J*MUTunes*: A Digital Audio Player
Software Requirements Specification
Version 2.02

1. Introduction

1.1. Objectives

1. This assignment has six (6) objectives:
   - To continue the development of the *JMUTunes* application by adding functionality to that provided in PA1;
   - To give you practice in "applied abstraction" by replacing the internal structure used by the **`AudioList`.**
   - To give you practice at implementing a *Set* using an **`ArrayList`**
   - To give you practice in placing **`AudioFile`** objects into alphabetical order within the **`AudioList`**;
   - To give you practice in controlling user input and handling errors and exceptions;
   - To give you practice in reading and writing data files.

1.2. Deliverables Overview

1. *JMUTunes* is a program designed to manage digital audio files. Digital audio files can be either **`mp3`** (compressed format), **`m4a`** (compressed format), or **`wav`** (uncompressed format). The program can be extended to allow for other types of audio files, but decoders for some other audio types are not readily available for JAVA.

2. The program will allow the creation of a data store for digital audio files (see below, under "Non-functional Requirements"). The user will be able to add audio titles, edit audio titles, delete audio titles, search for audio titles, and play audio files.

3. The final version of the *JMUTunes* Audio Manager will allow the user to play audio for supported audio formats as well.

1.3. Operating Environment

1. *JMUTunes* will run under under Windows, Linux, and Mac OS.

2. *JMUTunes* will run using the Sun Java SDK 1.8 or higher.

2. Functional Requirements (Version 1)

2.1. Product Services

1. JMUTunes will offer the capability for storing and displaying information about digital audio files.

2. The program will allow the user to enter information for the artist and title of a digital audio file and optionally the album and track number for the digital audio file.

3. The program will store the information in an **`AudioFile`** object in a **`AudioList`** class that is implemented using an **`ArrayList`** that functions as a Set.

4. The program will store the **`AudioFile`**s in the **`AudioList`** in alphabetical order by Artist/Album/Track/Title.

5. The program will allow the user to edit the information for any **`AudioFile`** in the **`AudioList`**.

6. The program will allow the user to delete any **`AudioFile`** from the **`AudioList`**.

7. The program will populate the **`AudioList`** with data read from a file upon program start and write to the file when the program ends.

2.2. Error Handling

1. The program will ensure that all user input is in range for any prompt. For example, if the user enters a value for a track number, that number must be an **`int`** greater than 0 and less than 100. If the user enters a value for the operation to be performed (see below), that entry must be a **`String`** with a length of one (1) and must be one of the following: “A/a”, “E/e”, “D/d”, “S/s”, “P/p”, “N/n”, “B/b”, “or Q/q” (see the “User Interface Specifications” below). If a particular data item is required, the program will ensure that a value is entered.

2. Methods that accept parameters will ensure that invalid values are handled gracefully. For example, a method that accepts an object as a parameter should test that the object is not **`null`** before continuing. A method that requires an **`int`** that is greater than 0 should ensure that values less than 1 are not accepted. You can generally handle these by enclosing the method within an **`if`** clause that tests for valid input. For example:

```
public void someMethod( String s ) {

    if ( s != null ) {

        // rest of method
    }
}
```

or

```
public void someMethod( String s ) {

    if ( isValidInput( s ) ) {

        // rest of method
    }
}
```

2.3. User Interface Specifications

1. The user interface will adapt its spacing to that of a standard character terminal (25 lines x 80 columns).

2. In all cases where the user is asked to enter information following a prompt, the user's input must be on the same line as the prompt (and not on the line below).

3. When the program begins, the following screen will display:

```
                          JMUTunes Audio Player
                            CS159 (Fall 2018)
                               <blank line>
                               <blank line>
 1.
 2.
 3.
 4.
 5.
 6.
 7.
 8.
 9.
10.
11.
12.
13.
14.
15.
16.
                               <blank line>
     (A)dd, (E)dit, (D)elete, (S)earch, (P)lay, (N)ext, (B)ack, (Q)uit.
                               <blank line>
     Choose Operation ->
```

4. The first two lines will be centered at the top of the screen. These will then be followed by 2 blank lines. The next 16 lines should be numbered from 1 to 16 as shown above. All should be followed by a period, and the periods should be vertically aligned. This will be followed by another blank line followed by list of the possible operations that can be performed. This list of operations should be centered. Note: parentheses are used to designate the letter to be used to perform a particular operation. This, in turn, is followed by another blank line which is followed by the user's prompt ("Choose Operation"). The user's prompt should be aligned with the list of operations above. Make sure to include a single space following the prompt ("-> ") so that the user entry does not abut against the prompt.

5. The user must choose one of the letters enclosed by parenthesis in the screen above. If the user enters an incorrect value, the program should print an error message along with the incorrect entry and and force the user to re-enter the value using the same prompt as above. For example, if the user enters the number "3" instead of one of the allowable choices, the output will appear like this:

```
Choose Operation -> 3
<blank line>
Incorrect entry (3).
Choose Operation ->
```

This error message and re-prompt should be indented at the same level as the prompt above and should continue to appear so long as the user persists in entering incorrect data.

