by this mixer, check which line supports the audio format

[5] Remember: a static method is a C-like function in Java, a kind of "standalone" method! That is, a function which is by no means related to a class or an object. This explains why you can call a static method without having to instantiate an object with the "new" keyword. The only thing you need in order to call a static method is to prepend the name of the function with the name of the class where it is located (but think of the class here as a simple "storage" where you can store static functions that make sense together). A famous example is the collection of math function like Math.cos() or Math.log() that are all stored inside the Math class, although you never have a to instantiate a Math object before calling cos() or log(). This is because they are all static!

you want (not every line supports stereo for example), and eventually obtain a `TargetDataLine` from the `AudioSystem.getTargetDataLine(AudioFormat, Mixer.Info)` method. Quite cumbersome, but unfortunately this is the only reliable way of obtaining exactly the audio line you want (and once connected to the JavaFX interface, this will definitely be far straightforward to implement in a ComboBox object).

Implement empty methods in the AudioIO class skeleton. How are you going to test them? Here again you may want to write a small piece of test code inside a main() method located inside the AudioIO class and run your test using `java audio.AudioIO` until you are sure the class does what you want. Feel free to use AudioSignal here inside the test code now that you know it works properply!

3.3.3 The AudioProcessor class

This class is responsible for performing the audio processing. It takes an input AudioSignal, fills it from a TargetDataLine, applies some effect, and then writes the resulting output AudioSignal to a SourceDataLine. A skeleton is shown in Listing 4. This class implements the Runnable interface, which means the code inside the `run()` method can be executed in a separated `Thread`(through a hidden fork() process) using the following code:

```
Thread t = new Thread(new AudioProcessor()); t.start();
```

Having a **multithreaded** implementation of your code becomes mandatory when running the audio processing code from the JavaFX graphic interface as what would happen otherwise is that the audio processing code would block the whole JavaFX process until it is finished (that is, until the infinite while loops terminate! a non-sense...), making the UI unresponsive.

From here on, if your test code for AudioProcessor works as expected, we suggest you a couple of things for the final polishing of the audio package:

- For a higher reliability of your code, you may want to move all audio initialization steps inside a single place, for exemple inside the AudioIO class:

  ```
  void startAudioProcessing(String inputMixer, String outputMixer,
  int sampleRate, int frameSize){ ... }
  ```

  which could obtain the appropriate Target/SourceDataLine's, instantiate an AudioProcessor, open and starts the audio line's above, create a Thread for the AudioProcessor and start it

- add another method to AudioIO that would just stop the AudioProcessor thread:

  ```
  void stopAudioProcessing(){ ...  }
  ```

This way all the resource initialization and the thread management is done from a single place, namely the AudioIO class.

4 Connecting the audio part with JavaFX

If everything inside the audio package works as expected, it is time to connect it to classes inside the "ui" package. We will be much less explicit here, as if you have reached this point this probably means you may have gained sufficient autonomy to design things on your own.

There is a first series of things to be done to make it possible for the user to set audio parameters (sample rate, audio inputs and outputs, etc) and start audio processing directly from the JavaFX interface: this is the role of the various widgets in the Toolbar, see Figure 1. Here it will prove useful to resort not only to Button's, but also to ComboBox's, using the various static methods inside AudioIO to fill the content of the various ComboBox's. Beware here: it is strictly forbidden to call a resource intensive piece of code from inside an EventHandler! (this should remind you of a similar rule when writing IRQ handlers for microcontrollers...). So as general rule of thumb: always move resource intensive code inside a separate thread, and just start the thread from inside your EventHandler, letting it execute nicely on its own (whatever the time this resource intensive task takes to terminate, YOUR EventHandler can swiftly terminate and give the control back to the JavaFX machinery, and this machinery can thus proceed back to handling mouse events and displaying things on the screen... otherwise it would just FREEZE). As a rule: audio processing is resource intensive ; writing huge files to the network is resource intensive ; doing artificial intelligence calculation is resource intensive ; System.out.println() alone... is not. See the point?

