CSC 110: Fundamentals of Programming I
Assignment #7: Classes and OOP

Due date

December 2nd, 2016 (Friday) at 9:00 pm via submission to conneX.

Late submissions will be accepted until December 5th, 2016 (Monday) at 11:00 am. The late penalty will be a 20% deduction. (Note that a re-submission occurring after the December 2nd due date will be considered a late submission.)

How to hand in your work

Submit the requested files for Part (b) (see below) through the Assignment #7 link on the CSC 110 conneX site. Please make sure you follow all the required steps for submission (including confirming your submission). Part (a) is not marked.

Learning outcomes

When you have completed this assignment, you should understand:

- How to *create an instance of a class* (*object instantiation*).
- How to invoke an object's *instance methods*.
- How to create and use an *array of objects*.
- How to *send program output to a file*.

Part (a): Problems from the Textbook

Complete chapter 8 self-check problems 1 to 3, 5 to 8, 10, 11, 13, 14, 16, 17, 19 to 21, 24 and 25. Compare your answers to those available at:

`http://www.buildingjavaprograms.com/self-check-solutions-4ed.html`

Part (b): Transposing Music

Problem description

Your task in this assignment is to write *TransposeSong.java* and as part of completing this program you will write and use other classes:

- *Note.java*
- *Voice.java*

The basic problem is we have files containing *notes* (i.e., *pitches* and *durations*) in a text-like format. The overall task of your assignment is to transpose the notes in song files. Here are the contents of one file named *01_example.sng*:

```
tempo 120
voices 1
title O Canada

instrument soprano
notes 4
g4 2
b_4 4.
b_4 8
e_4 2.
```

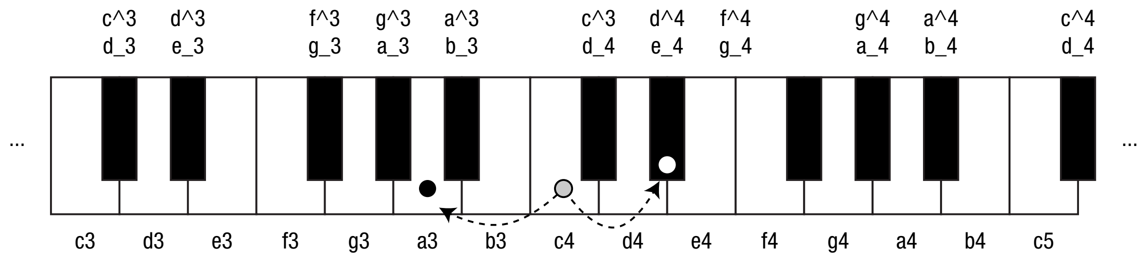
You can play this song by using the provided JAR file. Note that this is the first time we have used JAR files in the course; it is one way that a large number of classes can be combined together into one archive. (And do not worry—you will *not* be asked to create a JAR file for this assignment!) Assuming *a7.jar* and *01_xample.sng* are in the same directory, type the following:

```
$ java -jar a7.jar 01_example.sng
```

If your computer is properly configured for Java audio you will hear the first four notes of our national anthem. (*Most computers are already correctly configured and so you probably do not need to do anything special for this assignment. Ignore any error messages printed by the a7.jar program. If you are in ECS 258 you will need to plug your earphones or earbuds into the workstation in order to hear the sound.)*

Notes consist of a *pitch* and *duration*. Pitches consist of a *note letter* (or *tone)* plus an *octave number*, and there may also be a single *accidental* marking, where "^" means "sharp" and "_" means "flat". (You may have seen these symbols in your musical education as ♯ and ♭.) For example, the first note "g4 2" means G in the fourth octave, also known as *G above middle C*, and the note's length or *duration* is "2" (a *half note*). The second note "b_4 4." is B flat above middle C and with a duration of "4." (a *dotted-quarter note*).

Transposing music means raising (i.e., *transpose up*) or lowering (i.e, *transpose down*) the pitches of notes in a consistent way. (Transposition *does not* change the duration of notes.) Consider the diagram below showing part of piano's keyboard.



