CS 145 Homework 5
Computer as Factory Worker

1 Overview

In this homework, you will learn how to organize and process sequences of data using arrays. You will do this in the context of composing and transforming music.

2 Background

You do not need to be a musician to complete this homework. But because we want this exercise to be somewhat faithful to the actual discipline of music, we have tried not to oversimplify matters. Below we try to provide as much background information as is necessary to complete this homework.

2.1 Letters and Octaves

The "atom" of music is the note. Each note represents a pressure wave that travels from an emitter to our ear. Western musicians have settled on 12 base frequencies for these pressure waves. Seven of them are referred to with the first seven letters of the alphabet: A, B, C, D, E, F, and G. The other five are offset from these: A♭, B♭, C♯, E♭, and F♯. (♯ is read "sharp," and ♭ is read "flat." The sharpness or flatness of a note is called its *accidental*.) Musicians do use other labels from time to time, but these labels are just aliases to one of the 12 base frequencies.

If you emit a pressure wave whose frequency is a multiple of some other wave, the two waves sound similar when they reach our ears. In fact, we say the two waves are the same note, just in different octaves. For example, A4 (note A in the 4th octave) oscillates at 440 cycles per second, and A5 (note A in the 5th octave) oscillates at 880 cycles per second. You can go the other direction too: A3 oscillates at 220 cycles per second, A2 at 110, and A1 at 55. Eventually you leave the range of frequencies that humans can perceive and computers can synthesize.

2.2 Halfstep IDs

To describe a note without ambiguity, we introduce the numeric *halfstep identifier*. We say that C in octave 0 has halfstep ID 0. The IDs increase from there. Consider this table enumerating the halfstep IDs for the first five octaves:

Octave	C	C♯	D	E♭	E	F	F♯	G	A♭	A	B♭	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
...	...											

A couple of ideas emerge from this labeling. First, we see that sharp notes are just one halfstep above their corresponding non-sharp note. Flat notes are just one halfstep below. Second, we can determine a note's distance from another. Four example, three halfsteps above G is B♭. We will use this notion of halfstep distance later to generate scales and transpose and invert music.

2.3 Duration

Besides frequency, notes also have a duration—how long is the pressure wave emitted? Just like with measuring cups and primitive types, durations come in a fixed set of choices. A *whole note* is the longest. A *half note* is half the length of a whole. A *quarter note* is a quarter of a whole. And so on, with *eighth*, *sixteenth*, and *thirty-second* notes.

Additionally, a note may be *dotted*, meaning that it's 1.5 times its usual length. For example, a quarter note may last 0.5 seconds, while a dotted quarter note lasts 0.75 seconds.

2.4 Amplitude

We control the volume of a sound wave through its *amplitude*. For this homework, each note has its own amplitude that ranges from 0 (silent) to 1 (loud).

3 Supporting Classes

Now that we have described the properties of a note, we describe several classes used to represent one in code. These classes are provided to you in the SpecChecker JAR file.

3.1 Letter

To specify a note's letter, we use one of the following constants:

```
Letter.A
Letter.B
Letter.C
Letter.D
Letter.E
Letter.F
Letter.G
```

3.2 Duration

To specify a note's duration, we use one of the following constants:

```
Duration.WHOLE
Duration.HALF
Duration.QUARTER
Duration.EIGHTH
Duration.SIXTEENTH
Duration.THIRTYSECOND
```

3.3 Accidental

To specify a note's sharpness or flatness, we use one of the following constants:

```
Accidental.SHARP
Accidental.FLAT
Accidental.NATURAL (neither sharp nor flat)
```

3.4 Note

The previous classes contained only constants, but `Note` is more involved. We can make a `Note` in many different ways, depending on what information we have and what we are willing to accept for default values. The amplitude defaults to 1.0, the accidental defaults to `Accidental.NATURAL`, and the dottedness defaults to `false`.

If we know the note's letter and octave, we can use one of these constructors:

```
Note(Letter letter, int octave, Duration duration)
Note(Letter letter, int octave, Duration duration, double amplitude)
Note(Letter letter, Accidental accidental, int octave, Duration duration)
Note(Letter letter, Accidental accidental, int octave, Duration duration, double amplitude,
     boolean isDotted)
```

For example, to make an A4 quarter note, we write `new Note(Letter.A, 4, Duration.QUARTER)`. If we know the note's halfstep ID, we can use one of these constructors:

```
Note(int halfstepID, Duration duration)
Note(int halfstepID, Duration duration, double amplitude)
Note(int halfstepID, Duration duration, double amplitude, boolean isDotted)
```

There's one last constructor for advanced music programmers:

```
Note(String abbreviation)
```

The abbreviation is a single `String` that describes all of a note's properties in a compact form. It is best described with a few examples:

```
new Note("4C5") -- quarter note C in 5th octave
new Note("2.Ab2") -- dotted half note A flat in 2nd octave
new Note("8F#3(0.5) -- eighth note F sharp in the 3rd octave at volume 0.5
```

The `Note` class provides the following getters, which will be helpful when you need to create a new note based on an existing one:

```
int getHalfstepID()
boolean isDotted()
Letter getLetter()
Accidental getAccidental()
int getOctave()
double getAmplitude()
Duration getDuration()
```

Several other methods are provided that are not really necessary to complete this homework:

```
double getFrequency()
short[] getSamples(int beatsPerMinute)
String toAbbreviation()
static Note[] parse(String abbreviations)
```

The last of these may be convenient to compactly produce an array of notes for testing. For example:

```
Note[] twinkle = Note.parse("4C4 4C4 4G4 4G4 4A4 4A4 2G4");
```

3.5 WavIO

The `WavIO` class converts an array of `Note`s into a sound file (of the WAV format) that we can actually playback on our computer. It provides the following method:

```
static write(Note[] notes, int beatsPerMinute, String path)
```

The beats per minute determines how fast the song is played. Different musical genres have different average beats per minute, with slower music under 100 and faster above. If it matters to you, a whole note is assumed to be four beats long.

