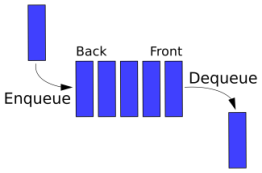
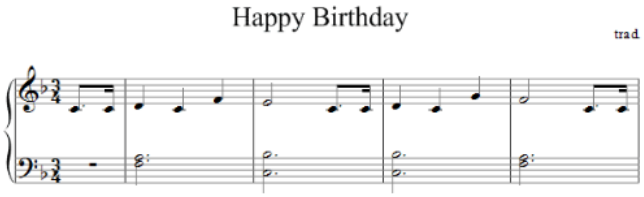
Melody Maker



Music consists of a series of ***notes*** which have ***lengths*** and ***pitches***. The pitch of a note is described with a letter ranging from A to G. Seven notes would not be enough to play very interesting music - luckily there are multiple octaves; after we reach note G we start over at A. Each set of seven notes is considered one octave. Notes may also be ***accidentals*** - this means that they are not in the same key as the music is written in. We normally notate this by calling them sharp, flat or natural. Music also has silences which are called ***rests***.

For this assignment, we will be representing notes using "scientific pitch notation". This style of notation represents each note as a letter and a number specifying the octave it belongs to. For example, middle C is represented as `C4`. You do not need to understand any more than this about scientific pitch notation but you can read more about it here if you are interested: en.wikipedia.org/wiki/Scientific_pitch_notation.

You will write a class that represents a song, where a song is comprised of a series of notes (and may have repeated sections). As we don't like to have any redundancy, we will only store one copy of a repeated chunk of notes.



Music is usually printed like the example above on the right, where the notes are a series of symbols. Their position in relation to the lines determines their pitch and their "tops" and color (among other things) determine their length. Since it would be difficult for us to read input in this style, we will instead read input from a text file.

```
0.2 C 4 NATURAL false
0.4 F 4 NATURAL true
0.2 F 4 NATURAL false
0.4 G 4 NATURAL false
0.2 G 4 NATURAL true
0.2 A 4 NATURAL false
0.4 R false
0.2 C 5 NATURAL false
0.2 A 4 NATURAL false
…
```

An example input file is shown to the left. Each line represents a single note, which is comprised of the following (more info on the Note class to follow):

- The first number describes the length of the note in seconds
- The letter that follows describes the pitch of the note.
  - This will be the standard set of letters (A – G), or R if the note is a rest
  - The different pitches are represented as an enumerated type
- The third item on the line is the octave that the note is in
- Next is the note's accidental value (e.g. sharp (♯) or flat (♭))
  - The accidental values are also enumerated types
- The final value will be true if the note is the start or stop of a repeated section, and false otherwise.

You will write a class, called **Melody.java**, which will allow you to use MelodyMainGUI to play songs with mp3 player like functionality (GUI contributed by Andrew Sen, Liberty student 2019). Use Java's Queue type (and LinkedList objects) from `java.util`. You must use them as queues; you may **NOT** use any index-based methods, iterators or enhanced for loops (`while` and regular `for` loops are fine).

You have been provided with a class named Note that your Melody class will use. As stated before, a Note object represents a single musical note that will form part of a melody. It keeps track of the length (duration) of the note in seconds, the note's pitch (A through G, or R if the note is a rest), the octave, and the accidental (sharp, natural or flat). Note also keeps track of whether it is the first or last note of a repeated section. The Note class API is as follows (this class has been finished for you, but you will interact with its objects):

`getAccidental`
`getDuration`
`getOctave`
`getPitch`
`isRepeat`
	Getter methods, returns the state of the note.

`play`	Plays the note so that it can be heard from the computer speakers.

`setAccidental`
`setDuration`
`setOctave`
`setPitch`
`setRepeat`
	Setter methods, sets aspects of the state of the note based on the given value.

`toString`	Returns a text representation of the Note.

The Note class is dependent on two enumerated types you must create, called **Accidental.java** and **Pitch.java**. Pitch will have eight values (A through G, and R if the note is a rest). Accidental will have three values (sharp, natural and flat).

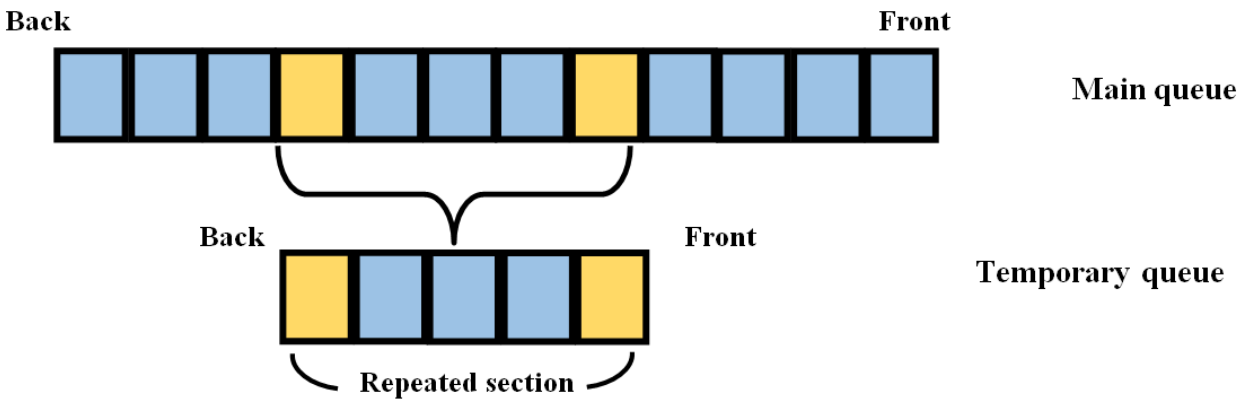
You will be writing the Melody class (the below is just a description of the members, read on for instructions).

