Melody Player

(CS1 array version)

The purpose of this assignment is to practice writing algorithms that use 1-D and 2-D arrays.

This problem concerns playing music. A song consists of notes, each of which has a *length* (duration) and *pitch*. 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*, meaning that they are not in the same key in which the music is written. We normally notate this by calling them *sharp*, *flat*, or *natural*. Music also has silences that are called *rests*.

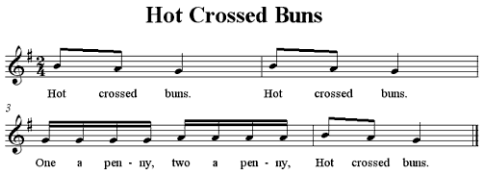
For this program we will be representing notes using scientific pitch notation. This style of notation represents each note as a letter and a number specifying the octave it belongs to. For example, middle C is represented as `C4`. You do not need to understand any more than this about scientific pitch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

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

You will write a `Melody` class that uses an array to represent a song 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. Your `Melody` class will read files in a format described below and represent the song's notes using an array of `Note` objects. The most challenging part of this assignment is handling melodies that contain repeated sections.

Input File Format:

Music is usually printed like the example sheet music at 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 read input from a text file in a specific format.



An example input file is shown at right. The first two lines contain the song **title** and song **artist**, respectively. The third line contains the number of notes in the song; this is equal to the number of lines that will follow in the file. Each subsequent line represents a single note in the following format:

- ***duration pitch octave accidental repeat***

The first number on each line describes the duration of the note in seconds. The next letter describes the pitch of the note, using the standard letters A-G or R for a rest. The third token is the octave that the note is in. The fourth is the note's accidental value of sharp, flat, or natural. *(For a rest, the octave and accidental are omitted.)* The final token indicates whether the note is the start or stop of a repeated section: `true` if so, and `false` otherwise. In the example at right, notes 3-5 and 9-12 (lines 6-8 and 12-15) represent two repeated sections. The meaning of the data is that the song should play notes 1, 2, 3, 4, 5, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 9, 10, 11, 12, 13. Our format does not allow nested repetition, nor sections that repeat more than twice.

Example input file (line numbers added):

```
 1| My Song Title
 2| Joe Smith
 3| 13
 4| 0.2 C 4 NATURAL false
 5| 0.4 F 4 NATURAL false
 6| 0.2 F 4 NATURAL true
 7| 0.4 G 4 NATURAL false
 8| 0.2 A 4 NATURAL true
 9| 0.2 A 4 NATURAL false
10| 0.4 R false
11| 0.2 B 4 NATURAL false
12| 0.2 C 4 NATURAL true
13| 0.4 D 4 NATURAL false
14| 0.2 C 5 NATURAL false
15| 0.2 A 4 NATURAL true
16| 0.4 D 4 NATURAL false
```

Note class *(provided)*:

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 as a `double`, the note's pitch (A-G, or R if the note is a rest), the octave as an `int`, and the accidental (sharp, natural or flat). Each `Note` object also uses a `boolean` field to keep track of whether it is the first or last note of a repeated section of the melody. You pass this information to the `Note`'s constructor when you create a `Note` object.

The `Note` uses two types of constants named `Pitch` and `Accidental`.

- A `Pitch` is a constant from `Pitch.A` through `Pitch.G` or `Pitch.R`, meaning the frequency of the note.
- An `Accidental` indicates whether a note is sharp, flat, or neither using the constants `Accidental.SHARP`, `Accidental.FLAT`, and `Accidental.NATURAL` respectively.

The `Note` class provides the following constructors and methods that you should use in your program. The various `get` and `is` methods are accessors that return the values that you previously passed to the `Note` constructor.

Method	Description
`new Note(`***`duration, pitch, octave, accidental, repeat`***`)`	Constructs a new `Note` object.
`new Note(`***`duration, repeat`***`)`	Constructs a new `Note` object, omitting the pitch, 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(`***`accidental`***`), setDuration(`***`duration`***`), setOctave(`***`octave`***`), setPitch(`***`pitch`***`), setRepeat(`***`repeat`***`)`	Sets 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 Class *(for you to implement)*:

Implement the `Melody` class and its methods listed here and described in detail on the following pages:

- `getTitle`, `getArtist`, `getTotalDuration` *(information about the song's state)*
- `play` *(plays the song on the computer's speakers)*
- `octaveDown`, `octaveUp`, `changeDuration`, `reverse` *(methods that manipulate the song's state)*

You will need several fields to implement all of the required behavior. Your class must use a field that is an **array of `Note` objects** to store the notes in the song. You will need other fields to implement all of the behavior shown below, but you should not create any other data structures (such as arrays or lists) to help you.