To save `twinkle` to a file in our `Desktop` folder, we write:

```
WavIO.write(twinkle, 100, new String(System.getProperty("user.home") + "/Desktop/twinkle.wav"));
```

4 Requirements

Complete the two classes described below. Place all classes in package `hw5`. Make all methods `static`.

4.1 Main

Write a class `Main` with a `main` method, which you are encouraged to use to test your code. Nothing in particular is required of it, but it must exist.

4.2 MusicUtilities

Write a class `MusicUtilities` with several methods for generating and composing musical sequences. In all cases, these methods return a brand new array. None of the methods that receive an array of `Note`s as a parameter modify the array. This class has the following:

1. Method `getScale`, which accepts two parameters in this order:

   - A root note of type `Note`
   - An array of halfstep ID offsets, each of type `int`

   It returns an array of `Note`s for the requested scale. Element 0 in the scale array is the given root. Element 1 is offset from the root by the offset 0. Element 2 is offset from element 1 by offset 1. And so on until the final offset. All notes have the same duration, amplitude, and dottedness as the root.

2. Method `getMajorScale`, which accepts one parameter: a root note of type `Note`. It returns an `Note` array of 8 notes of the major scale starting at the root. The offsets for the major scale are 2, 2, 1, 2, 2, 2, and 1. Applied to root C, this method returns and array of C, D, E, F, G, A, B, C. The offsets add up to 12, and since there are 12 halfsteps in an octave, we end on the same letter as we started.

3. Method `getMinorPentatonicBluesScale`, which accepts one parameter: a root note of type `Note`. It returns an `Note` array of 6 notes of the minor pentatonic blues scale starting at the root. The offsets for this scale are 3, 2, 2, 3, and 2.

4. Method `getBluesScale`, which accepts one parameter: a root note of type `Note`. It returns an `Note` array of 7 notes of the blues scale starting at the root. The offsets for this scale are 3, 2, 1, 1, 3, and 2.

5. Method `getNaturalMinorScale`, which accepts one parameter: a root note of type `Note`. It returns an `Note` array of 8 notes of the natural minor scale starting at the root. The offsets for this scale are 2, 1, 2, 2, 1, 2, and 2.

6. Method `join`, which accepts two parameters in this order:

   - A first array of notes, each of type `Note`
   - A second array of notes, each of type `Note`

   It returns a new array containing the elements of the first array followed by the elements of the second array.

7. Method `repeat`, which accepts two parameters in this order:

   - An array of notes, each of type `Note`
   - A number of cycles of type `int`

   It returns a new array in which the note sequence in the parameter array is repeated the specified number of times. 0 cycles means it's not played at all, 1 cycle means it plays once, 2 means it plays twice, and so on.

8. Method `ramplify`, which accepts three parameters in this order:

   - An array of notes, each of type `Note`
   - A starting amplitude of type `double`
   - An ending amplitude of type `double`

   It returns a new array that "ramps" up or down the amplitude of the given note sequence. The first note in the returned array has the starting amplitude, and the last note has the ending amplitude. The intervening notes step between these two amplitudes equidistantly. For example, ramplifying four notes from amplitude 0.5 to 0.8 will yield an array with amplitudes 0.5, 0.6, 0.7, and 0.8. Apart from the amplitudes, the notes in the returned array are the same as the parameter array.

9. Method `reverse`, which accepts one parameter: an array of notes, each of type `Note`. It returns a new array that is the reversal of the parameter array. For example, reversing an array of C, D, E yields an array of E, D, C. Apart from the note order, the notes in the returned array are the same as the parameter array.

10. Method `transpose`, which accepts two parameters in the following order:

    - An array of notes, each of type `Note`

- A new root of type `Note`

It returns a new `Note` array that is like the parameter array, but which has been shifted up or down some number of halfsteps. The amount of this shift is determined by the difference between halfstep IDs of the first note of the sequence and the new root. The first note of the returned array, then, will have the same halfstep ID as the new root. The duration, amplitude, and dottedness of the notes in the returned array are the same as the parameter array.

Since all notes are shifted by the exact same offset and because music is perceived in a relative manner, transposition doesn't significantly alter a piece of music. It is often used to bring a piece of music into a range that's more easily played or sung.

11. Method `invert`, which accepts two parameters in the following order:

    - An array of notes, each of type `Note`
    - A pivot note of type `Note`

    It returns the *inversion* of the parameter array. This method is a lot like `transpose` in structure, but instead of shifting a note up or down by a fixed offset, it "flips" each note over the pivot. If a note is 5 halfsteps above the pivot, its inversion will be 5 halfsteps below. If a note is 10 halfsteps below, its inversion is 10 halfsteps above. If a note coincides with the pivot, its inversion is unchanged. The duration, amplitude, and dottedness of the notes in the returned array are the same as the parameter array.

    Inversion is used by composers to generate a variation of a sequence.

5 Extra

For an extra credit participation point, share a WAV file for melody that you've programmed on Piazza under folder `hw5_share`.

6 Submission

To submit your work for grading:

1. Put the SpecChecker for this homework in your Build Path. Run the SpecChecker as a Java Application and fix problems until all tests pass.

2. Commit and push your work to your repository. Verify that your solution is on Bitbucket.

A passing SpecChecker does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The SpecChecker checks some of them, but not all.

- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, CS 145 moles, and a piece of software

called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

- Your code must be submitted correctly and on time. Most excuses devolve into, "I started too late." The fix for this problem is not an extension.