ASSIGNMENT 5

COMP-202, Fall 2014, All Sections

Due: December 4thth, 2014 (23:59)

Please read the entire pdf before starting. You must do this assignment individually and, unless otherwise specified, you must follow all the general instructions and regulations for assignments. Graders have the discretion to deduct up to 20% of the value of this assignment for deviations from the general instructions and regulations. These regulations are posted on the course website. Be sure to read them before starting.

Question 1:	20 points
Question 2:	20 points
Question 3:	20 points
Question 4:	24 points
Question 5:	16 points
	100 points total

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment through automated tests. While these tests will not determine your entire grade, it will speed up the process significantly, which will allow the TAs to provide better feedback and not waste time on administrative details. Plus, if the TA is in a good mood while he or she is grading, then that increases the chance of them giving out partial marks. Marks can be removed if comments are missing, if the code is not well structured, or if your solution does not follow the assignment specifications. **Start early, you won't be able to finish this assignment on the last day**.

Assignment

Part 1 (0 points): Warm-up

Exercise 1: Write a program that *opens* a **.txt**, *reads* the contents of the file line by line, and *prints* the content of each line. To do this, you should use look up how to use the `BufferedReader` class[1]. Remember to use the `try` and `catch` statements to handle errors like trying to open an non-existent file.

Exercise 2: Modify the previous program so that it stores every line in an `ArrayList` of `String` objects. You have to properly declare an `ArrayList` to store the results, and use `add` to store every line that your program reads in the `ArrayList`.

Exercise 3: Finally, modify your program so that, after reading all the content in the file, it prints how many words are inside the text file. To do this, you should use the `split` method of the `String` class. Assume the only character that separates words is whitespace `" "`.

[1]The documentation of the `BufferedReadder` class is available at `http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html`. You can find an example on how to use it at `http://www.tutorialspoint.com/java/io/bufferedreader_readline.htm`

Part 2:

Question 1: Implementing a Doubly LinkedList (20 points)

For this question, you will implement a **doubly linked list**[2]. To do this we provide you the **DoublyLinkedList.java** and the **StringNode.java** files. First, modify the `StringNode` class so that it contains a reference to both the previous and next elements in the list. Then, you should complete the `DoublyLinkedList` class by implementing the following methods:

1. `find`: This method receives a String value as input, and returns the first `Node` that matches the value. It returns `null` otherwise.
2. `insertStart`: This method receives a String value, and insert a `Node` with the given value at the beginning of the list.
3. `insertEnd`: This method receives a String value, and inserts a `Node` with the given value at the end of the list.
4. `remove`: This method receives a String value, and removes *all* occurrences of the value in the list.
5. `removeAtIndex`: This method receives an `int` index, and removes from the list the element at the position given by the index.
6. `toStringReverse`: This method returns a string containing all the elements in the list in *reverse* order.

To test if your code works, use the `main` method in the **DoublyLinkedList.java** file.

Question 2: Playing music from notestrings (20 points)

In this part, you will write code to make your computer play simple melodies from a note string specification. Your task is to **transform** that given `string` into **instances** of notes. These instances will be used to produce sounds out of your computer speakers using the provided interpreter. You will need the following files for this question: **MusicInterpreter.java**, **MidiTrack.java** and **MidiNote.java**.

A *notestring*[3] is a sequence of characters which encodes the order and timing of musical notes in a melody. This is an example of a notestring: "`<<3E3P2E2GP2EPDP8C<8B>`". The letters `'A'`, `'B'`, `'C'`, `'D'`, `'E'`, `'F'`, `'G'` correspond to the notes on a musical scale, each one having a corresponding *pitch*. The letter `'P'` represents a pause, or the absence of a note. The numbers represent the duration, measured in *beats*, of the note or pause they precede. The symbols `'>'` and `'<'` will change how high or low a particular note will sound like.

Here's an overview of the purpose of each file (also detailed in Figure 2, at the end of this document). The `MusicInterpreter` class takes care of all the sound generation. It uses a *synthesizer* to generate sounds, and a *sequencer* to determine the order and timing of sounds. You don't have to worry about implementing this, it is already implemented for you. `MidiTrack` class stores all the information from a notestring: it has an `instrumentId`, to determine which instrument sound should be used, and a list of notes, implemented as an `ArrayList` of `MidiNote` objects. A `MidiNote` object stores two properties of a single note: its `pitch`, its `duration`.

