PIC 20A: Homework 5 (due 3/17 at 5pm)

Submitting your homework

The zip file you extracted to find this pdf includes a file called `WAVHeader.java`.
In this assignment, you will edit this file and submit it to Gradescope.

- Upload `WAVHeader.java` to **Gradescope** before the deadline.
- **Name** the file exactly as just stated.
- Do **not** enclose `WAVHeader.java` in a folder or zip it. (Do not place it inside a package.)

  Do **not** submit `WAVStereoException.java`, `WAV.java`, `Test.java`, or `Test2.java`.

  You should be submitting exactly **one file** and it should have the **extension** `.java`.
- Be sure that your code **compiles and runs** with the audio files, `WAVStereoException.java`, `WAV.java`, and `Test.java` using **Adoptium's Temurin Version 11 (LTS)**.

Tasks

1. Check that you can play

   - `fmt1_chan1_sr44100_bit16.wav`
   - `fmt1_chan2_sr44100_bit16.wav`
   - `fmt1_chan1_sr48000_bit16.wav`
   - `fmt1_chan2_sr48000_bit16.wav`
   - `fmt1_chan1_sr44100_bit24.wav`
   - `fmt1_chan2_sr48000_bit24.wav`
   - `fmt3_chan1_sr48000_bit32.wav`
   - `fmt3_chan2_sr44100_bit32.wav`
   - `euclid.wav`

   Confirm that `not_WAV.wav` does not play.

2. Confirm that you can compile and run `Test.java`. Using an IDE will complicate this process (add it to the list of ways an IDE makes your life less good) since you will have to store the files in an appropriate place and/or give full paths to their location. If you use the Terminal or command prompt, after `cd`ing to the directory called `HW5`, you'll type `javac Test.java` followed by `java Test` and it will work.

   Notice that two files are created: `220_m.wav` and `220_440_s.wav`. Before solving the homework assignment, these will not play successfully (even though they store audio data).

3. Open `WAVHeader.java`.

   (a) A WAV file consists of two parts: a part before the audio data, what we'll refer to as the header, and the audio data (sometimes together with footers which we'll ignore).

   (b) `WAVHeader.java` allows us to store the information found in a WAV file *header*.
   It also allows the reading and writing of WAV file headers.

   (c) Read the description before the class interface.
   Read the section which explains the format of a WAV header (before the instance fields).
   Look at the declarations of the instance fields.
   Read the section which explains the format of a WAV header again,
   darting back and forth between the instance fields as you do so.
   Spend a long time doing this,
   especially thinking about the number of bytes and data types when I mention them.
   Remember that a `short` is 2 bytes and that an `int` is 4 bytes. Also, remember that ASCII characters can be stored using one byte - this is what the WAV file does - but that JAVA `char`s are two bytes (storing ASCII characters by having one of the bytes set to `0`).
   It's essential to understand the structure of WAV header, but not necessary to have a deep understanding of what the instance fields mean in DSP.

   (d) Understand the initializer block.

   (e) You don't need to understand the constructor `WAVHeader(int, int, int, int, int)` carefully. It's probably a little more satisfying if you do, but what matters most is that you realize there are various quantities…

   - `audio_fmt`,
   - `channels`,
   - `sample_rate_per_chan`,
   - `bit_depth_of_sample`,
   - `samples_per_channel`

   …which determine all of the others.
   Lots of the WAV files in `HW5.zip` are labelled by the first four quantities. For those files, the last quantity is determined by the formula `8 * sample_rate_per_chan` because there's 8 seconds of audio.

(f) Compact disk quality audio uses a sample rate (per channel) of $44,100$Hz, a bit depth of 16, and this forces the audio format to be 1. `makeWAVHeaderForCompactDiskQualityAudio` calls the previous constructor with those values and `channels` set to `1` or `2` depending on whether it is the mono or stereo version. Notice how the stereo version throws a custom exception if the lengths of arrays for the left and right channels disagree.

(g) `readWAVHeaderFromFile` reads in the first 4096 bytes of a WAV file. It is careful about catching and sometimes rethrowing exceptions, but otherwise, all it does is call a private constructor with those bytes passed as an argument: `return new WAVHeader(bytes)`. It is one of your jobs to make sure that this constructor does the correct thing. Now you see why I suggested that you pay attention to the number of bytes each piece of information uses.

(h) Upon running `Test.java`, you can see the result of 10 calls to `toString`. This will gives a useful way to track your progress on `WAVHeader(byte[])`.

(i) Making sure that you understand the numbers in the definition of `getHeaderSize` will be a good way to assess how well you are understanding the structure of a WAV file header. `getDataSize` is easy.

(j) `private WAVHeader(byte[] b)` and `public byte[] getBytes()` have bad definitions. Your homework assignment is to fix them.

4. Open `WAVStereoException.java` and read it. Look back on the definition of

   `makeWAVHeaderForCompactDiskQualityAudio(short[], short[])`

   to see why it exists and how it is used.