`Queue<Note> notes`	Instance variable, the queue containing the notes of this melody
`Melody(Queue<Note> song)`	Initializes your melody to store the notes supplied in the queue of notes parameter. Don't forget to instantiate `notes` prior to adding to it (if iterating through the notes in `song`).
`double getTotalDuration()`	Returns the total length of the song in seconds. If the song includes a repeated section, the length should include that repeated section twice.
`String toString()`	Returns a String containing info. about each note. Each note should be on its own line, using its `toString` method.
`void changeTempo(double tempo)`	Changes the tempo of each note to be `tempo` percent of what it formerly was. Supplying a `tempo` of `1.0` will make the tempo stay the same. `tempo` of `2.0` will make each note twice as long. `tempo` of `0.5` will make each note half as long.
`void reverse()`	Reverses the order of notes in the song, so that future calls to the `play` methods will play the notes in the opposite of the order they were in before `reverse` was called. For example, a song with notes `A, F, G, B` would become `B, G, F, A`.
`Queue<Note> getNotes()`	Getter method to return the notes of this Melody (for the GUI)

`void append(Melody other)`	Adds all notes from `other` to the end of this song. For example, if this song is `A, F, G, B` and the `other` song is `F, C, D`, your method should change this song to be `A, F, G, B, F, C, D` (`other` should be unchanged after the call).
`void play()`	Plays the song by calling each note's `play` method. The notes should be played from beginning to end, unless there are notes that are the beginning or end of a *repeated section*.

When the first note that is a beginning or end of a repeated section is found, you should create a second queue. You should then get notes from the original queue until you see another marked as being the beginning or end of a repeat. As you get these notes you should play them and then place them back in both queues.

Once you hit a second marked as beginning or end of a repeat you should play everything in your secondary queue and then return to playing from the main queue. It should be possible to call this method multiple times and get the same result. The graphic below may help:



The yellow blocks represent notes with start or end of a repeat set to true. They and the other notes in between them should be moved to a separate queue when played so that they can be repeated.

Implement the labs as follows:

1. Import the necessary files (starter code and sample songs).

2. Create the Melody class and declare every method. Leave every method's body blank; if necessary, return a "dummy" value like `null` or `0`. Get your code to run in the MelodyMainGUI program, though the output will be incorrect.

   a. The MelodyMainGUI class represents the user interface. When you run its `main` method, a menu offers you options for loading songs (from text files, by specifying the song's file name), playing, reversing, etc. You shouldn't need to view its code to use it.

3. Complete the Melody class' constructor, and the `toString` method.

4. Implement the `getTotalDuration` and `changeTempo` methods. You can check the results of the `changeTempo` method by loading one of the sample files, calling `changeTempo` and then calling the `toString` method and checking that your output matches what you expected.

    a. The **"tetris.txt"** file should have a duration of 15.5 seconds. Changing its tempo by 2.0 should make its duration 31.0 seconds. **Note:** you must hit enter when making a tempo change to register the change with the GUI.

5. Write the `reverse` and `append` methods.

    a. You may use one additional data structure (implicit or explicit) to complete the `reverse` method.

6. Write an initial version of `play` that assumes there are no repeating sections.

    a. You may use additional queues to complete this method. After the `play` method is done, the `notes` queue should be in its original condition, such that another call of `play` would play the melody again.

7. Add the `play` code that looks for repeated sections and plays them twice, as described previously.

8. Thoroughly test your program by running it on various inputs (sample songs) using the MelodyMainGUI client. For example:

    a. You should be able to play a song, change the tempo, and then play it again with the new speed.

    b. Calling the `reverse` method then the `play` method should play the song backwards.

    c. Calling the `append` method with the same file should play the song twice.

    d. The song in "**twinkle.txt**" has a repeated section, run this song and make sure it works.

        i. The duration of **"twinkle.txt"** should be 24.5s when adding the repeated section.

(Optional) Make your own song!

Create a file called `my_song.txt` that contains a song that can be used as input (look at the sample songs for the required format). You can either invent a song, transcribe a song written by someone else, or generate notes randomly. Writing a program for this, rather than hard-coding it, would be a good idea.