You have to implement the `loadNoteString` method of the **MidiTrack.java** file. This method receives a String variable containing a notestring as input. Your method should process the notestring character by character, creating `MidiNote` objects with the appropriate pitch and duration. The items a. to e. describe this process in more detail.

In order to test your notestring parsing, use the `loadSingleTrack` and the `play` methods of the `MusicInterpreter` class, in a similar way to the following code sample. *Hint*: place this code in a `main` method and try running an instance of **MidiTrack.java** file.

[2] `http://en.wikipedia.org/wiki/Doubly_linked_list`

[3] This is inspired by `http://www.danielzingaro.com/sound_proc/assignment.html`.

```
// Build the MidiTrack object
String notestring = "3C>3C<<3A>3A<3A#>3A#18P3C>3C<<3A>3A<3A#>3A#18P";
int instrumentId = 0;
MidiTrack newTrack = new MidiTrack(instrumentId);
newTrack.loadNoteString(notestring);

// Build a MusicInterpreter and set a playing speed
MusicInterpreter mi = new MusicInterpreter();
mi.setBPM(1200);

// Load the track and play it
mi.loadSingleTrack(newTrack);
mi.play();
// close the player so that your program terminates
mi.close();
```

Make sure you **test** each item before moving to the next one. *Hint:* To process a notestring character by character you need a *loop* and *conditional statements* for each of the following cases.

(a) **Creating MidiNote objects**: When processing a notestring, you must create a new instance of `MidiNote` class each time you find any of the following characters `'A'`, `'B'`, `'C'`, `'D'`, `'E'`, `'F'`, `'G'`. Every `MidiNote` that you create should be added to the `notes` attribute of the `MidiTrack` class. Since `notes` is an ArrayList, use the `add` method to do this.

The constructor of a `MidiNote` object requires a pitch value, and a duration. To determine the *pitch* value, you should use the `noteToPitch` Hashtable of the `MidiTrack` class. For example, calling `noteToPitch.get('C')` will return the value of `60`. By default, the *duration* should be 1, corresponding to one beat.

For instance, if you pass the notestring `''CDEFGAB''` to the `loadNoteString` method, your code should add 7 `MidiNote` objects to the `notes` ArrayList of the `MidiTrack` object, with pitches corresponding to 60, 62, 64, 65, 67, 69, and 71; and all of them with a duration of 1 beat.

(b) **Pauses between notes**:

Your code must support reading the character `'P'` as a silence instead of playing a note. For example, playing the `MidiTrack` corresponding to the notestring `''CCCC''` should sound as the same note played 4 times, while `''CPCPCPCP''` should sound as the same note played 4 times, but with a pause after each note. The *duration* of a pause is 1 beat. Use the `setSilent` method of the `MidiNote` class to mark a note as a pause, and add this silent `MidiNote` each time you encounter a `'P'`.

As an example, if you pass the notestring `''CPC''` to the `loadNoteString` method, your code should add 3 `MidiNote` objects to the `notes` ArrayList of the `MidiTrack` object all of them with a duration of 1 beat. The first and third `MidiNote` will have `pitch` equal to 60, and the `silent` attribute equal to `false`. The second one can have any `pitch` value, as long as its `silent` attribute is equal to `true`.

(c) **Note durations**:

This reader supports notes of different durations. When a number N appears before a note or pause, then it means that the note or pause should have a duration of N beats. For example, the following `''CCCC''` should sound like 4 separate notes, where are `''4C''` should sound like a single longer note. On the other hand `''C2C4C8C''` should sound like 4 notes of increasing duration. You code should support a duration N that spans over multiple digits, *e.g.* `''18C''` should sound like a note that last for 18 beats.

(d) **Octaves**:

In the notestring notation, the character `'>'` stands for *increasing* the pitch of **all subsequent notes** by 1 octave, while the `'<'` character stands for *decreasing* the pitch of **all subsequent notes** by 1 octave. Your code must support reading these two characters.

When you increase/decrease the pitch of a note by one octave, it *sounds like the same note* but with a higher/lower pitch. With the `MidiNote` class, going an octave up corresponds to increasing the pitch by 12, while going an octave down corresponds to decreasing the pitch by 12. Thus, every time the reader encounters the `'>'` character, you should *add* 12 to the `pitch` of all subsequent notes. Conversely, when your code processes a `'<'` character, you should *subtract* 12 from the `pitch` of all subsequent notes.