5. Open `WAV.java`. It is useful if you understand this file.

   The most important things to take away are...

   - A WAV file header is written by a call to `fout.write(headerBytes)` where `headerBytes` is assigned the return value of `header.getBytes()`.
   - The audio data is written by a call to `fout.write(dataBytes)` where `dataBytes` is filled up using a `ByteBuffer` and calls to `putShort`.

   I suggest that you write `getBytes` by using a `ByteBuffer` and calls to `put`, `putShort`, and `putInt`.

6. Open `Test.java`.

   - The calls to `WAVHeader.readWAVHeaderFromFile` should be used to check your progress on `WAVHeader(byte[])`. The correct output is listed at the end and saved in `output.txt`. The last two test cases are important since one involves `junk`, and the other involves a bad `riff_header`, a bad `wave_header`, and a bad `fmt_header`.
   - If your `getBytes()` method is correct (except perhaps with regards to junk),
     then the audio files written, `220_m.wav` and `220_440_s.wav`, will sound correct.

7. Write `private WAVHeader(byte[] b)`.
   - I would wrap the bytes in a `ByteBuffer`.
   - Then you can extract the information you need by calling `get`, `getShort`, and `getInt`.
   - Remember, WAV files store ASCII characters using one byte, not like JAVA with `chars`.
   - **Remember the last two test case outputs in** `output.txt`.

8. Write `public byte[] getBytes()`.
   - You need to make sure that the array you return has the correct size.
   - To fill it up, I would wrap it in a `ByteBuffer`.
   - Then you can create `bytes` by calling `put`, `putShort`, and `putInt`.
   - If you are correct regarding junk, then `Test2.java` will print `true`.
     You should make sure that `private WAVHeader(byte[] b)` handles junk correctly first.

```
riff_header          RIFF
riff_size            705636
wave_header          WAVE
fmt_header           fmt
fmt_size             16
audio_fmt            1
channels             1
sample_rate_per_chan 44100
bytes_per_sec        88200
block_size_in_bytes  2
bit_depth_of_sample  16
data_header          data
data_size            705600

riff_header          RIFF
riff_size            1411236
wave_header          WAVE
fmt_header           fmt
fmt_size             16
audio_fmt            1
channels             2
sample_rate_per_chan 44100
bytes_per_sec        176400
block_size_in_bytes  4
bit_depth_of_sample  16
data_header          data
data_size            1411200
```

```
riff_header          RIFF
riff_size            768036
wave_header          WAVE
fmt_header           fmt
fmt_size             16
audio_fmt            1
channels             1
sample_rate_per_chan 48000
bytes_per_sec        96000
block_size_in_bytes  2
bit_depth_of_sample  16
data_header          data
data_size            768000

riff_header          RIFF
riff_size            1536036
wave_header          WAVE
fmt_header           fmt
fmt_size             16
audio_fmt            1
channels             2
sample_rate_per_chan 48000
bytes_per_sec        192000
block_size_in_bytes  4
bit_depth_of_sample  16
data_header          data
data_size            1536000
```

```
riff_header          RIFF
riff_size            1058436
wave_header          WAVE
fmt_header           fmt
fmt_size             16
audio_fmt            1
channels             1
sample_rate_per_chan 44100
bytes_per_sec        132300
block_size_in_bytes  3
bit_depth_of_sample  24
data_header          data
data_size            1058400

riff_header          RIFF
riff_size            2304036
wave_header          WAVE
fmt_header           fmt
fmt_size             16
audio_fmt            1
channels             2
sample_rate_per_chan 48000
bytes_per_sec        288000
block_size_in_bytes  6
bit_depth_of_sample  24
data_header          data
data_size            2304000
```

```
riff_header          RIFF
riff_size            1536036
wave_header          WAVE
fmt_header           fmt
fmt_size             16
audio_fmt            3
channels             1
sample_rate_per_chan 48000
bytes_per_sec        192000
block_size_in_bytes  4
bit_depth_of_sample  32
data_header          data
data_size            1536000

riff_header          RIFF
riff_size            2822436
wave_header          WAVE
fmt_header           fmt
fmt_size             16
audio_fmt            3
channels             2
sample_rate_per_chan 44100
bytes_per_sec        352800
block_size_in_bytes  8
bit_depth_of_sample  32
data_header          data
data_size            2822400
```

```
riff_header          RIFF
riff_size            35478580
wave_header          WAVE
junk_header          JUNK
junk_size            28
fmt_header           fmt
fmt_size             16
audio_fmt            1
channels             2
sample_rate_per_chan 44100
bytes_per_sec        176400
block_size_in_bytes  4
bit_depth_of_sample  16
data_header          data
data_size            35476048

riff_header          NOPE
riff_size            1411236
wave_header          BYTE
fmt_header           Oops
fmt_size             16
audio_fmt            1
channels             2
sample_rate_per_chan 44100
bytes_per_sec        176400
block_size_in_bytes  4
bit_depth_of_sample  16
data_header          data
data_size            1411200
```