CSE1322L - Assignment 6 (Summer 2025)

Introduction:

Files, like anything else in a computer, are composed of bytes, which are sequences of 8 bits. In order to read the information contained in a file, a program must know its structure. Each file has its own structure, and these structures may be available online if the structure is an open standard. As long as a program knows how to decode this structure, it will be able to read any file that follows that structure (assuming the file isn't corrupted). Here are the specifications of some well-known file structures:

- MS-DOCX: https://learn.microsoft.com/en-us/openspecs/office_standards/ms-docx/b839fe1f-e1ca-4fa6-8c26-5954d0abbccd
- JPEG: https://jpeg.org/jpeg/
- MP3: https://www.loc.gov/preservation/digital/formats/fdd/fdd000105.shtml

Likewise, if your program wishes to make changes to the contents of a file in such a way that other programs are still able to read and interpret the changes, your program needs to know how to encode the information according to the file's structure and place it in the correct spot.

This is also true for text files which the computer is expected to be able to understand, which are usually called "Machine Readable Files". While files containing textual structure generally occupy more disk space compared to files using a byte structure, textual files are a lot easier for humans to read, making it easier to understand and work with them. A good example of that would be the FYE Website: If you visit the site, right-click anywhere, and click on "Inspect", your browser will open the textual representation of the FYE Website, which the computer uses to display the site and which you can use to understand how the site is structured.

All of this implies that you can create your own file formats and, as long as this format is known by whoever is receiving it, they will be able to read its contents. To prove this, we will create our own file format.

In this assignment, you will write the file reader for a new music format, known as the FYEMUS Standard. You will be provided with some files that follow this standard (their file format will be *.fyemus), some mp3 audios of what the files are supposed to sound like, and a Java file containing a music player that also follows that standard, so you can play the music files.

The FYEMUS Standard

FYEMUS files are text files (UTF-8 encoded) which contain numbers between 0 and 127 inclusive and whitespaces. If an FYEMUS file contains any characters besides the ones above, it is not a valid FYEMUS file.

FYEMUS files contain MELODIES, which are the largest unit of information in the standard. A MELODY has two components: the NOTE COUNT, and a list of FYENOTES. The NOTE COUNT is a number between 0 and 127 inclusive. Empty files are not valid FYEMUS files. The NOTE COUNT will then be followed by a list of FYENOTES, which can contain between 0 and 127

FYENOTES (inclusive). The number of FYENOTES in a MELODY is determined by the NOTE COUNTER.

An FYENOTE is a pair of two numbers, both between 0 and 127 inclusive, where the first number is the actual note to be played, and the second number is for how long it must be played (the note's timing). The note's timing is expressed in 1/16th of a second (one sixteenth of a second). As such, if a note's timing is 1, it should be played for 1/16 seconds; if it is 2 it should be played for 2/16 seconds; if it is 4, it should be played for 4/16 seconds; if it is 16, it should be played for 1 second (16/16 seconds); etc.

(The following paragraph is not relevant for your assignment. Feel free to skip it) A note is a sound that can be produced by an instrument. A note of 0 means "play no sound". All of the following notes, up to and including 127, follow the MIDI Standard: middle C is coded 60, higher-pitched notes are numbers between 61 and 127 inclusive and lower-pitched notes are numbers between 1 and 59 inclusive. Sharps appear after their respective base notes in a piano: 60 is middle C, 61 is middle C#, 62 is middle D, 63 is middle D#, 64 is middle E, 65 is middle F, etc. The FYEMUS standard also supports MELODY concatenation for multi-melody files, but FYEMUS file readers are under no obligation to implement this: any data past the last FYENOTE is not part of the current MELODY and can be ignored when reading that MELODY. Below, you can see the expressions that form the FYEMUS standard:

FYEMUS_FILE	→	MELODY+
MELODY	→	NOTE_COUNT FYENOTE_LIST
FYENOTE_LIST	→	FYENOTE*
FYENOTE	→	NOTE TIMING

Where NOTE_COUNT, NOTE, and TIMING are all numbers between 0 and 127 inclusive.