For example, in the notestring `''CDE>CDE<CDE''` the first three notes will be the same as the last three, but the middle three will have a higher pitch. The notestring `''4E>4E>4E>4E<<<4E''`, will sound like 4 E notes with increasing pitch, and a fifth E note with the same pitch as the first one. Take a look at at `http://newt.phys.unsw.edu.au/jw/notes.html` to see the pitch numbers of notes at different octaves.

(e) **Flat and Sharp notes**:

The final requirement are the *sharp* modifier, which increases the pitch by 1, and *flat* modifier, which decreases the pitch by 1. We will use the `'#'` symbol for sharp notes and the `'!'` symbol for flat notes. This will work differently from the octave change. The flat and sharp symbols apply only to the single note that is immediately before the flat/sharp symbol. For example, `''C#''` is a C sharp note, `''B!"` is a B flat note and `''FF#''` corresponds to a regular F note followed by an F sharp. The pitch values of the previous examples are 61, 70, 65 and 66, respectively.

Visit this webpage `http://cs.mcgill.ca/~cs202/2014-09/web/a5/notestrings.html` for a list of example notestring and sample sounds, to compare the result from your code.

Question 3: Loading multiple notestrings from a text file. (20 points)

For this part, you will need the file **Song.java**, in addition to the files from Question 3. We provided you some text files inside the *data* folder to test your code. An object of the `Song` class contains information about the *speed* at which the song will be played, the *instrument* sounds that will be available for the `MusicInterpreter` synthesizer, and a list of *tracks* to be played.

The speed of the `Song` is stored in its `myBeatsPerMinute` attribute. The attribute `mySoundbank`, defines a location of a *soundbank* file, which contains a collection sounds for the synthesizer. The attribute `myTracks` is an ArrayList of `MidiTrack` objects. This will allow us to play polyphonic tunes, by playing multiple notestrings at the same time.

You will write code to open a *Song* file similar to the one shown in Figure 1, load its contents to create an object of `Song` class, and play it using the `MusicInterpreter` class. The following items describe what you need to do in more detail.

(a) **Opening a Song file**:

In the **Song.java** file, implement the `loadSongFromFile` method. This method receives a *file path* as an input, which corresponds to the location of a *Song file*. Each line in a Song file consists of a property *name* followed by a property *value*, separated by the `''='' ` symbol. Figure 1 depicts an example of one such Song file.

```
name = SimpleTune
bpm = 100
soundbank = ./data/Famicom.sf2
instrument = 0
track = CDEFGAB
instrument = 1
track = GABCDEF
```

Figure 1: An example of a Song file, with two tracks played with two different instruments

Inside a Song file, we define the following properties of a `Song` object:

- **name**: The song name
- **bpm**: The speed of the song in beats per minute, this value should be interpreted as an integer number
- **soundbank**: The path to a file containing a collection of sounds that will be available to the MusicInterpreter class, this value should be interpreted as a String
- **track**: A sequence of symbols corresponding to a *notestring*, This value should be interpreted as a String.
- **instrument**: When this element appears before a track, it defines the instrument from the soundbank that will be used for playing the track, this value should be interpreted as an integer number

You will have to write the code that opens the file at the specified `path`, and use a `BufferedReader` to process it line by line. For determining the property name and value of each line, you might want to use some of the following methods of the `String` class: `replace`, `startsWith`, `split`, and `trim`.

The lines specifying the **name**, **bpm** and **soundbank** properties must be used to set the `myName`, `myBeatsPerMinute` and `mySoundbank` attributes of the `Song` class. The lines specifying the **track** and **instrument** properties must be used to construct a new `MidiTrack` object. Each `MidiTrack` object that you create has to be added to the `myTracks` ArrayList of the `Song` class.

If the file that your method is trying to open does not exist, or if it cannot be opened, the code will throw an `IOException`. You must NOT catch the exception, but pass it to the caller. To do this, add `throws IOException` to the declaration of your method. This will postpone the error handling to the next part, where the `loadSongFromFile` method will be used.

(b) **Creating and playing a Song object**:

In the **PlaySong.java**, modify the main method, so that it creates a `Song` object, and calls the `loadSongFromFile` with a input filename String (`song_file_path` in the example below).

