Enumerations and Generic Containers

CSE/IT 213

NMT Department of Computer Science and Engineering

"Phrasing can never be made a mechanical process, without perverting and artificializing the whole manner of delivery."

— Samuel Silas Curry

"Everyday life is like programming, I guess. If you love something you can put beauty into it."

— Donald Knuth

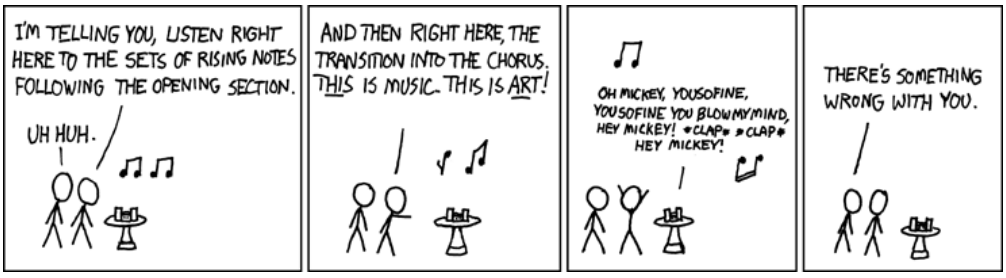


Figure 1: `https://xkcd.com/193/`

Introduction

In this assignment you will learn how to use a few more handy tools in the Java language. In particular, you will see some new features of enumeration classes which you may not have been familiar with, and get an early look at how generic classes work in Java. The assignment has interesting applications to digital music production, which just goes to show how a little bit of code can go a long way when the right libraries are available!

Enumerations

You've already used enumerations in previous projects, so the basic concepts behind `enum` classes should already be familiar to you. You use an `enum` when you have a type that is best represented as a small finite list of possible instances. A good example is the set of all days in a week:

```
1 public enum Weekday {
2     SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
3 }
```

Figure 2: Weekdays represented as an `enum`

But enumerations in Java are more powerful than they seem at first glance. Just like any class, and enumeration can have its own methods and instance variables. You can write code around your enumerations to give additional meanings or representations to their values. By adding some code, we can turn our `Weekday` enum into a lookup table that tells us more about each day:

```java
public enum Weekday {
    SUNDAY("Sun", false),
    MONDAY("Mon", true),
    TUESDAY("Tue", true),
    WEDNESDAY("Wed", true),
    THURSDAY("Thu", true),
    FRIDAY("Fri", true),
    SATURDAY("Sat", false);

    private String abbrev;
    private boolean workday;

    private Weekday(String abbrev, boolean workday) {
        this.abbrev = abbrev;
        this.workday = workday;
    }

    public boolean isWorkday() {
        return workday;
    }

    public String toString() {
        return this.abbrev;
    }
}
```

Figure 3: Weekday enumeration extended into a small lookup table

Here, each of the enumerated values, `SUNDAY` through `SATURDAY`, is defined by its own call to the `enum`'s constructor. Since the constructor is required to be `private`, the basic premise of enumerations still stands – there can still only ever be 7 instances of `Weekday`.

Apart from allowing you to define your own methods, enumerations also come with some useful builtin methods that are worth knowing about. For instance, if you have a `Weekday` called `day`, then `day.name()` will convert it to a `String`, like `"FRIDAY"`. You can also use the static method `Weekday.values()` to create an *array* containing every value of the enumeration, in order. This is particularly useful if you need to iterate over every value in the enumeration, like so:

```java
for (Weekday day : Weekday.values()) {
    System.out.printf("%d %s (%s)\n", day.ordinal(), day.name(), day);
}
```

Figure 4: Using the builtin methods for enumerations

Finally, the method `day.ordinal()` returns the numeric value of that day in the enumeration. This is also the index where that day can be found in `Weekday.values()`. Together, they make it easy to convert an enum to and from an integer flag, without having to write a `switch` statement.

```java
Weekday[] week = Weekday.values();
Weekday day = Weekday.FRIDAY;
Weekday threeDaysLater = week[(day.ordinal() + 3) % week.length];
System.out.println(threeDaysLater.name());
```

```
MONDAY
```

Figure 5: Converting between values and ordinals

Array Lists

If there's a main reason why Java has been successful, it probably comes down to its standard library. While other languages require you to implement fundamental data structures and algorithms on your own, Java comes with hundreds of libraries that have already done the most common jobs for you. It's easy to get started using linked lists, queues, or hash maps, without needing to worry about their implementation details.

