CSE 143, Autumn 2013
Programming Assignment #3: Melody (40 points)
Due Thursday, October 17, 2013, 11:30 PM

This program focuses on using `Stack` and `Queue` collections. Turn in files named `Melody.java`, `Note.java` and `song.txt` on the Homework section of the course web site. You will need `StdAudio.java`, `Pitch.java`, `Accidental.java` and `MelodyMain.java` from the web site; place them in the same folder as your program.

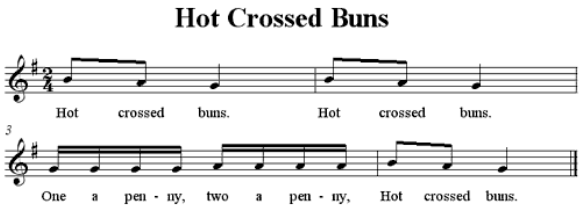
Background Information about Music:

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. 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: http://en.wikipedia.org/wiki/Scientific_pitch_notation.

What you will do:

You will write two classes that allow you 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 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 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.

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 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.

```
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 two classes; `Melody.java` and `Note.java`. You must use Java's `Stack` and `Queue` from `java.util`. You must use them as stacks and queues; you may **NOT** use any index based methods, iterators or for-each loops. 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, lists, stacks, queues) to help you solve any method below.

Note.java

A `Note` object represents a single musical note that will form part of a melody. It should keep track of the length of the note, the note's pitch (A-G) or R if the note is a rest, the octave and the accidental type (sharp, natural or flat). Sometimes, a note is the first or last note of a repeated section of the melody. Each `Note` object should also keep track of whether or not this is the case.

The `Note` class interacts with two provided types named `Pitch` and `Accidental`. A `Pitch` is a constant from `Pitch.A` through `Pitch.G` or `Pitch.R`, partially representing the frequency of the note. An `Accidental` indicates whether a note is sharp, flat, or neither by the constants `Accidental.SHARP`, `Accidental.FLAT`, and `Accidental.NATURAL` respectively.

We have provided a starter file for this class. It contains the method headers and comments for the methods below. You may keep all the code in the starter file.

Method	Description
`public` **`Note`**`(double duration, Pitch note, int octave, Accidental accidental, boolean repeat)`	Takes a number representing the length of the note in seconds, a `Pitch` representing the pitch (eg. `Pitch.A`), a number representing the octave of the note, an `Accidental` representing the accidental (eg. `Accidental.NATURAL`), and true if this is the first or last note in a repeated section. Throw an `IllegalArgumentException` if the duration is less than or equal to 0 or it the octave is less than 0 or greater than 10.
`public` **`Note`**`(double duration, Pitch note, boolean repeat)`	Takes a number representing the length of the note in seconds, a `Pitch` representing the pitch (eg. `Pitch.A`), and true if this is the first or last note in a repeated section. Throw an `IllegalArgumentException` if the duration is less than or equal to 0.
`public double` **`getDuration`**`()`	Returns the length of the note in seconds.
`public void` **`setDuration`**`(double d)`	Sets the duration of the note to be the passed in value. Throw an `IllegalArgumentException` if the duration is less than or equal to 0.
`public boolean` **`isRepeat`**`()`	Returns `true` if this note is either the first or last note in a repeated section of the melody, or `false` otherwise.
`public void` **`play`**`()`	Plays the note by calling the following static method and passing in the appropriate data. `StdAudio.play(double duration, Pitch note, int octave, Accidental accidental)`