Test your `Melody` class by running our instructor-provided class `Main` that allows you to select text files and play them. Each time you click Load and choose a file from the disk, a `Melody` object will be created for that file. Most of the other buttons essentially map to the various methods in your Melody class. When the user clicks Play, your `Melody` object's `play` method is called. When the user clicks Reverse, your `Melody` object's `reverse` method is called. And so on. Not every button directly maps to a `Melody` method; for example, the Stop button tells the underlying audio system to halt, but this has the effect of making it temporarily ignore any further notes that your `Melody` code tries to play.

`public Melody(String filename)`

In this constructor you should populate your melody's array of notes by reading note data from the specified file. The file format was described previously; it begins with the song title, author, and number of notes, followed by a series of lines, each of which describes a single `Note` object. You should tokenize the contents of each note line, use those tokens to construct a `Note` object, and put this `Note` into your internal array.

To convert a string into a `Pitch` or `Accidental` constant value, like turning the string `"SHARP"` into the equivalent constant of `Accidental.SHARP`, use the `valueOf` method, as shown in the following example:

```
String s = input.next();                    // "SHARP"
Accidental acc = Accidental.valueOf(s);     // Accidental.SHARP
```

The constructor is the only part of your code that should read data from the input file. All other methods should refer to your internal array of notes. Re-reading the file to implement other methods is forbidden.

Assume valid input. You may assume that the file exists, is readable, and that its contents exactly follow the format described on the previous page. The file contains exactly the number of note lines equal to the number written on the file's third line. You may assume that each note line contains a valid note, such as `"1.2 C 4 NATURAL false"` for a 1.2-second natural C note in the 4th octave, or `"0.4 R false"` for a 0.4-second rest. No line will contain a note other than A-G or R (rest); none will contain any badly formatted tokens or lines; etc.

`public String getTitle()`

In this method you should return the title of the song, as was found in the first line of the song's input file.

`public String getArtist()`

In this method you should return the artist of the song, as was found in the second line of the song's input file.

`public double getTotalDuration()`

In this method you should return the total duration (length) of the song, in seconds. In general this is equal to the sum of the durations of the song's notes, but if some sections of the song are repeated, those parts count twice toward the total. For example, a song whose notes' durations add up to 6 seconds that has a 1.5-second repeated section and a 1-second repeated section has a total duration of (6.0 + 1.5 + 1.0) = 8.5 seconds.

`public void play()`

In this method you should play your melody so that it can be heard on the computer's speakers. Essentially this consists of calling the `play` method on each `Note` in your array. The notes should be played from the beginning of the list to the end, unless there are notes that are marked as being part of a repeated section. If a series of notes represents a repeated section, that sequence is played twice. For example, in the diagram below, suppose the notes at indexes 3, 5, 9, and 12 all indicate that they are start/end points of repeated sections (their `isRepeat` method returns `true`). In this case, the correct sequence of note indexes to play is 0, 1, 2, 3, 4, 5, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 9, 10, 11, 12, 13. Note that notes at indexes 3-5 and 9-12 are played twice in our example.

This method should not modify the state of your array. Also, it should be possible to call `play` multiple times and get the same result each time.

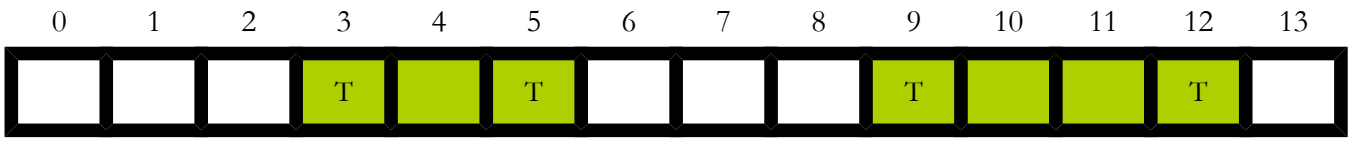


diagram of array of notes with repeated sections at indexes 3-5 and 9-12

`public boolean octaveDown()`

In this method you should modify the state of the notes in your internal array so that they are all exactly 1 octave lower in pitch than their current state. For example, a C note in octave 4 would become a C note in octave 3. Rests are not affected by this method, and the notes' state is otherwise unchanged other than the octaves.

