COSC102 Spring 2015

DS 2: Music Player Homework 6

Due date: **Thursday, April 9th at 5:00 p.m. on Moodle**

The goals of this assignment are

- to gain experience using stacks and queues, using their traditional operations that are fast,
- to expose you to audio concepts and
- to improve your class design skills.

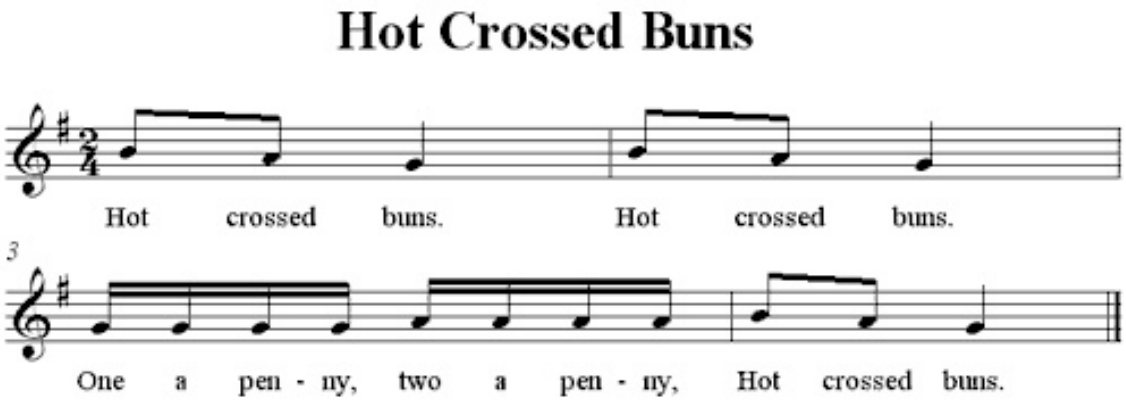
This program focuses on using two specific implementations for `Stack` and `Queue`: `VectorStack` and `LinkedQueue` from our textbook. The provided files include the textbook code we studied in class: `StackInterface.java`, `QueueInterface.java`, `VectorStack.java` and `LinkedQueue.java`.

Problem Description

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. 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 feel free to read more about it here if you are interested.

You will write a class `Melody.java` to represent 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 above. 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.

An example input file is shown below. Each line in it 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.

```
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
…
```

```
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
…
```

You will implement several methods in the `Melody` class which will allow you to use `MelodyMain` 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 its 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.

Setup

From the course website download the `zip` file which contains

- `MelodyMain.java`
- `Note.java`, `Pitch.java`, `Accidental.java` and
- `StdAudio.java`

in addition to the textbook queue and stack implementations which you need to use in your `Melody.java`.

Implementation Requirements

You will write one class, `Melody.java`, and update `MelodyMain.java` to use it. You must use the **provided `Stack` and `Queue`** from the textbook. (Don't use the Java API ones from `java.util`.) You may NOT use any index based methods, iterators or for-each loops: there aren't available in the provided textbook implementations.

Your classes must have the constructors/methods described next. 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 one queue to store the notes in the song. Unless otherwise specified, you may not create any other auxiliary data structures (such as arrays, lists, stacks, queues) to help you solve the methods below.

Do not make unnecessary or redundant passes over a queue when the answer could be computed with fewer passes. Part of your grade will come from appropriately using stacks and queues.

Note.java

The provided class `Note` will be used by your `Melody` class. 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 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.

Methods	**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()`	Returns 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(acc)`, `setDuration(dur)`, `setOctave(oct)`, `setPitch(ptch)`, `setRepeat(rep)`	Sets aspects of the state of the note based on the given value.
`toString()`	Returns a text representation of the note.

Look at the content of the provided `Note.java` to answer any further questions about how it works.

Melody.java

The class you implement needs the following methods.

`public Melody(QueueInterface<Note> song)`

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 above have length 3.6. You should not loop through the notes every time this method is called.

`public String toString()`

Returns a `String` containing information about each note. Each note should be on its own line and output using its `toString` method. The output should look similar to the input file presentation. **But `toString` 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
- a `tempo` of 2.0 will make each note twice as long
- a `tempo` of 0.5 will make each note half as long
- and so on

Keep in mind that when the tempo changes the length of the song also changes.

`public void reverse()`

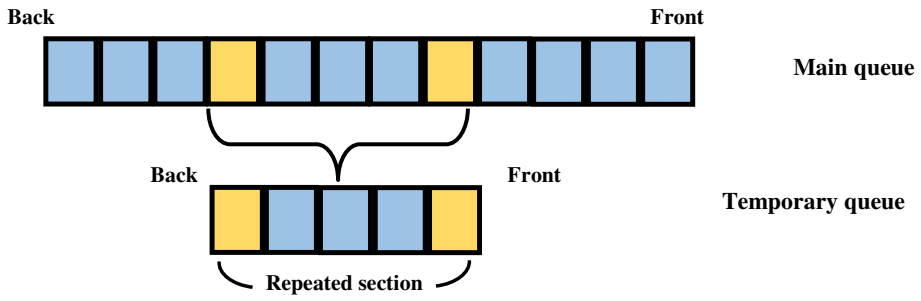
Reverses the order of notes in the song, so that future calls to the `play` methods will play the notes in the opposite of the order they were in before `reverse` was called. For example, a song containing notes `A, F, G`, then `B` would become `B, G, F, A`. You may use one temporary stack **or** one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

`public void append(Melody other)`

Adds all notes from the passed `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

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 the **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: give credits in your `readme` or write if the song is your own composition. This will be worth a small portion of your grade.

Development Strategy and Hints

We suggest the following development strategy for solving this program. You might want to implement your code in two passes. First write the code to handle a song that has no repeat sections. Second adjust your code to consider songs that have repeated sections. You should test for multiple repeat sections.

0. Run the `main` of `StdAudio.java`. Can you hear the music? If not see us.
1. 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 it to run by updating `MelodyMain.java`, though the output will be not implemented yet or incorrect.
2. Implement the constructor in `Melody`. Read/study `MelodyMain.java` helper methods are provided that you should use.
3. Implement the `toString` method. Remember the restrictions: no iterator, for each, **no local** queue or stack are needed to implement this functionality.
4. Implement the `getTotalDuration` and `changeTempo` methods. You can check the results of the `changeTempo` method by reading in one of the sample files, calling `changeTempo` and then calling `toString` and checking your output matches what you expected.
5. Write the `reverse` and append‘ methods.
6. Write an initial version of `play()` that assumes there are no repeating sections.
7. 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 inputs using the input console provided by the `MelodyMain` client.

About style

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; and not having any lines of code longer than 100 characters.

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, pre/post-conditions, 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.

Submit

Turn in a `.zip` file containing `Melody.java`, `MelodyMain.java`, your `song.txt` and a `readme`, which includes

- your student information at the top,
- description and/or credit of how your `song.txt` and
- if you have not fully implemented the functionality of `Melody`, list the parts that work and the ones you attempted so as to receive partial credits.

Credit This assignment was created by Allison Obourn and Marty Stepp.