Musicians refer to middle C as *c4* (i.e., the piano key with the gray dot). If we were to transpose *c4* up by three pitches (i.e., *three semitones*), then the new pitch will be *e_4* (or "E-flat above middle C"), or the key with the white dot. This new pitch also has two possible names (i.e., it could also be named *d^4* or "D-sharp above middle C"). A note will have at most one accidental (i.e., notes such as *c^^4* are not possible).

Transposing down is similar. If we do this to *c4* by transposing down three pitches, the new pitch will be *a3* (or "A natural below middle C")—see the black-dot key.

Your task is to write *TransposeSong.java*. It will take three arguments:

1. The *name* of some song file.
2. The *name of the file* in which to store the transposed song (and make sure this is different than the filename in 1).
3. The *amount of the transposition* as a positive or negative integer (or zero).

For example, to transpose the example down by three pitches and to store the result in *example_down3.sng*, we would enter the following at the command prompt:

```
$ java TransposeSong 01_example.sng example_down3.sng -3
```

Note that the program does not play the song but simply performs the transposition. (You can play the result stored in the new file using the JAR file in the manner described on page 2.) Here is one possible *example_down3.sng*:

```
tempo 120
voices 1
title  O Canada

instrument soprano
notes 4
e4 2
g4 4.
g4 8
c4 2.
```

One last detail: Some of the note entries in a song are not pitches at all but are instead *rests* (or *silences*) having a duration. When transposing a song, you need only reproduce the same rest (i.e., a note of "*r 4*" before transposition begins is just "*r 4*" after transposition is ends).

What you are to write

The class ***Note.java*** will be used to store information about a single note's pitch and duration. You are welcome to choose instance variables that make most sense to you (i.e., you may or may not choose to separate the pitch into a *tone* and an *octave*.) Methods in this class will also be used to transpose the note.

- There must exist private attributes to store information about the note.
- The constructor *Note(String pitch, String duration)*: This will assign the Note's instance variables with correct values given the *pitch* and *duration* provided to the constructor.
- *private void transposeUp()* and *private void transposeDown()*: These methods will transpose the note up one pitch/semitone and down one pitch/semitone. (See the Appendix for some ideas on how to determine the higher or lower pitch given some existing pitch.)
- *public void transpose(int semitones):* This will call *transposeUp()* or *transposeDown()* as many times as is needed. For example, in order to transpose up by three semitones, the method will end up calling *transposeUp()* three times. A negative value for *semitones* means "transpose down", and a positive value means "transpose up".
- *public String toString():* Create a string representing the note. The string would represent the note as it would appear in a song file.

The class ***Voice.java*** will be used to store information about a sequence of notes. Notice on page 2 that a voice always begins with an *instrument* and the *number of notes* in the voice part. These can easily be read using a scanner that is passed to the Voice constructor. The number of notes should be used to create an array of *Note* instances and to control the for-loop needed to read notes from the scanner.

- There must exist a private array of *Notes* called *notes* and a private *String* called *instrument*.
- The constructor *Voice(Scanner input)*: Using the scanner parameter this constructor will read the instrument, number of notes, and notes themselves; and the method will store these in instance variables. This constructor will only ever be called when the scanner's cursor is just before the string "instrument" in some song file. You will need to call *next()* or *nextInt()* on the scanner as appropriate. This method must need to create an array of *Note* with the correct size given the number of notes in the voice.
- *public String toString()*: Create a string representing the instrument, number of notes, and note values themselves. The result will be a string concatenation probably involving, amongst other things, the use of the newline character

("\n"). Put differently, the result of *toString()* with an untransposed *Voice* instance would be exactly the same as the text of the voice within the file.

- *public void toStream(PrintStream ps)*: This is used to print the contents of the *Voice* object to the opened *PrintStream* named *ps*. The method should depend upon the result of *toString().*
- *public void transpose(int semitones)*: Transpose each *Note* in the *notes* array by the given number of *semitones*. A negative amount means transposing down, while a positive amount means transposing up.

And finally there is ***TransposeSong.java*** which will directly use instances of the *Voice* class.

