Melody Player

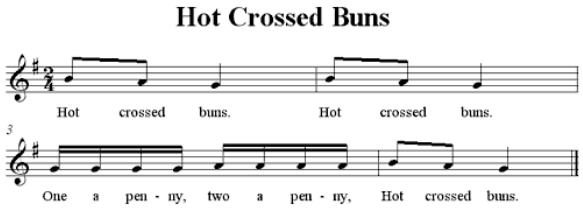
Overview

This program focuses on using the `Stack` and `Queue` data structures as you build a class that allows music to be played and modified.

Background Information about Music:

Music consists of notes which have **lengths** and **pitches**. The pitch of a note is described with a letter ranging from A to G. As 7 notes would not be enough to play very interesting music, there are multiple octaves; after we reach note G we start over at A. Each set of 7 notes is considered an **octave**. Notes may also be **accidentals** which means that they are not in the same key as the music is written in. We normally notate accidentals by calling them *sharp*, *flat* or *natural*. Music also has silences which are called **rests**.

For this assignment you 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: http://en.wikipedia.org/wiki/Scientific_pitch_notation.



What you will do:

You will create a class that represents a song. A song is comprised of a series of notes. It 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 on the right. The notes are a series of dots. 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 at the left. Each line represents a single note. The first number describes the length of the note in seconds. The letter that follows describes the pitch of the note, using the standard set of letters (A – G) or R if the note is a rest. For notes other than rests, the third item on the line is the octave that the note is in and the following is the note's accidental value. The final piece of information for all notes is true if the note is the start or stop of a repeated section and false otherwise.

You will implement several methods in the `Melody` class which will allow you to use the `MelodyMain` class to play your song with mp3 player-like functionality. Your melody will be able to play as well as append another melody to itself, reverse and have the it's tempo changed.

The most challenging part of this assignment is getting melodies to play with repeats correctly. The file above, which contains 4 repeated notes, is equivalent to the repetitive file displayed to the right. When you play the above file you should play it the same as you would play the file to the right. We highlight the repeated section in red.

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

Implementation Details:

You will write one class: `Melody.java`. You must use your `Stack` and `Queue` classes developed in Labs 6 & 7. Your classes must have the constructors/methods below. It must be possible to call the methods multiple times in any order and get the correct results each time. Your `Melody` class will use a queue to store the notes in the song. But unless otherwise specified, you may not create any other auxiliary data structures (such as arrays or lists) to help you solve any method below.

`Note Class` *(provided by the instructor):*

We have provided you with a class named `Note` that your `Melody` class will use. 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-G, or R if the note is a rest), the octave, and the accidental (sharp, natural or flat). Each `Note` object also uses keeps track of whether it is the first or last note of a repeated section of the melody.

The `Note` class provides the following constructors and methods that you should use in your program.

Method	Description
`new Note(`**duration**`,` **pitch**`,` **octave**`,` **accidental**`,` **repeat**`)`	Constructs a new `Note` object.
`new Note(`**duration, pitch**`,`**repeat**`)`	Constructs a new `Note` object, omitting the octave and accidental values. Used to construct a rest (`Pitch.R`).
`getAccidental(), getDuration(), getOctave(), getPitch(), isRepeat()`	Getters that return the state of the note as passed to the constructor.
`play()`	Plays the note so that it can be heard from the computer speakers.
`setAccidental(`**accidental**`), setDuration(`**duration**`), setOctave(`**octave**`), setPitch(`**pitch**`), setRepeat(`**repeat**`)`	Setters of the aspects of the state of the note based on the given value.
`toString()`	Returns a text representation of the note.

You can look at the contents of the provided `Note.java` to answer any further questions about how it works.

`Melody.java` *(for you to implement)*:

`public` **`Melody`**`(Queue<Note> song)`
Constructor that initializes your melody to store the passed in `Queue` of `Notes`.

`public 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. For example, both sample files shown on page 1 have length 3.6 seconds.

`public String` **`toString`**`()`
Returns a String containing information about each note. Each note should be on its own line and output using the `toString()` method from `Note`. An example file is displayed on page 1. Your `String` should reflect the changes that have been made to the song by calling other methods.

`public void` **`changeTempo`**`(double tempo)`

Changes the tempo of each note to be `tempo` percent of what it formerly was. Passing 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. Keep in mind that when the tempo changes the length of the song also changes.