After the file has been loaded, call the `loadSong` and `play` methods of a `MusicInterpreter` object. Your `main` method should look similar to the following code snippet

```
String song_file_path = "./data/07.txt";

Song mySong = new Song();

/*
 * call loadSongFromFile, handling the Exeptions correctly
 */

MusicInterpreter mi = new MusicInterpreter();

// Load the Song and play it
mi.loadSong(mySong);
mi.play();
// close the player so that your program terminates
mi.close();
```

Remember to **handle the exceptions that might arise when opening a file**; *e.g.* exceptions of the `IOException` type. Print a meaningful message when you catch an Exception; i.e. something short that tells the user what the error was (``Could not open the file'', ``The file does not exist'', etc.).

If all goes well, when you execute the `main` method of an instance of type `PlaySong` class with `song_file_path ``./data/07.txt''`, you should be listening to something similar to the Mario Underworld Theme [4].

[4] http://youtu.be/c0SuIMUoShI

(c) **Bonus marks - 5 pts**:

Create a new Song file that reproduces a simplification some tune that you like. It should have at least 10 notes and 2 tracks. In the **name** property of the Song file, you should put an URL pointing to a Youtube video of the tune that you are reproducing. We will use your own code to test your creation.

Question 4: Writing a Reverse Song File (24 points)

In the previous part you wrote a class that would open a file for reading, and build a `Song` object using the properties specified inside a Song file. In this part, you will write code that works in the opposite direction, you will create a class named `SongWriter`, that will take `Song` objects, and convert them into text files with the correct format. To do this, you will have to complete the following tasks inside the **SongWriter.java** file.

a. **The noteToString method**:

This part consists of converting a single note into its notestring representation. Implement the `noteToString` method, which takes as input a `MidiNote` object and returns a String representation of the note in the notestring format of the previous question. You should use the `pitchToNote` Hashtable which translates pitch numbers to note names.

For example, if the `MidiNote` has a `duration` of 12 beats and a *pitch* equal to 65, then the method returns the string `''12F''`. If the `isSilent()` method of such object returns `true`, then the method returns `''12P''` instead.

This method ignores octaves; *e.g.* if the `MidiNote` has a duration of 12 beats a pitch equal to 89 (corresponding to an F note, two octaves up) the method returns `''12F''`. Use the `getOctave()` method of the `MidiNote` class, to get the number of octaves that you need to add or subtract to find the corresponding note symbol in the `pitchToNote` Hashtable. You don't have to worry about a note being sharp or flat, since the `pitchToNote` Hashtable has entries for those pitch values.

b. **The trackToString method**:

Implement the `trackToString` method, which takes a `MidiTrack` object as input, and returns a valid notestring representation of the `MidiTrack`. In this method, use the `noteToString` method to get the notestring representation of each `MidiNote` in the track. You should handle octave changes by checking the octave difference of consecutive `MidiNote` objects in the `MidiTrack`. Add the correct number of octave change symbols, `'<'` or `'>'`, depending of the octave difference.

For example, if a note with pitch equal to 48 (corresponding to a C, with `octave = -1`) is followed by a note with pitch equal to 89 (corresponding to an F note, with `octave = 2`), then the resulting notestring should look like `''...C>>>F...''`. Use the `getOctave()` method of the `MidiNote` class, to compute this difference.

You can test this method by loading a Song file using the code from the Question 3, and checking if the strings returned by the `trackToString` method are equivalent to those in the Song file.

c. **The writeToFile method**:

Implement the `writeToFile` method. This method receives as input a String `filename` and a `Song` object to write. This method will open a file with the given filename *inside the data folder*, and write its content using the format of the Song files of Question 3.

Use the `Filewriter` and `BufferedWriter` to open and write the contents of the file. Your code should write one line for each of the `myName`, `myBeatsPerMinute`, and `mySoundbank` attributes of the `Song` object. These lines correspond to the **name**, **bpm**, and **soundbank** properties of the Song file format.

Your code should write a pair of lines for the `instrumentId` and the notestring representation of each `MidiTrack` in the `myTracks` attribute of the `Song` object. Use `trackToString` to get the

notestring representation of each track. These lines correspond to the **instrument** and **track** properties of the Song file format.

If the path that your method is trying to open is not valid or if the file cannot be opened, the code throws an `IOException`. You should NOT catch the exception, but pass it to the caller. To do this, add `throws IOException` to the declaration of your method.

d. **The main method in SongWriter.java**:

Inside the `Song` class, we provided you with a `revert` method. This method reverses the notes order of every track, by calling the `revert` method of each `MidiNote` class.[5] What you need to do is to write the main method in the **SongWriter.java** file, so that it performs the following steps:

- Load a Song file into a `Song` object.
- Call the `revert` method of the `Song` object.
- Use the `writeToFile` method of a `SongWriter` object to write the new Song file. The name of the new file should be the `name` of the `Song` followed by ``_reverse''. For example, if you load the file ``07.txt'', which has the line `name = underworld`, then the name of the new file will be ``underworld_reverse.txt''.

