

Stanford University, CS 106A, Homework Assignment 5

Melody Maker & Image Algorithms (PAIR ASSIGNMENT)

Thanks to Keith Schwarz and Allison Obourn for problem ideas, spec text, and code used in this assignment.

The purpose of this assignment is to practice writing algorithms that use 1-D and 2-D arrays. You will **turn in**:

- **Melody.java** and **mysong.txt**, your solution to Problem 1: Melody Player
- **ImageAlgorithms.java**, your solution to Problem 2: Image Algorithms

Problem 1, Melody Player:

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*.



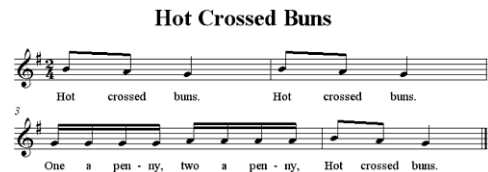
For this program we will be representing notes using scientific pitch notation. This style of notation represents each note as a letter and a number specifying the octave it belongs to. For example, middle C is represented as **C4**. You do not need to understand any more than this about scientific pitch 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. You will also submit a file **mysong.txt** representing a song of your own that you have written in our specified format.

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 at right. The first two lines contain the **title** and **artist**, respectively. The third line is the number of notes in the song; this is the number of lines that will follow in the file. You may assume that every input file is valid and follows this exact format. 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 by instructor):

We have provided you with a class named **Note** that your **Melody** class will use. To access the **Note** class in your code, write the following atop your class: **import stanford.cs106.audio.*;**

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(<i>line</i>)	Constructs a new Note object with data from the given String line, such as " 0.2 C 4 NATURAL false " or " 0.4 R false ".
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(<i>accidental</i>) , setDuration(<i>duration</i>) , setOctave(<i>octave</i>) , setPitch(<i>pitch</i>) , setRepeat(<i>repeat</i>)	Sets aspects of the state of the note based on the given value.
toString()	Returns a text representation of the note.

See the contents of the provided **Note** class documentation provided on the class web site to answer any further questions about how it works. We also provide its source code, though you don't need to paste it into your project.

Melody Class (for you to implement):

You will 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 may not create any other data structures (such as arrays or lists) to help you.

Testing: Test your **Melody** class by running our instructor-provided class **MelodyMain** 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.

Debugging: Since your **Melody** class is not a complete program, you can't use the **println** command for debugging. But there is a similar command named **System.out.println** that can be used from any class; it prints output to a console at the bottom of the Eclipse IDE.

```
public Melody(Scanner input)
```

In this constructor you should populate your melody's array of notes by reading note data from the specified **Scanner**. 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 read each note line, use each line to construct a **Note** object, and put this **Note** into your internal array.

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.

Tip: If the provided GUI's slider doesn't move properly when playing, your total duration code is probably wrong.

```
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, 6, 7, 8, 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.

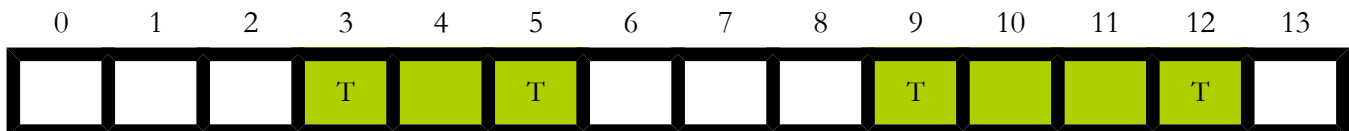


diagram of array of notes with repeated sections at indexes 3-5 and 9-12

```
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 changeDuration(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 are not allowed to call an existing Java 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 **MelodyMain** 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 **MelodyMain** 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. To receive full credit, your **mysong.txt** file should be **different** from the other song files we have provided with the starter ZIP, and it should contain at least **5 notes**, but otherwise it can be anything you want. You can make up a song of your own, or you can make a version of an existing song that you like. If you work as a **pair**, you must each make a song. Turn in **mysong.txt** and **mysong2.txt**; make sure to label the author of each song.