6. If the user chooses "A" or "a" (for "Add"), the program will print 25 blank lines to the screen (to clear the original screen) and prompt the user the enter the artist, song title, album, and track number for the musical composition. The add operation should begin with a title followed by two (2) blank lines followed a prompt for the artist:

```
                          Add Audio File
                           <blank line>
                           <blank line>
Artist:
```

"Artist" is a required field. If the user presses **<ENTER/RETURN>** without entering a value (or just whitespace), the program should print an error message and force the user to re-enter the information as follows:

```
                                    <blank line>
Required entry.
Artist:
```

This error message and re-prompt should continue to appear so long as the user persists in entering incorrect data.

Once the artist has been entered, the program should prompt for the title:

```
                                Add Audio File
                                 <blank line>
                                 <blank line>
Artist: Beatles
Title:
```

The title should be vertically aligned with the artist. “Title” is also a required field. If the user presses **<ENTER/RETURN>** without entering a value (or just whitespace), the program should print an error message and force the user to re-enter the information as follows:

```
                                    <blank line>
Required entry.
Title:
```

This error message and re-prompt should continue to appear so long as the user persists in entering incorrect data.

Once the user has entered the title, the program should prompt for the album and then the track number. Both album and track number are optional fields — that is, the user can choose to enter nothing here by pressing the **<ENTER/RETURN>** key. If the user does enter a track number, then it must be an **int** greater than 0 and less than 100 (1–99).

```
                                Add Audio File
                                 <blank line>
                                 <blank line>
Artist:
                                 <blank line>
Required Entry.
Artist: Beatles
Title:  I Saw Her Standing There
Album:  Meet the Beatles
Track:  2
```

If the user enters a value that is out of range for the track number, then the program should print an error message and force the user to re-enter the value as follows:

```
                                <blank line>
Must be a number between 1 and 99.
Track:
```

This error message and re-prompt should continue to appear so long as the user persists in entering incorrect data.

Once all information has been entered, the program should print a blank line followed by a prompt asking the user to confirm (or deny) the addition:

```
                              Add Audio File
                               <blank line>
                               <blank line>
Artist:
                               <blank line>
Required Entry.
Artist: Beatles
Title
                               <blank line>
Required Entry.
Title:  I Saw Her Standing There
Album:  Meet the Beatles
Track:  0
                               <blank line>
Must be a number between 1 and 99.
Track:  2
                                <blank line>
Add (Y/N):
```

If the user chooses “Y” or “y”, the program should add the record to the **`AudioList`** (see below, under “Non-functional Requirements”). If the user chooses “N” or “n”, the program should not add the record to the **`AudioList`**. If the user enters anything other than “Y/y” or “N/n”, the program should print an error message and re-prompt the user for the correct input as follows:

```
                                <blank line>
Must be 'Y' or 'N'.
Add (Y/N):
```

This error message and re-prompt should continue to appear so long as the user persists in entering incorrect data.

The program should then print 25 blank lines to the screen (to clear the original screen) and return to the opening screen, although with the added **`AudioFile`** showing in the list.

```
                          JMUTunes Audio Player
                            CS159 (Fall 2018)
                              <blank line>
                              <blank line>
 1. Beatles, I Saw Her Standing There (Meet the Beatles)
 2.
 3.
 4.
 5.
 6.
 7.
 8.
 9.
10.
11.
12.
13.
14.
15.
16.
                              <blank line>
     (A)dd, (E)dit, (D)elete, (S)earch, (P)lay, (N)ext, (B)ack, (Q)uit.
                              <blank line>
     Choose Operation ->
```

If there is no album, then you should omit the parentheses. For example, if the above example did not include an album, then the line would read:

```
Beatles, I Saw Her Standing There
```

If the text is too wide for the screen (assume 80 characters for the screen width less whatever space is taken at the left side), then you should truncate the entry and conclude with 3 dots ("…"). The String for "artist, title (album)" should not exceed a maximum length of 72 characters. If the String is longer than this, then use the first 69 characters followed by 3 dots. For example:

```
Weird Al Yankovich, Some Really Bizarre Song by Weird Al (Weird Al's...
```

7. **Edit**. If the user chooses "E" or "e" (for "Edit"), the program should ask the user to indicate the number of the **`AudioFile`** to edit. This prompt should appear on the line directly following the line "**`Choose Operation ->`** " and should be indented to the same level. The number entered must match an entry on the current page (the number must be valid). That is, the number should reference a **`AudioFile`** object that exists in that position in the **`AudioList`**.

```
Choose Operation -> e
Edit Audio File number ->
```

If the user enters a number that is invalid, the program should report the following error and re-prompt the user to enter a correct file number (both indented to the same level as the text above).

```
                                    <blank line>
Must be a number between <first> and <last>.
Edit Audio File number ->
```

Note: `<first>` represents the first number on the screen (1, 17, 33, etc) and `<last>` represents the last number containing an **`AudioFile`** (not necessarily the last line on the screen). This error message and re-prompt should continue to appear so long as the user persists in entering an invalid number.