Here are some examples, with the fyenotes underlined for clarity, and the explanation on whether the file is valid or not under the file contents:

(empty file)

Invalid file. FYEMUS files must always contain at least the note count; this file is empty.

```
0
```

Valid file. It contains the note count, 0, and is followed by 0 fyenotes.

```
1 60 16
```

Valid file. It contains the note count, followed by at least that many fyenotes. The following two numbers, 60 and 16, compose the fyenote: the note to be played is middle C and its timing is 1 second.

```
2 60 8
```

Invalid file. It contains the note count, 2, but there is only one fyenote: middle C (60) being played for half a second (8).

```
4 60 8 62 8 64 8 65 8
```

Valid file. It contains the note count, 4, and at least 4 fyenotes: middle C (60), middle D (62), middle E (64), and middle F (65), each played for half a second (8).

```
4 60 16 62 16 64 16 65
```

Invalid file. The last fyenote ends abruptly as it does not contain its timer.

```
4 60 16 62 16 64 16 65 16 67 16 69 16 71 16
```

Valid file. It contains the note count, 4, and at least 4 fyenotes. The information after the last note does not belong to this melody and could represent anything. You should probably just ignore it.

```
0 60 16 62 16 64 16 65 16 67 16 69 16 71 16
```

Valid file. Contains the note count, 0, so nothing else needs to be read. The information after the note count does not belong to this melody and could represent anything. You should probably just ignore it.

```
-1 60 16
```

Invalid file. The note count is a negative number. None of the numbers in the file can be negative.

```
128 60 16
```

Invalid file. The note count is greater than 127. None of the numbers in the file can be greater than 127.

Requirements

The features described below must be in your program. Before starting, copy and paste "FYENote.java" into your project, as some of the classes mentioned below are in it. You can find that file in the zip folder "supplemental.zip".

- A class FYEMusicReaderException, which is a subclass of FYEMusicException. It has an overloaded constructor which accepts a string and passes it to its superclass constructor.
- A class Note, which implements the FYENote interface.
    1. Create two fields to store the object's information: the note that needs to be played and its timing. You can find out what their types and names are by checking the methods the interface requires you to implement.
    2. **Note(byte, int)**: Initializes the object's fields with the arguments. No need to check if the values are valid according to the specification: this will be done elsewhere.
- Your driver class will have 2 static methods:
    1. **static ArrayList<FYENote> loadMusic(Scanner)**

- Throws any explicit exceptions it encounters (i.e.: there should be no TRY-CATCH blocks in this method's body)
- This method uses the argument's nextByte() to retrieve information from the underlying file. The first byte read is the note counter. The method then tries to read that many FYENotes from the file (also using nextByte() as required). A Note object must be created for each FYENote read, and these notes must be stored in an arraylist. The method should then return this arraylist.
- **The timing read from the file needs to be adjusted**. The standard specifies that the timing is stored as 1/16 of a second. However, the music player provided works in microseconds. As such, you need to convert the timing you read in the file into microseconds before passing it to the Note's constructor. 1 second is 1000000 (one million) microseconds, so if you read "16" for a FYENote's timing, the corresponding Note object must have its timing field set to 1000000.
- If any of the bytes read is a negative number, the method must throw a FYEMusicReaderException with one of the following messages:
  - "Note counter out of range", if the note counter is negative
  - "Note out of range", if the note is negative
  - "Timing out of range", if the timing is negative
- Likewise, any bytes above 127 are also invalid and must throw exceptions. However, if you stick to only using nextByte() you will **not** need to throw any exceptions manually.
  - If you wish to do this manually, know that non-numbers and numbers above 127 must throw a "InputMismatchException".

- **static void main():** Must implement the menu options below:
  1. **Load music:**
     - Prompts the user for a file name.
     - Creates a File object with that information, which is itself used to create a Scanner object. This Scanner is then passed to loadMusic().
     - If an arraylist of FYENotes is returned successfully, passes that arraylist to FYEMusicPlayer.loadNotes().
     - Any exceptions of type FYEMusicReaderException must be caught explicitly, printing an error message saying "Unable to load file: ", followed by the message in the exception.
     - Exceptions of type FileNotFoundException must be caught explicitly. If caught, the error message needs to be printed verbatim using getMessage().
     - Exceptions of type InputMismatchException must be caught explicitly. If caught, their error messages must be "File contains invalid information.".
     - Exceptions of type NoSuchElementException must be caught explicitly. If caught, their error messages must be "File ended abruptly.".

