CS 211, ALL SECTIONS

PROJECT 3

DUE SUNDAY, MARCH 31TH AT MIDNIGHT

The objective of this project is to implement a multimedia player which can load and playback different types of multimedia files (in this case only still images and audio data, and output only to a text terminal), in order to understand and exercise the use of interfaces and exceptions.

OVERVIEW:

1. Download and review the provided interfaces, `Loadable`, `Player`, `StillImage`, and `AudioStream`.
2. Implement the exception type `LoadException` which will be used to signal errors while loading multimedia files.
3. Implement the class `LoadableImage` which would allow loading a still image from a text file.
4. Implement the abstract class `ImageViewer` and concrete subclass `TextImageViewer`, which allows a `StillImage` to be viewed in a terminal using text art. Test your viewer with the provided sample images.
5. Implement the class `LoadableAudio` which would allow loading an audio stream from a text file.
6. Implement the abstract class `AudioPlayer` and concrete subclass `TextAudioPlayer`, which allows an `AudioStream` to be viewed as a wave output in a terminal using text art. Test your viewer with the provided sample.
7. Students in the honors section will have an additional subclass and functionality to complete for full credit.
8. Download and use the tester module to ensure that your program is correct.
9. Prepare the assignment for submission and submit it.

A multimedia player is a piece of software which needs no introduction. We're all likely to have used one to do anything from playing CDs or DVDs, to audio or video files. One thing we might notice is that when we do this, we don't have to think too hard about what kind of file we're clicking on. We just click on it, and then it opens up in the player, whatever the format and whatever the type of media it is (because if we think about it, images and sound are two different things). This is something we can do thanks in part to object-oriented programming. When we select a file, we don't have to care too much about what it is, so long as it's something which is loadable by the player. And whatever kind of media it is, the player figures out the right thing to do with it.

For this project, we will write a simplified version of this sort of interface. We will write our own multimedia player, which is capable of playing different kind of media files. We will of course make a few simplifications to fit this project within a two week window, but it will still be enough to demonstrate the mechanism. For example, we will only be implementing an *audio* and *still image* player; we will use our own text-only media file formats; and our output will all be done visually on the terminal using text art.

When we implement this project, we will practice two main language features: interfaces and exceptions. We will see the power of interfaces to abstractly define the types of things our classes will do, which leaves us free to perform the tricks we were talking about earlier - being able to load files without worrying too closely about the type, and being able to play a media type without knowing in advance what kind of media we're dealing with. We also see how exceptions are useful in reporting errors without disrupting the flow of code.

RULES

1. This project is an individual effort; the Honor Code applies
2. You may import Java built-in libraries, although avoid doing so unless necessary.
3. You may add your own helper methods or data fields, but they should be private or protected. You may add additional constructors which are public.

INSTRUCTIONS: Begin by downloading the provided interfaces, `Loadable`, `Player`, `StillImage`, and `AudioStream`.

- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/Loadable.java
- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/Player.java
- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/StillImage.java
- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/AudioStream.java

The interfaces can be described as follows:

- `Loadable`: a media format which can be loaded. We're assuming that we've read the contents of a file into an array of integers already, and are passing that integer data into this interface to try to load the media. Any class which implements this interface would need to be able to do two things: to `match` the data to the format (in other words, to check the format of the input data, and see if it matches this media format); and to `load` the data and use it to create a new instance of this loadable. We can see an example of how it could be used below:

```
Loadable xyzImageLoader = new XyzImageLoader();
int[] data1 = mediaPlayer.read("image1.xyz");
int[] data2 = mediaPlayer.read("image2.xyz");
Loadable img1 = null, img2 = null;
```

```
if (xyzImageLoader.match(data1)) img1 = xyzImageLoader.load(data1);
if (xyzImageLoader.match(data2)) img2 = xyzImageLoader.load(data2);
```