This introduces a new problem for the language to solve. How do you make a class to store data of any arbitrary type? The answer is to create a *template* class, or a *generic* class, as it's called in Java. We'll learn more about generics further on in this course, but for now you only need to be familiar with the syntax to use them. So let's dissect a short example program:

```
1  import java.util.ArrayList;
2
3  public class Main {
4      public static void main(String[] args) {
5          ArrayList<String> crew = new ArrayList<String>();
6
7          crew.add("James T. Kirk");
8          crew.add("Spock");
9          crew.add("Leonard McCoy");
10         crew.add("Nyota Uhura");
11
12         String spock = crew.get(1);
13         crew.remove(spock);
14
15         for (String member : crew) {
16             System.out.println(member);
17         }
18
19         // Search for "Spock"
20         System.out.println(crew.indexOf(spock));
21     }
22 }
```

Figure 6: Example use of `ArrayLists`

The new syntax appears on line 5, where we declare the variable `ArrayList<String> crew = ...`. The angle brackets enclose a *type parameter* to the `ArrayList` constructor, which tells the class what *type* of array list it's making. Before you can create an array list Java needs to be able to answer the question, *an array list of what?*

You can see the same thing at the constructor call, `new ArrayList<String>();`, for the same reason. You cannot construct an `ArrayList`, because that's actually an incomplete type. It's best to think of the full type of `crew` as being an `ArrayList<String>` instead. However, since this sometimes makes the code a little too verbose, Java added some syntactic sugar for constructing generic classes. Optionally, you can also say:

```
ArrayList<String> crew = new ArrayList<>();
```

And the constructor's type argument, `<String>`, will be implicitly filled in for you. When you leave the parameter blank, it just assumes you mean to construct the same type you're declaring.

Array lists can be much more useful than ordinary arrays in Java. With an array, you need set a static size when it is declared, and it can never change. An array list can grow to hold an arbitrary number of elements, dynamically reallocating space as more elements are added. This is a much more flexible solution when you don't know ahead of time how much data you will need to store.

You can use `alist.get(idx)` and `alist.set(idx, elem)` to read and write to the array list. These are roughly equivalent to `array[idx]` and `array[idx] = elem` for ordinary arrays. Using either of

these to access an out of bounds index will still throw an `IndexOutOfBoundsException`, so when necessary you can use `alist.size()` to check the size of the array list [1].

There are many other useful builtin tools that make array lists more convenient than ordinary arrays. You can use `alist.add(elem)` to insert a new item at the end of the array, and `alist.remove(elem)` to delete an element. The problems of resizing the array or rearranging elements are all taken care of behind the scenes, so you don't have to worry about what the array "looks like" in memory!

JUnit Test Suites

In the previous lab you used JUnit to verify your implementation in your classes were correct and working as intended. Having a lot of individual test classes can lead to a lot of files/tabs opened in your IDE or work space. Luckily, there is a way to run all your individual tests classes from a single test class. This is known as a Test Suite.

Let us assume we have three classes (`Calendar.class`, `Weekday.class`, `Time.class`) and have test classes for each of them:

```
1  import org.junit.Test;
2
3  public class CalendarTest {
4
5      @Test
6      public void shouldReturnDayOfYear() {
7          ...
8      }
9      ...
10 }
```

Figure 7: Calendar test class

```
1  import org.junit.Test;
2
3  public class WeekdayTest {
4
5      @Test
6      public void shouldReturnFriWorkDay() {
7          ...
8      }
9      ...
10 }
```

Figure 8: Weekday test class

[1] Vanilla arrays use an attribute `array.length` to tell you their size. The mismatch between the two naming schemes can sometimes be confusing.

```java
import org.junit.Test;

public class TimeTest {

    @Test
    public void shouldReturnAfterNoon() {
        ...
    }
    ...
}
```

Figure 9: Time test class

We can put these tests classes in a test suite and run the test suite instead of each individual test class. You need to create a new class under your testing directory. When you do, you need to provide it with the annotation `@RunWith` and pass it with `Suite.class` which will import 'org.junit.runners.Suite'. Junit will recognize that this is a test class based on the annotations. You need to provide an annotation named `@Suite.SuiteClasses` and fill it with the test classes you have written for your other classes. You can name your test suite class however you would like as long as you understand that it is meant for all tests and meaningful.

```java
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
        CalendarTest.class,
        WeekdayTest.class,
        TimeTest.class
})

public class JUnitTestSuite {
    // This class is to remain empty.
    // It is used as a holder for the above annotations.
}
```

Figure 10: Structure of setting up a Test Suite.