- The program will take three arguments on the command line (as shown on page 3).
- The input file will be read in using a scanner (***do not read from the console!***), and some lines of information are to be stored in variables such that they can be copied to the output file (e.g., tempo, title, number of voices). Make sure you use *next()* or *nextLine()* with the scanner as appropriate. You are responsible for opening the file and connecting it to a scanner.
- *There may be up to four voices in a song file*. Once all voices are read into the program, they are to be transposed by the same number of semitones (i.e., transposed down for a negative value, transposed up for a positive value). The amount of the transposition is given by the third command-line argument.
- Once the voices are transposed, the resulting song (i.e., song information such as its *tempo*, *title* and *number of voices*; then each voice with its *instrument*, *number of notes*, and the *notes themselves*) is to be output to the file whose name is provided as the second command-line argument. You are responsible for opening the file and for connecting it to a print stream. You will want to use calls to the *toString()* method implemented in the *Voice* class as part of your solution. (And make sure you always *close()* the print stream when output for the whole song is completely finished.)

What the marker will look for

- Compiles & runs on the lab machines.
- Produces the expected output (i.e., notes of input file properly transposed such that output file is correct).
- Correctly implements and uses the requested methods.
- Is well-structured through the use of meaningful methods.
- Is appropriately documented and matches the style guidelines.

Files to submit: ***TransposeSong.java, Note.java*** and ***Voice.java*** *via the Assignment #7 link on conneX.*

Grading scheme

- "A" grade: A submission completing the requirements of the assignment. The program runs without any problems using structures as required (i.e., *Note*, *Voice* and *TransposeSong* classes). Some marks will be given for the quality of the solution.

- "B" grade: A submission completing most of the requirements of the assignment. The program runs with some problems but uses the required classes.

- "C or D" grade: A serious attempt at completing requirements for the assignment. The program runs with major problems, or does not use two-dimensional arrays or is not decomposed into methods.

- "F" grade: Either no submission given, the submission does not compile, or submission represents very little work.

Appendix: Transposing up or down one semitone

Many students will be new to the description of musical *pitch* and *duration* as used in this assignment. Even students with some experience working with western musical notation may wonder how best to transpose a pitch up or down. I offer three observations.

1. Although you may be tempted to solve the general transposition problem, I recommend you stick with transposing up one semitone or down one semitone. To transpose for larger numbers of semitones just repeat these simpler operations the appropriate number of time.

2. Determining the octave of a transposed note is easier if you stick with the simpler operations. When transposing down one note from a C, the octave number always decreases by one (e.g., from *c4* to *b3*). When transposing one note from a B, the octave number always increased by one (e.g., from *b3* to *c4*). These two intervals are the only spots where there is a change in octave number.

3. *Enharmonic spellings* of notes are to be expected. For example, *d^3* is the same as *e_3*. There are also some spellings not often found in western notation (e.g., *f_5* is just *e5*). Because of this, determining the next pitch up or down a semitone can be so darn tricky, and so I'll suggest you take advantage of the two-dimensional table of strings shown on the next page. At any row you will see that the pitch in column 0 is a semitone lower than the pitch in column 1 (and that the pitch in column 1 is a semitone higher than the pitch in column 0). If you have this table as a class variable in your *Note* class, you can use it to help implement *transposeUp* and *transposeDown*.

```
  private static final String[][] transposeTable = {
        {"c_", "c"},
        {"c", "d_"},
        {"c", "c^"},
        {"c^", "d"},
        {"d_", "d"},
        {"d", "d^"},
        {"d", "e_"},
        {"d^", "e"},
        {"e_", "e"},
        {"e", "e^"},
        {"e_", "f_"},
        {"e", "f"},
        {"e^", "f^"},
        {"f", "f^"},
        {"f", "g_"},
        {"f^", "g"},
        {"g", "g^"},
        {"g", "a_"},
        {"g^", "a"},
        {"a_", "a"},
        {"a", "a^"},
        {"a", "b_"},
        {"a^", "b"},
        {"b_", "b"},
        {"b_", "c_"},
        {"b", "b^"},
        {"b", "c"},
        {"b^", "c^"}
    };
```