**Melody Maker**

A ***queue*** is an abstract data type (ADT) that you see a lot in real life – lines of cars at stoplights, people waiting in line at the bank, documents waiting to be printed – they all follow this "first in, first out" (FIFO) sequence. In a queue, addition only happens at the "rear", and removal only occurs at the "front". Solve the problems below to learn more about queues.

1. Create a class **Runner.java** with a main method and a class **QueueProbs.java** that will define the methods below. Make a QueueProbs object in the Runner class to test the methods.
2. In Java, Queue is an interface, a description of how something that ***is-a*** queue should behave - Queue can NOT be instantiated (unlike the java.util.Stack type, which *could* be instantiated).
   1. Use the java.util.LinkedList class to instantiate a "queue object"; this class implements Queue (it will behave like a queue when queue methods are utilized).
   2. Queue methods are as follows:
      1. peek() Returns (but does not remove) the value at the head of the queue
      2. poll() Removes ("dequeues") and returns the element at the head of the queue
      3. offer() Adds ("enqueues") an element to the queue (at the tail of the queue)

There are other methods you can use to achieve similar results, but it's best to use a queue with queue-specific methods to avoid confusion.

1. Complete the method Queue<Integer> evenFirst(Queue<Integer> nums) that will move the even numbers to the front of the queue. You may use additional Queues, but no Iterators / for-each loops.

(3, 5, 4, 17, 6, 83, 1, 84, 16, 37) >>> head(4, 6, 84, 16, 3, 5, 17, 83, 1, 37)tail

1. (Riddle) How can you name three consecutive days without using the words Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, or Sunday? (Or day names in any other language.)
2. Write a method that computes all the prime numbers up to some integer n using a technique known as the Sieve of Eratosthenes, which was developed by a Greek (Eratosthenes!) who lived in the third century BC. Wiki it first to watch how it works. Pseudo-code:

*Fill a queue with the consecutive integers 2 through n inclusive.*

*Create a stack that will store the primes.*

*While queue is not empty...*

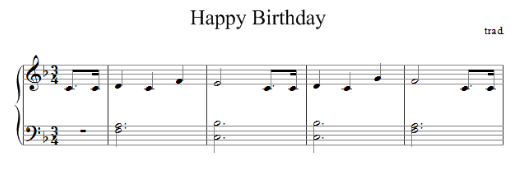
*Add the first value in the queue (P, the next prime) to the stack of primes.*

*Iterate through the queue, eliminating numbers divisible by the P that can't be primes*

**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](http://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.

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):

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

…

* 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
    - See the JavaDocs link in the lab folder for information on enums
    - An enum is a special *type*, like an abstract class or interface
* 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 one class, called **Melody.java**, which will allow you to use MelodyMain to play your song with mp3 player like functionality. 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 for-each loops.

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). 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 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. |

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

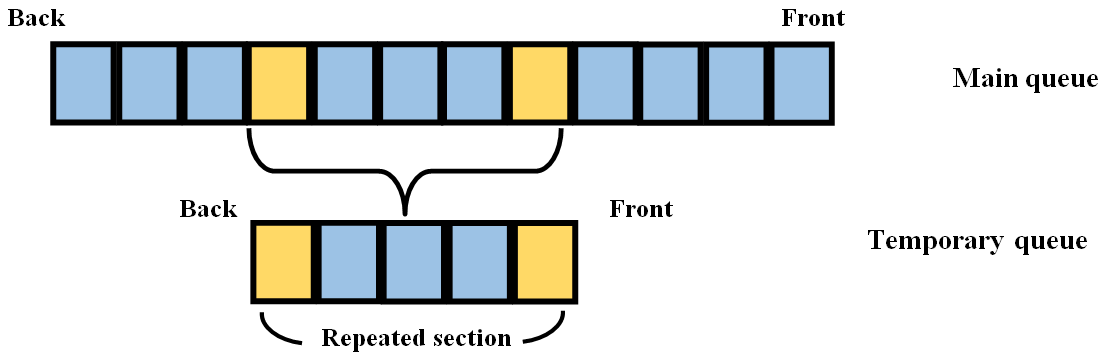
* Queue<Note> notes – queue containing the notes of this melody
* Melody(Queue<Note> song) - initializes your melody to store the notes supplied in the queue of notes parameter. Don't forget to instantiate notes prior to adding to it.

* 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 information about each note. Each note should be on its own line and output 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.
* 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, B would become B, G, F, A. You may use 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. Note that objects can access the private fields of other objects of the same type (though this is not considered "best practice" by many).
* 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 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 MelodyMain program, though the output will be incorrect.
3. The MelodyMain class essentially implements a console-based 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.
4. Complete the Melody class' constructor, and the toString method.

1. 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.
2. 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.
3. The **"twinkle.txt"** file (which includes a repeated section) should have a duration of 24.5 seconds.
4. Write the reverse and append methods.
5. You may use one additional data structure to complete the reverse method.
6. Write an initial version of play that assumes there are no repeating sections.
7. 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.
8. Add the play code that looks for repeated sections and plays them twice, as described previously.
9. Thoroughly test your program by running it on various inputs (sample songs) using the MelodyMain client. For example:
10. You should be able to play a song, change the tempo, and then play it again with the new speed.
11. Calling the reverse method then the play method should play the song backwards.
12. Calling the append method with the same file should play the song twice.
13. The song in "**twinkle.txt**" has a repeated section, run this song and make sure it works.

**(Optional creativity) 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.

**(Advanced) Playing harmonies**

Using a concept called ***multi-threading***, write a program that will play two melodies "simultaneously", producing a harmony. Note that the StdAudio class can save a song as a sound file and can play a sound file of various formats.

*Adapted from the* ***Melody Maker*** *project by Allison Obourn and Marty Stepp*

*nifty.stanford.edu/2015/obourn-stepp-melody-maker/*