`public void` **`reverse`**`()`

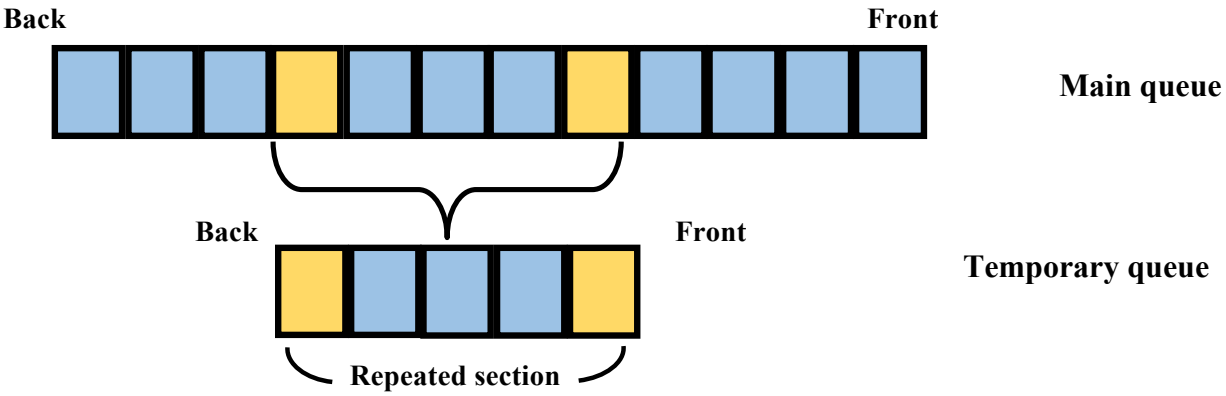
Reverses the order of notes in the song, so that future calls to the `play()` method will play the notes in the opposite of the order they were in before `reverse()` was called. For example, a song containing notes `A, F, G,` then `B` would become `B, G, F, A`. You may use one local (temporary) `Stack` **or** one local (temporary) `Queue` to help you solve this problem.

`public void` **`append`**`(Melody other)`

Adds all notes from the given other song 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`. The other song should be unchanged after the call. Remember that objects can access the private fields of other objects of the same type.

`public void` **`play`**`()`

Plays the song by calling each note's play method. The notes should be played from the beginning of the queue to the end unless there are notes that are marked as being 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 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.

Creative Aspect (`song.txt`):

Along with your program, submit a file called `song.txt` that contains a song that can be used as input. For full credit, the file should be in the format described above and contain at least 10 notes. It should also be your own work (you may not just turn in one of our sample songs) but you do not have to compose a song yourself. You are welcome to make `song.txt` be a song written by somebody else, such as a lullaby or nursery rhyme or song by your favorite band. This will be worth a small portion of your grade. (It **does not** have to be melodic! Have fun with this!)

Development Strategy and Hints:

We suggest the following development strategy for solving this program:

1. Create the `Melody` class and declare every method. Leave every method's body blank (stub); if necessary, return a "dummy" value like `null` or `0`. Get your code to run in the `MelodyMain` program, though the output will currently produce no sound.
2. Implement the constructor, and the `toString` method. At this point you should be able to print the song.
3. Implement the `changeTempo()` and `getTotalDuration()` methods. You can check the results of the `changeTempo()` method by reading in one of the sample files, calling `changeTempo()` and then calling the `toString()` method and checking your output matches what you expected. It may be easiest to initially write `getTotalDuration()` with a song that has no repeated sections. Once that works correctly, add the logic that determines the duration of a song with repeating sections.
4. Write the `reverse()` and `append()` methods.
5. Write an initial version of `play()` that assumes there are no repeating sections.
6. Add the `play()` code that looks for repeated sections and plays them twice, as described previously.

You can test the output of your program by running it on various songs that are provided in the `MelodyMain` client.

Stacks and Queues

You will use the implementation of the `Stack` and `Queue` classes you created in Labs 6 & 7. You may use any implementation of the `Stack` and `Queue` interface your wish.

Style Guidelines and Grading:

Redundancy is always a major grading focus; avoid redundancy and repeated logic as much as possible in your code.

Properly encapsulate your objects by making fields `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used within a single method. Initialize fields in constructors only.

Follow good general style guidelines such as: appropriately using control structures like loops and `if/else` statements; avoiding redundancy using techniques such as methods, loops, and `if/else` factoring; properly using indentation, good variable names, and proper types.

Comment descriptively at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, and any exceptions thrown. Write descriptive comments that explain error cases, and details of the behavior that would be important to the client. Your comments should be written in your own words and not taken verbatim from this document.