Then, as a second item of business, you may want to consider the following general architecture for the rest of the "ui" package:

- write a `ui.SignalView` class that extends `LineChart<Number,Number>` and can display an `AudioSignal`. You need to have a method `updateData()` that would update the chart content from an `AudioSignal`. This method may be called periodically from a `javafx.animation.AnimationTimer`, which is probably the most efficient way as THIS timer knows what is the best update frequency (it is useless and thus inefficient to update charts at a higher pace than that of the whole JavaFX interface, usually around 10Hz!).

- write a `ui.VuMeter` class that can displays the signal level (see Figure 1). It may extend `Canvas` and draw a vertical green/orange/red rectangle depending on the signal level. You may want to use `Canvas.getGraphicsContext2D()` to write geometrical shapes and fill them with colours. Here again you need an `update()` method that can update the color and size of the `VuMeter` rectangle, and may also be called from the same `AnimationTimer` as above.

- at this point, you may want to add spectrum vizualisation: for this you will first need to add a `Complex[] computeFFT()` method to the `AudioSignal` class. You may want to base your code on

  `https://introcs.cs.princeton.edu/java/97data/Complex.java.html`

and

`https://introcs.cs.princeton.edu/java/97data/FFT.java.html`

It is probably time to create a `math` package and add these two classes, `FFT` and `Complex`, to it, and then test them separately until they work as expected (for example by feeding the FFT algorithm with a well known signal created by hand).

Then if everything works great, create a `ui.Spectrogram` class that extends `Canvas`. We will use a `WritableImage` to draw the spectrogram in real time (see the rightmost image in Figure 1 to see what it could look like). `GraphicsContext.drawImage()` in effect displays this image on the screen (more exactly: inside the Canvas area). Using the `getPixelWriter()` method, you can obtain a tool that allwos you to modify the image pixels one by one, which, with some clever coding, should get the job done... The idea is to draw one vertical line after the other, filling each line with the (absolute value of the) coefficients of the Fourier transform of the signal...

```java
import ... (your job)

public class Main extends Application {

 public void start(Stage primaryStage) {
  try {
   BorderPane root = new BorderPane();
   root.setTop(createToolbar());
   root.setBottom(createStatusbar());
   root.setCenter(createMainContent());
   Scene scene = new Scene(root,1500,800);
   primaryStage.setScene(scene);
   primaryStage.setTitle("The JavaFX audio processor");
   primaryStage.show();
  } catch(Exception e) {e.printStackTrace();}
 }

 private Node createToolbar(){
  Button button = new Button("appuyez !")
  ToolBar tb = new ToolBar(button, new Label("ceci est un label"), new Separator());
  button.setOnAction(event -> System.out.println("appui!"));
  ComboBox<String> cb = new ComboBox<>();
  cb.getItems().addAll("Item 1", "Item 2", "Item 3");
  tb.getItems().add(cb);
  return tb;
  }

 private Node createStatusbar(){
  HBox statusbar = new HBox();
  statusbar.getChildren().addAll(new Label("Name:"), new TextField("    "));
  return statusbar;
  }

 private Node createMainContent(){
  Group g = new Group();
  // ici en utilisant g.getChildren().add(...) vous pouvez ajouter tout élément graphique souhaité de type Node
  return g;
 }
}
```

Listing 1: Building the component UI tree is simple: create a root node (here a `BorderPane`), fill its various inner areas (top, bottom, center), then attach it to a `Scene` and attach the `Scene` to the `Stage`. The traditionnal "static void main(String[] args)" method is optional as JavaFX always starts an application with a call to the start() method, no matter if main() is present or not.

```java
/** A container for an audio signal backed by a double buffer so as to allow floating point calculation
 * for signal processing and avoid saturation effects. Samples are 16 bit wide in this implementation. */
public class AudioSignal {

        private double[] sampleBuffer; // floating point representation of audio samples
        private double dBlevel; // current signal level

        /** Construct an AudioSignal that may contain up to "frameSize" samples.
         * @param frameSize the number of samples in one audio frame  */
        public AudioSignal(FrameSize frameSize) { ... your job ... }

        /** Sets the content of this signal from another signal.
         * @param other other.length must not be lower than the length of this signal. */
        public void setFrom(AudioSignal other) { ... your job ... }

        /** Fills the buffer content from the given input. Byte's are converted on the fly to double's.
         * @return false if at end of stream */
        public boolean recordFrom(TargetDataLine audioInput) {
                byte[] byteBuffer = new byte[sampleBuffer.length*2]; // 16 bit samples
                if (audioInput.read(byteBuffer, 0, byteBuffer.length)==-1) return false;
                for (int i=0; i<sampleBuffer.length; i++)
                        sampleBuffer[i] = ((byteBuffer[2*i]<<8)+byteBuffer[2*i+1]) / 32768.0; // big endian
                // ... TODO : dBlevel = update signal level in dB here ...
                return true;
        }

        /** Plays the buffer content to the given output.
         * @return false if at end of stream */
        public boolean playTo(SourceDataLine audioOutput) { ...  your job ... }

    // your job: add getters and setters ...
    // double getSample(int i)
    // void setSample(int i, double value)
    // double getdBLevel()
    // int getFrameSize()
    // Can be implemented much later: Complex[] computeFFT()
}
```