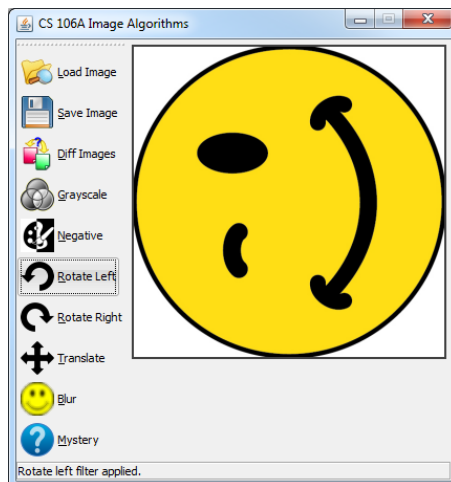
This part is worth a small part of your grade. If you don't want to spend time on this part of the assignment, just make a very simple file that plays a scale of notes or something and turn that in; that will be fine. But we encourage you to **be creative** and submit a neat song! Some especially cool songs may be played in class later on.

Problem 2: Image Algorithms

For the second part of this assignment, you will write a series of algorithms to manipulate the pixels of images. We provide you with an overall graphical user interface `ImageMain.java` and you will write your image manipulation algorithms in a file named `ImageAlgorithms.java`.

As shown in class, we will represent images as 2-dimensional arrays, where each array element is an integer containing the red, green, and blue components of a single pixel. The red, green, and blue components of a pixel are integer values from 0 (none of that color) to 255 (maximum amount of that color).

Each of the algorithms you will write is a method that accepts a `GImage` parameter representing the source image, the current state of the image on the screen. Each image algorithm method should modify the `GImage` passed in by setting its pixel array to contain the new state of the image after that algorithm has been applied to the source image. To interact with `GImage` objects and to get/set pixel integer values, you'll need to use the methods shown in class and in section 11.7 of the *A&S* textbook, such as `getPixelArray`, `setPixelArray`, `getRed`, `getGreen`, `getBlue`, `createRGBPixel`, and so on.



You will implement the following image manipulation algorithms:

- **grayscale:** Converts an image to a black-and-white version of itself
- **negative:** Inverts the colors of an image
- **rotateLeft:** Reorders the pixels to an orientation 90-degrees counterclockwise from their current state
- **rotateRight:** Reorders the pixels to an orientation 90-degrees clockwise from their current state
- **translate:** Moves the positions of the pixels by a given dx , dy offset
- **blur:** Averages pixel values with those of neighboring pixels to produce a "softening" effect
- **mystery:** Any other image manipulation algorithm of your choice (creative aspect)

Each algorithm is described in detail on the following pages. Your `ImageAlgorithms` class should not use any fields; each algorithm can be solved on its own without any fields (instance variables) in your code. Each algorithm should work on an image of any size, including very large images or very small images such as 1x1 pixels, etc. When describing the algorithms we may refer to pixels in the format (r, g, b) , such as $(24, 191, 65)$ to indicate a pixel with a red component of 24, green of 191, and blue of 65.

`public void grayscale(GImage source)`

In this method you should create a grayscale (black-and-white) version of the source image. To convert an image to black-and-white, for each pixel, set all three of its red, green, and blue values to be the average of those three values, rounded down to the nearest integer. For example, the pixel $(10, 52, 36)$ has an average color value of $(10+52+36)/3 = 32.667$, so the grayscale version of that pixel is $(32, 32, 32)$.