`public String` **`toString`**`()`	If the pitch is `Pitch.R` return a `String` in the format, "<duration> <pitch> <repeat>". Otherwise return a `String` in the format, "<duration> <pitch> <octave> <accidental> <repeat>". For example both `"2.3 A 4 SHARP true"` and `"1.5 R true"`. You can use the `toString` of each type to represent it.

Melody.java

`public` **`Melody`**`()`
Initializes your melody to store an empty queue of notes.

`public void` **`input`**`(Scanner input)`
Create a new queue of notes by getting note data from the passed in `Scanner`. Use `Pitch.valueOf(String s)` and `Accidental.valueOf(String s)` to convert the `String`s you read into `Pitch`es and `Accidental`s. You can assume that each line contains only one note and that each is output in the format described in note's `toString` method. An example file is displayed on page 1.

`public double` **`getLength`**`()`
Returns the total length of the song in seconds. You should not loop through the notes every time this method is called.

`public void` **`output`**`(PrintStream out)`
Outputs the song, one note per line, to the `PrintSream`. Each note should be output using its `toString` method. An example file is displayed on page 1. Your output 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.

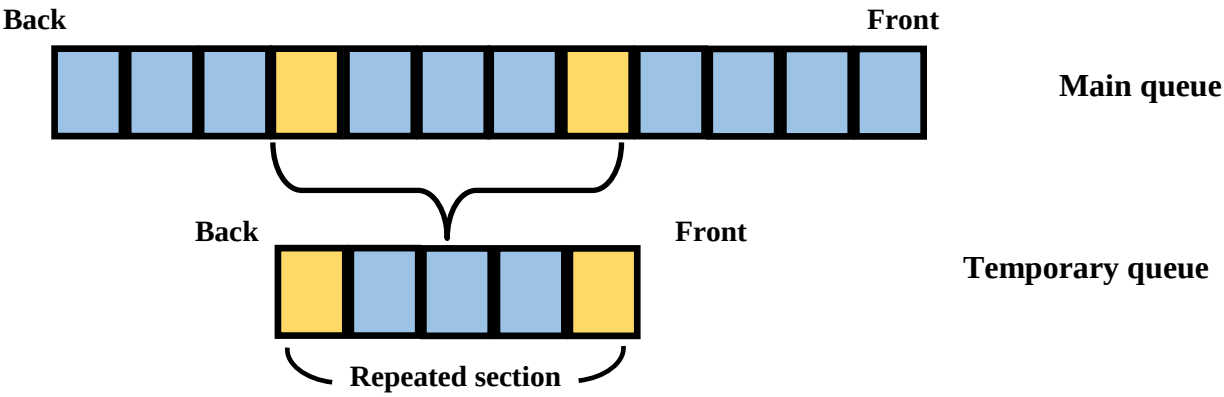
`public void` **`reverse`**`()`

Reverses the order of notes in the song, so that future calls to the `play` 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 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 objects 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.

`public void` **`play`**`(double time)`
Does the same thing as the no parameter `play` method except that it starts playing in a different place. This method should start playing the song at the first note at least `time` away from the start of the melody. For example, if all notes in the melody were 1 second long and `play(3)` was called, the first note played should be the 4th note. If `play(1.8)` was called on the same melody, the third note should be the first one played. If you start on the note in the middle of a repeated section you should ignore the repeat for that section. After your method finishes executing, your melody's queue must be in exactly the same state it was before. Future calls to `play()` should be unaffected.

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.

Development Strategy and Hints:

We suggest the following development strategy for solving this program:

1. Fill out the methods in the `Note` class.
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 `MelodyMain` program, though the output will be incorrect.
3. Implement the constructor, and the `input` and `output` methods. If you are feeling rusty on file I/O you can review it by reading chapter 6 in the textbook.
4. Implement the `getLength` 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 the `output` method 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.
8. Write `play(double time)`. Keep in mind, it is very similar to `play()`.

You can test the output of your program by running it on various inputs in the `MelodyMain` client. You can also use our Output Comparison Tool to see that your outputs match what is expected.

Style Guidelines and Grading:

Part of your grade will come from appropriately using stacks and queues. You may only use the `size`, `isEmpty`, `add`, `remove` and `peek` methods for queues. For stacks, you may only use the `size`, `isEmpty`, `pop`, `push` and `peek` methods. You may not call any `Stack` methods that accept index parameters. You may not examine a stack/queue using a "for each" loop or iterator. Do not make unnecessary or redundant extra passes over a queue when the answer could be computed with fewer passes. Declare queues as variables of type `Queue`, not variables of type `LinkedList` (on the left side of the equals sign).

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.

For reference, our solution is around **170 lines** long (105 "substantive"), though you do not have to match this exactly.