Remember to **handle the exceptions that might arise when opening a file**; *e.g.* exceptions of the `IOException` type. Print a meaningful message when you catch an Exception. If your code generates a correct Song file, then this file will be playable by the code you wrote on Question 3. Try it out with the ``07.txt'' file from the **data** folder. The resulting sound should be similar to the one on this video `http://youtu.be/lOT19KIwN_o` (the Mario Underworld theme in reverse).

Question 5: Programming questions (16 points)

(a) Let $d(n)$ be defined as the sum of the divisors of n (numbers less than n which divide evenly into n). If $d(a) = b$ and $d(b) = a$, where $a \neq b$, then a and b are an amicable pair and each of a and b are called amicable numbers.

For example, the proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 and 110; and $d(220)$ is 284. The proper divisors of 284 are 1, 2, 4, 71 and 142, their sum is 220, hence $d(284) = 220$.

Evaluate the sum of all the amicable numbers under 10,000.

Be careful in your solution not to double count numbers. For example, the sum of amicable numbers under 500 is 504 - the only amicable numbers under 500 are 220 and 284.

(b) For this problem you will define two arrays, `a` and `b`. Each array contains 10,000 random integers from zero (inclusive) to 10,000 (inclusive). Use the random generator Random ran1 = new Random(1337) to generate the integers for array `a`, and Random ran2 = new Random(5443) to generate the integers for array `b`.

Find the number of times each element in array `a` appears in array `b`. If an element appears in array `a` more than once, do not double count it in array `b`. **What is the sum of each element multiplied by the number of times it appears in this intersection?**

For example: If we are using smaller arrays and
`a = [0, 4, 8, 1, 9, 0, 8, 10, 2, 5]`
`b = [3, 6, 7, 10, 0, 1, 10, 7, 8, 9]`

The intersection using this definition would look like:
{10=2, 9=1, 8=1, 1=1, 0=1}
And so the sum of each element multiplied by its count is
$10 \times 2 + 9 \times 1 + 8 \times 1 + 1 \times 1 + 0 \times 1 = 38$

[5] **You don't have to implement the revert methods**, we already did it for you.

What To Submit

You have to submit one zip file with all your files in it to MyCourses under Assignment 5. If you do not know how to zip files, please ask any search engine or friends. Google might be your best friend with this, and a lot of different little problems as well.

`DoublyLinkedList.java, StringNode.java`	The complete Java code for question 1
`MidiTrack.java`	The complete Java code for question 2.
`Song.java, PlaySong.java`	The complete Java code for question 3.
`SongWriter.java`	The complete Java code for question 4.
`question5.txt`	Your answers for question 5.

Marking Scheme

Up to 20% of points can be removed from each question for bad indentation of your code as well as omitted comments, missing files, or just generally not following instruction. Marks will be removed as well if the class or method names are not respected.

Question 1

Added the reference to a previous node in the `StringNode` class	2	points
The `find` method in the `DoublyLinkedList` class works correctly	3	points
The `insertStart` method in the `DoublyLinkedList` class works correctly	3	points
The `insertEnd` method in the `DoublyLinkedList` class works correctly	3	points
The `remove` method in the `DoublyLinkedList` class works correctly	3	points
The `removeAtIndex` method in the `DoublyLinkedList` class works correctly	3	points
The `toStringReverse` method in the `DoublyLinkedList` class works correctly	3	points
	20	**points**

Question 2

Reading notes (`'A'`, `'B'`, `'C'`, `'D'` , `'E'`, `'F'` ,`'G'`)	4	points
Processing the pause `'P'` character	4	points
Reading notes and pauses with varying durations	4	points
Correctly applying octave changes (the `'>'` and `'<'` characters)	4	points
Reading flat and sharp notes correctly (the `'#'` and `'!'` characters)	4	points
	20	**points**