- `Player`: once we've loaded some piece of media, we'd like to be able to play it. This class describes some kind of media player (although the kind of player isn't specified - it could be audio, for instance, but it could be something else). Classes which implement this interface are ready to play media which has been loaded already. Any such class would need to be able to do the following two things: to check if the player `canPlay` the loaded media; and also to actually `play` the media. For example:

```
Loadable media = mediaPlayer.load("media_file.fmt");
Player videoPlayer = new VideoPlayer();
Player audioPlayer = new SystemAudioPlayer();
if (videoPlayer.canPlay(media)) videoPlayer.play(media);
else if (audioPlayer.canPlay(media)) audioPlayer.play(media);
```

- `StillImage`: this represents an ordinary, still image, like something you'd take with your camera. It has three methods: its `width` in pixels; its `height` in pixels; and a `getPixel` which gets the value of the pixel at the chosen coordinates. The coordinates `(0, 0)` represent the value of the pixel in the upper left of the image; the `x` coordinate goes from left to right, and the `y` coordinate goes from top to bottom. The pixel value returned by `getPixel` is a number which we assume to be in the range of brightness from 0 to 999, where 0 is black and 999 is white.
- `AudioStream`: this represents a stream of audio data. The audio data itself will be a stream of numbers representing the amplitude of the audio wave; we will assume that it ranges from -999 to 999. The interface has three methods: a `freq` method which indicates the playback frequency of the audio (since we will do text output in this project instead of true audio, this method won't have much practical effect, but we still include it for completeness); a `next` method to retreive the next point from the audio wave; and a `hasNext` method to tell us if we've reached the end of the audio data. Since streaming data is known for updating on the fly, we use `next`/`hasNext` to read the next data point, instead of using some kind of `getValueAt` to retrieve audio from anywhere in the sound wave.

Now that we know what each of these interfaces are for, let's use them to write some classes.

LoadException Task:
`public class LoadException`, which is an `Exception`

(*5pts*) We will use this class when we need to throw an exception to indicate a problem loading a media file. We would only need to have a constructor:

- `public LoadException(String msg)` initializes the exception using the given error message (use the `msg` to initialize the parent's constructor).

LoadableImage Task:
`public class LoadableImage`, which is both a `Loadable` and a `StillImage`

(*15pts*) We know what a loadable is and we know what an image is. Let's put them together to make an image which we can load. Our input, like with any media format in this project, will be an array of integers. In this array, a `LoadableImage` would be indicated by the number `55` appearing in the first spot in the array (we chose this number arbitrarily - don't expect a deeper reason). What does this mean? If the first number in the input is a 55, then that means that the input represents a `LoadableImage`. If it is not, then that means that the input is something different - whatever it is, it is not a `LoadableImage`. This idea is called a *magic number*. Most real-world file formats use a special value or combination of values at the beginning of the file to indicate what file type it is (for example, every `.pdf` file begins with the characters `%PDF` - you can check this yourself).

When loading the image from the data, the two numbers after the first spot would represent the width and the height, respectively. Every position after that represents the pixel value at a particular position, going from left to right and top to bottom. Thus, if we have the following input data, the resulting image would be a 5x5 "X" pattern:

```
55 5 5 999 0 0 0 999 0 999 0 999 0 0 0 999 0 0 0 999 0 999 0 999 0 0 0 999
```

Our class will need to provide functionality to load an image, as well as to access the image data once it has been loaded. Our class is actually doing double-duty: we can use it to represent a loader for image data, as well as to represent the image once it has been loaded. To do this, we need the following methods:

- `public LoadableImage()` a default constructor.
- `public LoadableImage(int w, int h)` (*not required but recommended*) initialize a `LoadableImage` with the given width and height.
- `public boolean matches(int[] data)` indicates whether the provided input data matches a loadable image format. That is, it returns true if the first element of `data` is `55`, and false if it is not, or if there is no first element of `data`.
- `public LoadableImage load(int[] data) throws LoadException` creates a new `LoadableImage`, loads it with the given data, and returns the result. If the input is not well-formed (meaning that there is no width or height elements in the input, or if there isn't

enough pixel data to match the dimensions of the picture, or if the pixel data is outside of the bounds 0-999), then this method should throw a `LoadException`. If the input data is ok, then this class should construct a new `LoadableImage` with the dimensions specified by the input data. Then, it should intialize the pixel data in the new `LoadableImage` using the input data. Finally, it should return the resulting image.
Tip: it is ok to access private methods from within a new `LoadableImage` here, because it is the same class type as this class.

- `public int width()` returns the width of a loaded image, in pixels. There is no specified behavior if this is called without loading an image first (there is no wrong answer in that case).
- `public int height()` returns the height of a loaded image, in pixels. There is no specified behavior if this is called without loading an image first (there is no wrong answer in that case).
- `public int getPixel(int x, int y)` returns the pixel value at coordinates `(x, y)` in the image. If someone asks for an out-of-bounds value, it's ok to throw an exception (which is likely what would happen by default). There is no specified behavior if this is called without loading an image first (there is no wrong answer in that case).

ImageViewer Task:
`public abstract class ImageViewer`, which is a `Player`

(*10pts*) This class provides the abstract viewer functionality used to "play" still images (when we say this, we really mean display the images). This player represents the basis for any image viewer used by the multimedia player, however, it has an abstract viewing method which does not restrict the way it displays images. This class should provide the following:

- `public ImageViewer()` a default constructor (this shouldn't need to do much if anything).
- `public abstract void view(StillImage img)` yet-to-be-defined functionality for viewing images.
- `public boolean canPlay(Loadable l)` indicates whether the provided loadable is playable by this player. It should return `true` if the input `l` is a `StillImage`, and `false` if it is not.
- `public void play(Loadable l)` plays (views) the input loadable `l`, using a call to the `view` method.

TextImageViewer Task:
`public class TextImageViewer`, which is an `ImageViewer`

(*10pts*) Now we'll write our first actual, working media viewer. This one will work by representing pixels as text art (the brighter the pixel, the larger the corresponding character which is printed, assuming a white-on-black screen). So for every line of pixels, we would print out the string of characters which correspond to each pixel in that row. It might give us an output such as this (using the "X" picture data descibed above in the `LoadableImage` section):

```
@   @
 @ @
  @
 @ @
@   @
```

We would only need three methods in this class, one of which will be given to you to ensure consistency across implementations:

- `public TextImageViewer()` a default constructor (it might not be necessary to do much here).
- `public char getChar(int i)` this will retreive the text character corresponding to a given pixel value. To avoid inconsistency, use the following implementation:
  ```
  private final char[] vals = {' ', '.', 'o', 'O', '@'};
  public char getChar(int i) {
    return vals[(int)( i*(vals.length/1000.0) )];
  }
  ```
- `public void view(StillImage img)` for the given input image, print out the characters corresponding to its pixels data as described above. Use the `getChar` method to get the character to print for each pixel in the image.

LoadableAudio Task:
`public class LoadableAudio`, which is both a `Loadable` and an `AudioStream`

(*15pts*) We have already written a `LoadableImage` class. This one follows a similar idea, but is used for audio data. We will assume a simple audio model here: a single channel, represented by a sequence of amplitude values (numbers ranging from -999 to 999) which form the audio wave. Like the `LoadableImage`, we will initialize our object using data from the input. However, when we read back the audio data, we will read it back one data point at a time using a `next()` method, so we will need some kind of variable to keep track of how far we've gotten so far.