The first test class you provide will execute all of the tests provided inside the test class first along with the following test classes. In this case, `CalendarTest` will execute all of its tests first followed by `WeekdayTest` and `TimeTest`. Leave the class below empty as this is needed as a holder for the annotations. When writing tests for your enumeration class (I.e. Weekday) to verify it defines names (SUNDAY, MONDAY, TUESDAY, etc..) you wrote is redundant and unnecessary. It is valuable to write tests for your methods of the enumeration class to ensure the behavior of them are working properly. For example using the `Weekday` class, you could test if your `isWorkday()` method is returning a work day or not depending on what day.

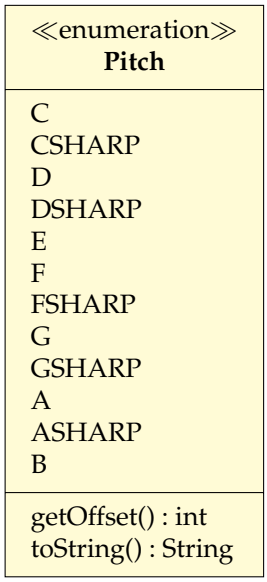
Problems

Problem 1: Music

For this problem, you will design and implement four classes to create a program that plays music. First you will use enumerations to represent the Pitch and Tempo of a note. Then you will write code to represent Notes by putting these pieces together. Finally, you will use an `ArrayList` to represent a song as a sequence of notes, and play the song! Relevant background information is listed in the appendix below.

Pitch

Create an enumeration called `Pitch.java` that represents the 12 different notes in an octave:



The `enum` class should have add two extra methods. The first, `getOffset()`, returns the offset shown in the table below; `Pitch.A.getOffset()` should return `0`, and `Pitch.C.getOffset()` should return `-9`.

Note	C	$C^{\#}$	D	$D^{\#}$	E	F	$F^{\#}$	G	$G^{\#}$	A	$A^{\#}$	B
Value	0	1	2	3	4	5	6	7	8	9	10	11
Offset	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2

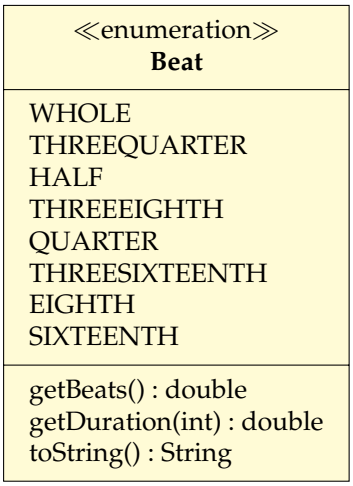
Figure 11: Table of Note values and offsets

The `toString()` method should return the note in the format `"C#"`. That is, use a `"#"` symbol instead of the word `"SHARP"` when you print out the note. For non-sharp notes, `toString()` returns the same as `name()`.

You can add private attributes and constructors to the enumeration as you see fit to make implementing these two methods easier.

Beat

Implement another enumeration, `Beat.java` to represent the duration of a note:



Assuming the music is in standard $\frac{4}{4}$ time, a quarter note plays for a duration of 1 beat. The other beats are defined predictably in terms of the quarter note:

Value	Beats
`WHOLE`	4
`THREEQUARTER`	3
`HALF`	2
`THREEEIGHTH`	3/2
`QUARTER`	1
`THREESIXTEENTH`	3/4
`EIGHTH`	1/2
`SIXTEENTH`	1/4

The `getBeats()` method simply returns the values from the above table. The method `getDuration()` calculates the time, in seconds, that the note should be played given a `tempo` in units of beats per minute. The input says how many quarter notes can be played in 60 seconds, and the output is the number of seconds it takes to play the beat (`((60.0 * beats) / tempo)`).