2. **Play music:** Calls FYEMusicPlayer.play(). Any exceptions thrown by this method call must be caught explicitly, printing the message verbatim using getMessage().
3. **Quit:** Calls FYEMusicPlayer.close() and then terminates the program.

Deliverables

- Assignment6.java (driver)
  - loadMusic()
  - main()
- Note.java
- FYEMusicReaderException.java

Considerations

- Remember that you will get partial credit for partial work. Try to deliver as much of the assignment as you can.
- You can add any helper methods you believe are necessary, but they will not count towards your grade.
- Despite what the deliverables above say, you can submit the Note class and the FYEMusicReaderException class in the same file as your driver class.
- When reading from the Scanner that is reading from the FYEMUS file, you should prefer using only the Scanner's nextByte() method, as that will simplify your code quite a bit: Any numbers greater than 127 will throw exceptions, meaning you only need to check for negative numbers between -1 and -128 inclusive.
  - This also means that you don't need to call the Scanner's hasNextByte(). If the Scanner tries to read something that is not a Byte (a number between -128 and 127, inclusive), it will throw an exception which you are being required to catch.
- You will only need to write one TRY-CATCH block. It should be inside your main()'s menu loop.
- While not a requirement, it's good practice for your program to catch general exceptions at the very end of your TRY-CATCH block. That way, your program can at least try to recover from any exceptions that you forgot to catch.
- There is no need to re-submit FYENote.java; your grader already has that file. There is also no need to submit any of the music files.
- Yes, all 5 songs are present in the zip file in their entirety.
- The assignment doesn't require you to call close() on your note Scanner to simplify the instructions. In the real world, you would always want to call close(), otherwise your program would be leaking file handles.
  - If you are up for it, try to implement a call to close() in the FINALLY block of your main()'s TRY-CATCH-FINALLY.
- Don't worry if your computer is incapable of playing music: you are being graded on writing code that can read FYEMUS files, not on playing them.
- Because of the underlying technology that the FYEMusicPlayer uses, sometimes a note might be played on top of another note, or it might take longer than it should to play.

Playing the song again usually fixes this problem. As long as you convert your timings correctly when you read them from the file, you will not lose points.

- You can open FYEMUS files as regular text files to inspect their contents.

Sample Output (user input in red)

```
[FYE Music Player]
1. Load music
2. Play music
3. Quit
Enter option: 1

Enter name of music file: badCounter.fyemus
File contains invalid information.

1. Load music
2. Play music
3. Quit
Enter option: 1

Enter name of music file: badNote.fyemus
File ended abruptly.

1. Load music
2. Play music
3. Quit
Enter option: 1

Enter name of music file: empty.fyemus
File ended abruptly.

1. Load music
2. Play music
3. Quit
Enter option: 1

Enter name of music file: missingNotes.fyemus
File ended abruptly.
```

```
1. Load music
2. Play music
3. Quit
Enter option: 1

Enter name of music file: negativeCounter.fyemus
Unable to load file: Note counter out of range.

1. Load music
2. Play music
3. Quit
Enter option: 1

Enter name of music file: song1.fyemus
Music loaded.

1. Load music
2. Play music
3. Quit
Enter option: 2

Playing music...
Done playing.

1. Load music
2. Play music
3. Quit
Enter option: 1

Enter name of music file: song3.fyemus
Music loaded.

1. Load music
2. Play music
3. Quit
Enter option: 2

Playing music...
Done playing.
```

```
1. Load music
2. Play music
3. Quit
Enter option: 1

Enter name of music file: my_mix_tape.fyemus
my_mix_tape.fyemus (The system cannot find the file specified)

1. Load music
2. Play music
3. Quit
Enter option: 3

Shutting off...
```