`public void negative(GImage source)`


In this method you should create a new image whose pixels are the inverse of those in the source image. To convert an image to its inverse, for each pixel, set all three of its red, green, and blue values to be the inverse of their current color value. The inverse of a color value k is defined as $255 - k$. For example, the pixel $(110, 52, 236)$ has an inverse of $(145, 203, 19)$.





before rotateLeft								after				
	0	1	2	3	4	5			0	1	2	3
0	A	B	C	D	E	F	-->	0	F	L	R	X
1	G	H	I	J	K	L		1	E	K	Q	W
2	M	N	O	P	Q	R		2	D	J	P	V
3	S	T	U	V	W	X		3	C	I	O	U
								4	B	H	N	T
								5	A	G	M	S

before							rotateRight	after							
	0	1	2	3	4	5			0	1	2	3	4	5	
0	A	B	C	D	E	F	-->	0	S	M	G	A	B	C	D
1	G	H	I	J	K	L		1	T	N	H	B	C	D	E
2	M	N	O	P	Q	R		2	U	O	I	C	D	E	F
3	S	T	U	V	W	X		3	V	P	J	D	E	F	G
								4	W	Q	K	E	F	G	H
								5	X	R	L	F	G	H	I



Be careful not to confuse the order and meaning of the two indexes in your 2-D array. The first of the two indexes is the row (y), and the second index is the column (x). `pixels[i][j]` refers to a pixel with ($x=j$, $y=i$).

```

before translate by (dx=2, dy=-1)          after
      0  1  2  3  4  5                  0  1  2  3  4  5
    0  A  B  C  D  E  F                  0  K  L  G  H  I  J
    1  G  H  I  J  K  L          -->      1  Q  R  M  N  O  P
    2  M  N  O  P  Q  R                  2  W  X  S  T  U  V
    3  S  T  U  V  W  X                  3  E  F  A  B  C  D

```


public void blur(GImage source)

In this method you should create a new image whose pixel values are averaged with the values of their immediate neighbors from the source image. The general idea is that for a given pixel (r, c) located at row r and column c in the source image, you will change its red, green, and blue components to be the average of the nine red, green, and blue components in the pixels at locations $(r-1, c-1)$ through $(r+1, c+1)$, rounded down to the nearest integer.



For example, in the diagram below, the pixel (row 1, column 2) should be modified to store the average of the nine pixels (0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), and (2, 2). These are the eight neighbors of (1, 2) as well as (1, 2) itself. So the red part of (1, 2) would be changed from 32 to $(84+74+16+66+32+95+28+47+31)/9 = 52$. The green component would be changed from 67 to $(22+38+17+53+67+65+49+21+41)/9 = 41$. The blue component would be changed from 12 to $(99+69+18+88+12+35+31+94+51)/9 = 55$. Therefore the overall pixel value at (1, 2) in the result image would be (52, 41, 55).

A special case is the set of pixels along the **edges** of the image. When blurring those pixels, they do not have eight neighbors like other pixels do, so the average includes fewer data points. For example, in the diagram below, the pixel at (row 0, column 0) has no neighbors above or left of it, so it should become the average of the four pixels (0, 0), (0, 1), (1, 0), and (1, 1). So the red component of (0, 0) would become $(14+84+21+66)/4 = 46$, and so on. The pixel at (3, 3) has no neighbors below it, so it should become the average of the six pixels (2, 2), (2, 3), (2, 4), (3, 2), (3, 3), and (3, 4). The red component of (3, 3) would become $(47+31+246+15+60+188)/6 = 97$, and so on. Take care that your algorithm does not crash by trying to access outside the bounds of the array.

A common bug in this algorithm is to try to modify the pixel array in-place. You should not do this; you should create a new **second pixel array** to store the result image's pixels. The reason is because you don't want modifications made to one pixel to impact another pixel in the same pass over the array. In our previous example, we already stated that pixel (1, 2) should be changed from (32, 67, 12) to (52, 41, 55). But if you store (52, 41, 55) into this pixel and then use that value for further calculations on pixels in the same pass over the array, their averages will be incorrect. For example, when computing the average for pixel (1, 3), the pixel (1, 2) is one of its neighbors. But you should use that pixel's original value of (32, 67, 12) when computing that average.