Listing 2: A skeleton for the AudioSignal class.

```java
/** A collection of static utilities related to the audio system. */
public class AudioIO {

        /** Displays every audio mixer available on the current system. */
        public static void printAudioMixers() {
                System.out.println("Mixers:");
                Arrays.stream(AudioSystem.getMixerInfo())
                .forEach(e -> System.out.println("- name=\"" + e.getName()
                + "\" description=\"" + e.getDescription() + " by " + e.getVendor() + "\""));
        }

        /** @return a Mixer.Info whose name matches the given string.
        Example of use: getMixerInfo("Macbook default output") */
        public static Mixer.Info getMixerInfo(String mixerName) {
            // see how the use of streams is much more compact than for() loops!
            return Arrays.stream(AudioSystem.getMixerInfo())
                .filter(e -> e.getName().equalsIgnoreCase(mixerName)).findFirst().get();
        }

        /** Return a line that's appropriate for recording sound from a microphone.
         * Example of use:
         * TargetDataLine line = obtainInputLine("USB Audio Device", 8000);
         * @param mixerName a string that matches one of the available mixers.
         * @see AudioSystem.getMixerInfo() which provides a list of all mixers on your system.
         */
        public static TargetDataLine obtainAudioInput(String mixerName, int sampleRate){ ... }

        /** Return a line that's appropriate for playing sound to a loudspeaker. */
        public static SourceDataLine obtainAudioOutput(String mixerName, int sampleRate){ ... }

        public static void main(String[] args){ ... your test code here ...}


```

Listing 3: A skeleton for the AudioIO class.

```java
/** The main audio processing class, implemented as a Runnable so
 * as to be run in a separated execution Thread. */
public class AudioProcessor implements Runnable {

	private AudioSignal inputSignal, outputSignal;
	private TargetDataLine audioInput;
	private SourceDataLine audioOutput;
	private boolean isThreadRunning; // makes it possible to "terminate" thread

	/** Creates an AudioProcessor that takes input from the given TargetDataLine, and plays back
	 * to the given SourceDataLine.
	 * @param frameSize the size of the audio buffer. The shorter, the lower the latency. */
	public AudioProcessor(TargetDataLine audioInput,  SourceDataLine audioOutput, FrameSize frameSize) {
	    ... your job ... }

	/** Audio processing thread code. Basically an infinite loop that continuously fills the sample
	* buffer with audio data fed by a TargetDataLine and then applies some audio effect, if any,
	* and finally copies data back to a SourceDataLine.*/
	@Override
	public void run() {
		isThreadRunning = true;
		while (isThreadRunning) {
			inputSignal.recordFrom(audioInput);

			// your job: copy inputSignal to outputSignal with some audio effect

			outputSignal.playTo(audioOutput);
		}
	}

	/** Tells the thread loop to break as soon as possible. This is an asynchronous process. */
	public void terminateAudioThread() { ... your job ... }

	// todo here: all getters and setters

	/* an example of a possible test code */
	public static void main(String[] args) {
		TargetDataLine inLine = AudioIO.obtainAudioInput("Default Audio Device", 16000);
		SourceDataLine outLine = AudioIO.obtainAudioOutput("Default Audio Device", 16000);
		AudioProcessor as = new AudioProcessor(inLine, outLine, 1024);
		inLine.open(); inLine.start(); outLine.open(); outLine.start();
		new Thread(as).start();
		System.out.println("A new thread has been created!");
	}
}
```

Listing 4: A skeleton for the AudioProcessor class.