The input data format is straightforward. The first three numbers must be `3`, `2`, and `1`, in that order, to be a `LoadableAudio` file. Otherwise, it must be some different format. The fourth number in the input data is a playback frequency value (which we would need to store, but has no other practical use in this project, because the playback is sent to the terminal). Everything after that is the audio wave. We need the following:

- `public LoadableAudio()` a default constructor.
- `public LoadableAudio(int frequency, int size)` (*not required, but recommended*) initializes the object using the given frequence and audio data size (the number of data points of audio amplitude data).
- `public boolean matches(int[] data)` returns `true` if the first three elements of `data` are `3`, `2`, and `1`, in that order, and `false` otherwise, or if there are less than three elements of `data`.
- `public LoadableAudio load(int[] data) throws LoadException` creates a new `LoadableAudio`, loads it with the given data, and returns the result. If the input is not well-formed (meaning that there is no frequency element in the input, or if the amplitude data is outside of the bounds -999-999), then this method should throw a `LoadException`. If the input data is ok, then this class should construct a new `LoadableAudio`. Then, it should intialize the audio data in the new `LoadableAudio` using the input data. Finally, it should return the resulting object.
- `public int freq()` returns the playback frequency of the loaded audio data.
- `public int next()` returns the next element of audio data from the current playback.
- `public boolean hasNext()` returns `true` for as long as there is still data left to play (i.e. for as long as the `next()` call has not stepped through all of the audio data yet).

AudioPlayer Task:
`public abstract class AudioPlayer`, which is a `Player`

(*10pts*) Like the abstract `ImageViewer`, this class provides the abstract player functionality used to play audio files. This player represents the basis for any audio player which is incorporated into the multimedia player, however, it has an abstract playback method which does not restrict the way it plays audio. This class should provide the following:

- `public AudioPlayer()` a default constructor (this shouldn't need to do much if anything).
- `public abstract void playback(AudioStream aud)` yet-to-be-defined functionality for playing audio.
- `public boolean canPlay(Loadable l)` indicates whether the provided loadable is playable by this player. It should return `true` if the input `l` is an `AudioStream`, and `false` if it is not.
- `public void play(Loadable l)` plays the input loadable `l`, using a call to the `playback` method.

TextAudioPlayer Task:
`public class TextAudioPlayer`, which is an `AudioPlayer`

(*10pts*) Just like the text-based image viewer, we will now write a text-based audio player. How do we do this for audio? By displaying the output sound wave on the terminal, much like an oscilloscope. We have a stream of audio data, we just need figure out where each point of that data fits on a line of the terminal, and print out that point one line at a time. Our output might look something like this:

```
*
*
*
 *
  *
    *
     *
      *
      *
      *
     *
    *
  *
 *
*
*
```

We would only need two methods in this class:

- `public TextAudioPlayer()` a default constructor (it might not be necessary to do much here).
- `public void playback(AudioStream aud)` the given input audio will come as a stream of numbers representing audio amplitudes. For each data point, print out a line containing a `"*"`, with the position of the asterisk depending on the amplitude of the data point. Do this until the audio data has been exhausted (until `hasNext()` is `false`). The position of the asterisk on the terminal line should be given by the following expression (where a zero corresponds to the first position in the line):
  ```
  int pos = (val+1000)*7/200;
  ```

MultimediaPlayer Task:

`public class MultimediaPlayer`, which is both a `Player` and a `Loadable`

(*25pts*) This is our big one, where we put it all together. This class will aggregate several `Loadable` classes, which allows it to load a media file. We will use this to make the `MultimediaPlayer` `Loadable` itself. Likewise, the class will aggregate several `Player`s, which will allow us to make this a `Player` itself. We will want to implement it from its parts:

- `public MultimediaPlayer()` a default constructor. This class should internally contain both a `TextImageViewer` and a `TextAudioPlayer` for image and audio playing. Additionally, it should internally contain the `LoadableImage` and `LoadableAudio` loaders (and for the honors section, the `CompressedImage` loader as well).
- `public void add(Loadable l)` add an additional media loader to the aggregate.
- `public boolean canPlay(Loadable l)` indicates whether the multimedia player can play the input `Loadable`, by checking whether any of its internal `Player`s `canPlay` the media.
- `public void play(Loadable l)` play back the input media, by searching the internal `Player`s and finding one which supports the `Loadable`. If one is found, then use it to play the input media, otherwise do nothing.
- `public int[] read(String filename) throws LoadException, IOException` reads the text data contained in the file and return the result as an array of integers. We are assuming that the input data is a series of numbers written in human-readable text. If the data is not in that format (if it contains tokens which are not integers), then this method should throw a `LoadException`. To read numbers from a file, we can use an ordinary `Scanner` which we've opened as follows:

  ```
  Scanner s = new Scanner(new File(filename));
  ```

  Tip 1: When we use a `Scanner` for reading a file, we have to remember to call `close()` on it when we are done, as well as to import `java.util.Scanner`, `java.io.IOException`, and `java.io.File`.
  Tip 2: It may help to pass through the file twice: the first time to count the number of tokens in the file and validate proper format, and the second time to actually read in and store the data.
- `public boolean matches(int[] data)` indicates whether the multimedia player is able to load the input media data, by checking whether any of the internal loaders recognize the format.
- `public Loadable load(int[] data) throws LoadException` loads the input media, by looking through all of the internal loaders and finding one which recognizes the format. If none are found, then this method should throw a `LoadException`.
- `public void play(String filename) throws LoadException, IOException` plays back the media file specified by the given filename. This would most likely be done in three phases: first, `read` the file into an an array; second, use the array to `load` the media; third, `play` the resulting `Loadable`.

Some things to think about when implementing this player:

- If we wanted to support more data types, how would that work? Would it make a difference if they were a variation on an existing type?
- What do we do if we want to make a player for another media type (such as video)? Since video is both audio and visual, is there a way to show just one or the other? What if we wanted to make a media format more complex (like adding color to images, or multiple channels to audio)?
- What do we do if we have more than one player for the same type of media (like both a text output and a graphical output)? How would we select between the two?

Honors Section:

If you are in the honors section, you must complete this part and it is worth 20 points of the project grade. If you are not in the honors section, you are welcome to attempt this but you do not need to complete it and it is not worth any points if you do.

`public class CompressedImage` which is a `LoadableImage`

The `CompressedImage` class is nearly the same as its parent, but it needs to override the `matches` and `load` methods. This class will match an input data format if the first number in the input is `56`, not `55` like with the parent. Additionally, the input image format will use a simple form of compression: after every pixel value there is a number indicating how many times that pixel repeats. So for example, the following two inputs produce the same image:

```
55 3 3 0 0 0 0 0 999 999 999 999
```

```
56 3 3 0 5 999 4
```

Additionally, you will be required to write a `main` method which recognizes command line parameters. Each command line parameter will represent one media file (of any supported type). These should be loaded and played back in order by your media player. If any media file results in a thrown exception, the program should print the following (where `FILENAME` is the name of the file which failed) and skip to the next file:

```
Error playing FILENAME
```

Testing:

- https://mason.gmu.edu/~iavramo2/classes/cs211/junit-cs211.jar
- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/P3tester.java
- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/P3testdata.zip
- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/Converter.java

The files in the test data zip should be uncompressed into your working directory.

The `Converter` program is an optional utility which will allow you to take an existing image and convert into the format used for this project, for the sake of testing.

Submission:

Submission instructions are as follows.

1. Let *xxx* be your lab section number, and let *yyyyyyyy* be your GMU userid. Create the directory `xxx_yyyyyyyy_P3/`
2. Place your files in the directory you've just created.
3. Create the file `ID.txt` in the format shown below, containing your name, userid, G#, lecture section and lab section, and add it to the directory.

   Full Name: Donald Knuth
   userID: dknuth
   G#: 00123456
   Lecture section: 004
   Lab section: 213

4. compress the folder and its contents into a .zip file, and upload the file to Blackboard.