	0	1	2	3	4
0	(14, 97, 63)	(84, 22, 99)	(74, 38, 69)	(16, 17, 18)	(85, 75, 75)
1	(21, 18, 45)	(66, 53, 88)	(32, 67, 12)	(95, 65, 35)	(6, 0, 2)
2	(37, 29, 61)	(28, 49, 31)	(47, 21, 94)	(31, 41, 51)	(246, 84, 13)
3	(82, 33, 90)	(42, 43, 44)	(15, 80, 50)	(60, 40, 12)	(188, 45, 1)

public void mystery(GImage source) *(creative aspect)*

After implementing the other image algorithms, you should come up with one more algorithm of your choice named **mystery** that does **anything you want**. This algorithm can do anything to the source image as long as it is not essentially the same as one of the existing image algorithms in this assignment or shown in lecture/section. Your algorithm should make some kind of visible change to the source image on the screen, but beyond that it can do anything you want. For example, you could zoom the image, or make funny changes to its colors, or sharpen it, or any number of interesting manipulations. You might want to search the web for ideas of interesting algorithms that can be performed on pixels of images. In your comments, explain what your **mystery** method is doing so that it is clear to the grader.



This method is worth a very small part of your grade and is meant to give you an outlet to **be creative**, but if you don't want to spend much time on it, just make a simple change to the image and that will be fine.

If you want to add more than one **mystery** method, our provided code will notice any methods you write with names like **mystery1**, **mystery2**, etc. and provide buttons for all of them in the main GUI. You must still have a method whose name is exactly "**mystery**", but any others whose names start with "**mystery**" will be used.

Extra Features: (Extra features are optional and will earn you a small amount of extra credit if you complete them.)

There are many possibilities for extra features that you can add if you like. If you are going to do extra features, **submit two versions** of the assignment: the basic Java file that meets all the normal assignment requirements, and an "extra" file such as **MelodyExtra.java** or **ImageAlgorithmsExtra.java** containing your extended version. (If your extra features don't break or change the base functionality in any way, you can just put the extra features in the existing **Melody.java** or **ImageAlgorithms.java** file.) At the top of your .java file in its comment header, you must **comment** what extra features you completed. Here are a few ideas for extra features:

- *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?
- *Ability to visually compose a song:* Provide a user interface where the user can click piano keys to input a song.
- *Multiple image mystery methods:* Write several neat image **mystery** manipulations that do interesting things.
- *Paint program:* Modify our provided **ImageMain** to have drawing tools like paint brush, line drawing, fill, etc.
- *Other:* Use your imagination. What other features could you imagine in a program like this?

Grading

Functionality: Your code should compile without any errors or warnings. For Problem 1, you can listen to the song to verify it, and you can also check Eclipse's console to see text output of each note as it is played. For Problem 2, we use an Image Comparison Tool to see that your image *exactly* matches the one expected. You can use the same Image Comparison Tool with expected output images from the class web site to verify your program's behavior.

Style: Follow style guidelines taught in class and listed in the course **Style Guide**. For example, 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, next to each field, and a **citation of all sources** you used to help write your program. If you complete any **extra features**, list them in your comments to make sure the grader knows what you completed. In general, limit yourself to using Java syntax taught in lecture and textbook parts we have read so far.

Fields: As mentioned earlier, in Problem 2 you should not use any **fields** (a.k.a. private instance variables). In Problem 1, fields are allowed and necessary, but you should still 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 you 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**.

Procedural decomposition: Most of the methods you'll need to write on this assignment are already specified for you. But you should still 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.

Honor Code: Follow the **Honor Code** when working on this assignment. Submit your own (pair's) work and do not look at others' solutions nor give out your solution. Do not place a solution to this assignment on a public web site or forum. Solutions from this quarter, past quarters, and any solutions found online, will be electronically compared.