There is one special case to watch out for. Octave 1 is the lowest possible octave allowed by our system. If any note(s) in your song are already down at octave 1, then the entire `octaveDown` call should do nothing. In such a case, no notes (even ones above octave 1) should be changed; the call should have no effect.

You should return `true` if this method lowered the octave, and `false` if you hit the above special case.

`public boolean octaveUp()`

In this method you should modify the state of the notes in your internal array so that they are all exactly 1 octave higher in pitch than their current state. For example, a C note in octave 4 would become a C note in octave 5. Rests are not affected by this method, and the notes' state is otherwise unchanged other than the octaves.

There is one special case to watch out for. Octave 10 is the highest possible octave allowed by our system. If any note(s) in your song are already up at octave 10, then the entire `octaveUp` call should do nothing. In such a case, no notes (even ones below octave 10) should be changed; the call should have no effect.

You should return `true` if this method raised the octave, and `false` if you hit the above special case.

`public void changeTempo(double ratio)`

In this method you should scale the duration of each note in your melody by the given ratio. For example, passing a `ratio` of `1.0` will do nothing, while a `ratio` of `2.0` will make each note's duration twice as long (slow down the song), or a `ratio` of `0.5` will make each note half as long (speed up the song).

`public void reverse()`

In this method you should reverse the order of the notes in your melody, so that future calls to `play` would play the notes in the opposite of the order they were in before the call. For example, a song containing notes `A, F, G, B` would become `B, G, F, A`. This amounts to reversing the order of the elements of your internal array of notes. Do not make a complete copy of your internal array, and do not create any other data structures such as arrays, strings, or lists; just modify your array in-place. You must write the reversal code yourself; you should not use an existing Java array reversal library.

`public String toString()` *(optional)*

You are not required to write a `toString` method in your `Melody` class, but if you do, it will be called by our `Main` program when any operations are performed. For example, after loading a song from a file, or reversing the song, or changing duration or octaves, the `Main` program prints out the `toString` representation of your `Melody` on the console. If you do write a `toString`, you can return any string you want. This may be useful for debugging. Recall that `Arrays.toString` returns a string representation of an array.

Creative Aspect, mysong.txt:

Along with your `Melody` class, turn in a file named **mysong.txt** that contains song data that you have made yourself. This file can have any contents you want, so long as it follows our specified song file format with the title on the first line, the song's artist or creator on the second line, the number of notes on the third line, and a valid note on each line after that. You can make up a song of your own, or you can make a version of an existing song that you like (Mario theme? Lady Gaga song? The Star Spangled Banner? etc.).

Possible Extra Features:

There are many possibilities for extra features that you can add if you like. Here are a few ideas:

- *Change key:* Can you change a melody from one key to another? This is harder than it sounds because of complexities in music and keying. Read online about different keys and how to convert between them.
- *Start playing from a given time offset:* Write a variation of the `play` method that accepts a start time offset as a parameter and plays the melody's notes starting from that time offset.
- *Merge two melodies:* Write a method that accepts another `Melody` as a parameter and appends its notes to the end of the current `Melody`.
- *Additional melody manipulation:* Write more methods that modify your melody's song.
- *Lyrics:* How would you represent a song that had lyrics that display when certain notes are played?
- *Playlist:* Write another class representing a list of melodies to be played in order.
- *Ability to visually compose a song:* Provide a user interface where the user can click piano keys to input a song.

Suggested Style Guidelines

Use descriptive **names** for variables and methods. **Format** your code using indentation and whitespace. Avoid **redundancy** using methods, loops, and factoring. Use descriptive **comments**, including the top of each .java file, atop each method, inline on complex sections of code, and next to each field.

Try to minimize the number of fields and generally not make a value a field unless it is necessary to do so. Write a brief **comment** on each field in your code explaining its purpose and why it is necessary to make that into a field. All fields must be private. If there are important fixed values used in your code, declare them as final **constants**.

Work to avoid **redundancy**. For example, if two or more specified methods have similar behavior, make one call the other, or create a private method that captures the redundancy and is called by both.