Question 3

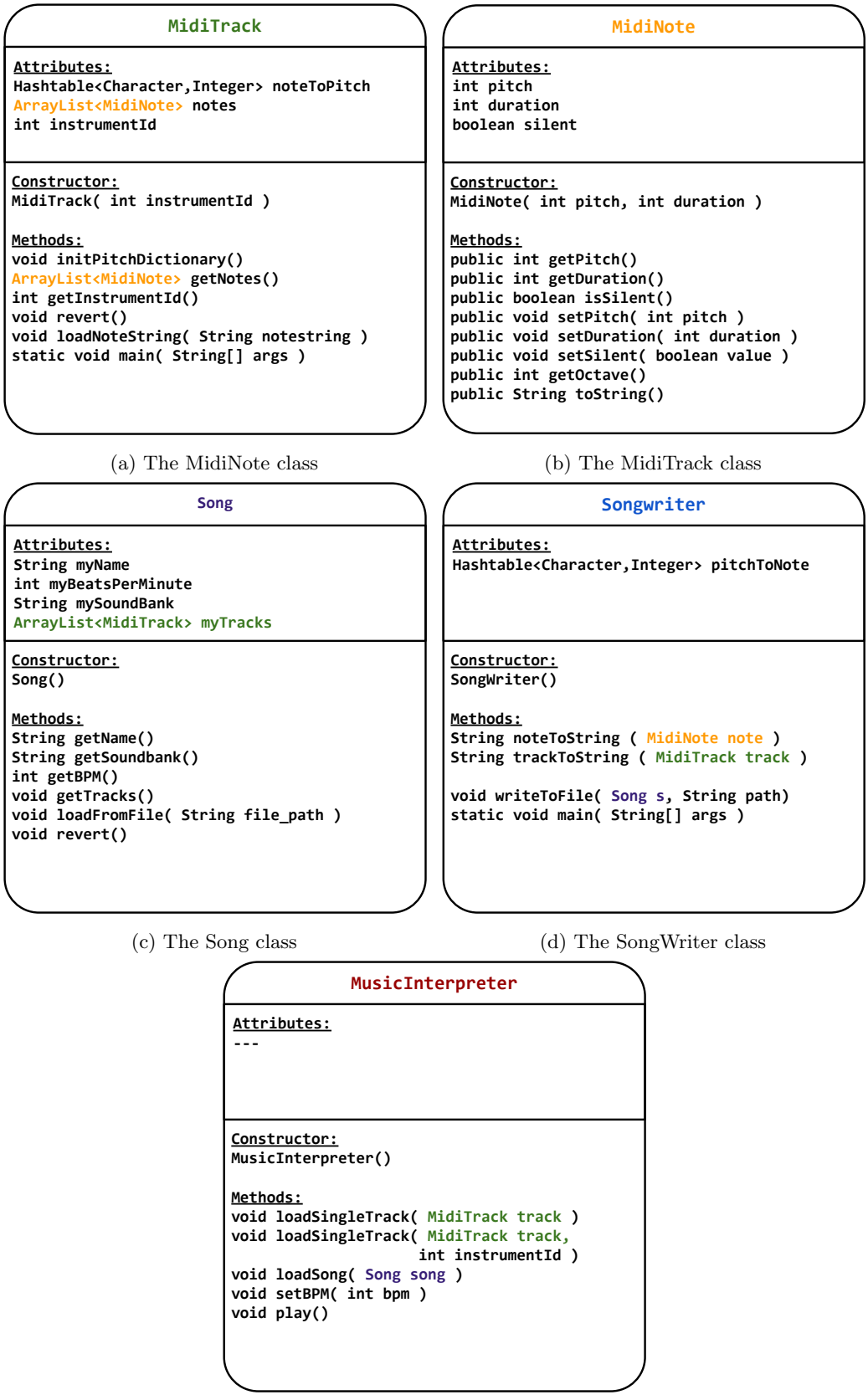
Setting the `myName`, `myBeatsPerMinute` and `mySoundbank` attributes	5	points
Correctly constructing a `MidiTrack` object for every **track** line	5	points
Implemented `main` method in **PlaySong.java**	5	points
When exceptions are caught, the program prints a short message	5	points
	20	**points**

Question 4

Correctly implemented the `noteToString` method	5	points
Correctly implemented the `trackToString` method	5	points
Opening a file and writing the **name**, **bpm** and **soundbank** lines	3	points
Writing the **instrument** and **track** for each `MidiTrack`	3	points
Implemented `main` method in **Songwriter.java**	5	points
When exceptions are caught, the program prints a short message	3	points
	24	**points**

Question 5

5.1	8	points
5.2	8	points
	16	**points**



(a) The MidiNote class

(b) The MidiTrack class

(c) The Song class

(d) The SongWriter class

(e) The MusicInterpreter class

Figure 2: An overview of the classes used in this part.