The `toString()` method prints the fraction of a `WHOLE` note that the beat represents, wrapped in parentheses. For example, `Beat.WHOLE.toString()` returns `"(1)"`, `Beat.QUARTER.toString()` returns `"(1/4)"`, and `Beat.THREESIXTEENTH.toString()` returns `"(3/16)"`.

Again, you can add private attributes to the enumeration to make the method easier to define.

Note

You can put these enumerations together to represent a Note. See the Appendix for additional information on the values and the conversions used. Write the class `Note.java` to match the following interface:

Note
- note : Pitch «get/set» - octave : int «get/set» - length : Beat «get/set»
+ Note() + Note(Pitch, int, Beat) + Note(String, Beat) + Note(int, Beat) + Note(double, Beat) + getSPN() : String + setSPN(String) : void + getMIDI() : int + setMIDI(int) : void + getFrequency() : double + setFrequency(double) : void + toString() : String

The class has five constructors:

1. The default constructor creates a C_4 quarter note.
2. The main constructor takes a `Pitch`, an octave, and a `Beat`, and checks that the octave is within the range from -1 to 9 — if the note is $G^\#$, A, $A^\#$, or B, then the maximum valid octave is reduced to 8. If the input is valid, then this constructor sets the pitch, octave, and beat length of the note directly.
3. The third constructor checks that a `String` is a valid note in scientific pitch notation (SPN), and if so parses it to set the pitch and octave.
4. The fourth constructor takes a MIDI number from 0 to 127, and converts it to set the note and octave.
5. And the final constructor takes a frequency and uses it to set the pitch and octave to the nearest corresponding MIDI number.

If an input to any of the constructors does not correspond to a valid MIDI number, fallback to the default of creating a C_4 note of the given length. Writing the constructors will be much easier after you have written the `get` and `set` methods to convert a note to and from its SPN string, MIDI number, and frequency.

The SPN string is just the name of the note followed by its octave. Examples of valid SPN strings are `"C-1"`, `"D#4"`, and `"E9"`. You can use a regular expression to check whether an SPN string is valid before parsing and setting the pitch and octave values:

```
if (spn.matches("(([CDFGA]#?|[EB])(-1|[0-8])|([CDF]#?|[EG])9)")) {
    // Convert the String to a MIDI number
}
```

The MIDI number numbers the notes from C_{-1} to G_9 from 0 to 127 (See Figure 12). Any note outside that range is invalid! You can convert the pitch and octave to a MIDI number with this equation:

$$midi(pitch, octave) = 69 + offset[pitch] + 12 \cdot (octave - 4) \tag{1}$$

And finding the reverse is simply a matter of dividing and taking the remainder:

$$octave = \lfloor midi/12 \rfloor - 1 \tag{2}$$
$$value[pitch] = midi \mod 12 \tag{3}$$

The frequency is defined in terms of the MIDI number, m, with this equation:

$$f = 440 \cdot 2^{(midi-69)/12} \tag{4}$$

And the inverse of this equation converts in the other direction:

$$midi = 12 \cdot \log_2\left(\frac{f}{440}\right) + 69 \tag{5}$$

When setting the note based on any of these, it is important to check that the result is within the range of valid MIDI notes. The setters should do nothing if given a note higher than C_{-1} or lower than G_9, and the constructors should fallback to creating a C_4 note.

Likewise, `setOctave()` should always check that the new octave is between -1 and 9, (or between -1 and 8 if the pitch is $\geq G^{\#}$). Similarly, `setPitch()` should avoid setting the note to $G^{\#}$, A, $A^{\#}$, or B if the current octave is 9.

The `toString()` method returns SPN of the note, followed by the beat, separated by a single space. For example, a C_4 quarter note would be displayed as `"C4 (1/4)"`, and an $F_3^{\#}$ eighth note would be `"F#3 (1/8)"`.

PlaySong

Download the file `Tone.java` from Canvas and add it to your project. It contains the following method, which you will use in your program:

```java
/**
 * Sends an audio tone to the speakers
 *
 * @param frequency The frequency of the sound to play, in Hertz
 * @param seconds The duration of the sound, in seconds
 */
public static void playTone(double frequency, double seconds) {
    ...
}
```

Create the class `PlaySong.java` which imports the package `oop.projects.tone.Tone` and uses it to play a song. The class should contain two methods:

```java
public class PlaySong {
    public static void playSong(ArrayList<Note> notes, int tempo) {
        ...
    }

    public static void main(String[] args) {
        ...
    }
}
```

The first takes in an `ArrayList` of notes, and a tempo in beats per minute. Iterate over every note in the given array list, and translate each value into a frequency and duration. Use the note's `getFrequency()` method, and `getDuration()` from the `Beat` enumeration. Pass each pair to `Tone.playTone()` to listen to the list of notes.

In `main`, create an `ArrayList` of `Notes`, and add notes to the list by reading input from the file `song.txt`. The file will contain one note per column, in the following format:

```
...
F#4 (1/4)
D4 (1/4)
E4 (1)
C-1 (1/4)
A4 (1/4)
A4 (1/2)
...
```

The first column of each line uses scientific pitch notation to tell you which note to play. The second column tells you the length of the note in the same format returned by `Beat.toString()`. You can search the `Beat.values()` array to find which `Beat`'s string representation matches the input. For each line in the file, create a new `Note` with the right pitch and length, and add it to the `ArrayList`. Once you've populated the `ArrayList`, use your `playSong()` method to play the song at 120 bpm.

Unit Tests

Please create a test suite to run all of the test classes. Write tests for `Pitch.java`, `Beat.java` and `Note.java`.

- Pitch: Write at least 2 tests for this class that ensures:
  - `Pitch.A.getOffset()` returns 0
  - `Pitch.C.getOffset()` returns -9.
- Beat: Write at least 2 tests for this class that ensures:
  - `getDuration()` with `WHOLE` and a tempo of 120 returns 2.

  - `getDuration()` with `SIXTEENTH` and a tempo of 120 returns 0.125.

- Note: Write a test for each of your constructors that tests:
  - (Constructor 1) A new object is created with C_4 quarter note.
  - (Constructor 2) The maximum octave of the new `Note` object returned is 8 if given `Pitch` is $A^{\#}$.
  - (Constructor 3) The pitch and octave of a valid note is set accordingly.
  - (Constructor 4) The note and octave of a valid midi range is set accordingly.
  - (Constructor 5) The new object sets the correct pitch and octave from nearest MIDI number.

Submission

Make sure that you have Javadoc style comments for every class and method in your source code, as described in Homework 0. Document any unresolved bugs in the Javadoc comments for each of your classes. When you are satisfied that your code is complete, create a TAR file containing all of the source code for this assignment called:

`cse213_<firstname>_<lastname>_hw4.tar.gz`

Upload your submission to Canvas before the due date.

Appendix: SPN and MIDI

Scientific Pitch Notation (SPN) is a scheme used to specify musical notes. SPN only differentiates between 12 distinct notes:

C	$C^{\#}$	D	$D^{\#}$	E	F	$F^{\#}$	G	$G^{\#}$	A	$A^{\#}$	B

The scheme does not include flat notes, since every flat has a corresponding sharp – for example, $D^{\flat}$ is equivalent to $C^{\#}$. In addition to the note, SPN appends a number between -1 and 9 to denote the *octave*.

The two values together uniquely define the pitch, which is a specific frequency of sound. For example, the note A_4 is defined to represent a frequency of exactly $440Hz$, and middle C on a piano, (C_4 in SPN), has a frequency of about $261.63Hz$.

In the age of digital music, there has been a need for a standard way to represent musical notes in computer programs. This is where MIDI [2] comes in. MIDI is a standard that, among many other things, assigns numerical values to the audible range of musical notes. There are 128 MIDI notes; where 0 represents the lowest possible note, (C_{-1}, or $8.18Hz$), and 127 is considered the highest, (G_9, or $12.544kHz$). According to MIDI, the notes $G^{\#}$, A, $A^{\#}$ and B are missing from the 9th octave.

	C	$C^{\#}$	D	$D^{\#}$	E	F	$F^{\#}$	G	$G^{\#}$	A	$A^{\#}$	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127	-	-	-	-

Figure 12: Table of MIDI numbers and their corresponding SPN notes

Knowing the MIDI number, m, it is easy to calculate the frequency of the tone with the following equation:

$$f = 440 \cdot 2^{(m-69)/12}$$

The 440 and 69 come from the established frequency value of A_4. The special ratio $2^{1/12}$ comes up often music theory; it gives this equation the property that moving up one octave *doubles* the frequency, and moving down an octave *halves* the frequency.

[2] Short for Musical Instrument Digital Interface