Once the user has entered a valid number that corresponds to a **`AudioFile`** object in the **`AudioList`**, the program should print 25 blank lines to the screen (to clear the original screen) and ask the user to confirm the item to be edited. The prompt should include artist, title (album) with surrounding quotation marks. If the line is too long for the space (60 characters max), then truncate as above, being sure to leave enough space at the end of the line for a Y/N answer::

```
                                  Edit Audio File
                                    <blank line>
                                    <blank line>
Edit “Beatles, I Saw Her Standing There (Meet the Beatles)”? (Y/N) ->
```

If the user enters anything other than “Y/y” or “N/n”, the program should print an error message and re-prompt the user for the correct input as follows:

```
                                    <blank line>
<blank line>
Must be 'Y' or 'N'.
Edit “Beatles, I Saw Her Standing There (Meet the Beatles)”? (Y/N) ->
```

If the user specifies “N” or “n”, the program should print 25 blank lines and return to the opening screen. If the user specifies “Y” or “y”, the program should print 25 blank lines and prompt the user to edit the artist, title, album, and track for the **`AudioFile`** as shown below. The edit operation should mirror the add operation, beginning with the title for the function (“Edit Audio File”) followed by two (2) blank lines followed a prompt for the artist. Each item for edit should include the original value in parentheses and a prompt on the line below to allow the user to change a value for an item. The prompt should be indented 2 spaces and should say (**`New Value ->`**). If the user presses **`<ENTER>`** for any given item, the original value will remain unchanged. If the user enters any other text, that value will replace the

value originally stored. The following example models the edit operation for the item added above:

```
                              Edit Audio File
                                <blank line>
                                <blank line>
Artist (Beatles)
  New Value ->
Title (I Saw Him Standing There)
  New Value -> I Saw Her Standing There
Album (Meet the Stones)
  New Value -> Meet the Beatles
Track (N/A)
  New Value -> 200
                                <blank line>
Must be a number between 1 and 99.
  New Value -> 2
                                <blank line>
Confirm Edit (Y/N):
```

In the above example, the artist remains unchanged, while the title, album, and track will have changed. If the user attempts to clear out a required item (such as Author or Title—by entering a space, for example) or enter an invalid value (as in track), the program should respond as in Add when a user attempts to skip a required entry (see above).

If the user chooses "Y" or "y", the program should record the changes in the **`AudioFile`** object. If the changes made to the **`AudioFile`** should change its position in the **`AudioList`**, the **`AudioList`** should be adjusted to reflect this change. If the user chooses "N" or "n", the **`AudioFile`** object should remain unaltered. If the user enters anything other than "Y/y" or "N/n", the program should print an error message and re-prompt the user for the correct input as follows:

```
                                <blank line>
Must be 'Y' or 'N'.
Confirm Edit (Y/N):
```

This error message and re-prompt should continue to appear so long as the user persists in entering incorrect data.

Once the user correctly enters "Yy" or "Nn", the program should print 25 blank lines to the screen (to clear the original screen) and return to the opening screen, although with the edited **`AudioFile`** showing up in its correct place in the list.

If there are no **`AudioFile`**s to edit (the list is empty), the program should print 25 blank lines to the screen (to clear the original screen) and then print the following:

```
                                Edit Audio File
                                  <blank line>
                                  <blank line>
Nothing to edit.
Press <ENTER/RETURN> to continue . . .
```

Once the user presses the **<ENTER/RETIRM>** key, the program should print 25 blank lines to the screen (to clear the screen) and return to the opening screen.

8. **Delete.** If the user chooses "D" or "d" (for "Delete"), the program should ask the user to indicate the number of the **AudioFile** to delete. This prompt should appear on the line directly following the line "**Choose Operation ->** " and should be indented to the same level. The number entered must match an entry on the current page (the number must be valid). That is, the number should reference an **AudioFile** object that exists in that position in the **AudioList**.

```
Choose Operation -> d
Delete Audio File number ->
```

If the user enters a number that is invalid, the program should report the following error and re-prompt the user to enter a correct **AudioFile** number.

```
                                  <blank line>
Must be a number between <first> and <last>.
Delete Audio File number ->
```

Note: `<first>` represents the first number on the screen (1, 17, 33, etc) and `<last>` represents the last number containing an **AudioFile** (not necessarily the last line on the screen). This error message and re-prompt should continue to appear so long as the user persists in entering an invalid number. Once a correct number has been entered, the program should print 25 blank lines to the screen, followed by the following title, blank lines, and validation message (as in PA1). The prompt should include the information presented in the opening screen with surrounding quotation marks. If the line is too long for the space (60 characters max), then truncate as above (ending with ". . ."), so that you leave enough space at the end of the line for a Y/N answer:

```
                                Delete Audio File
                                  <blank line>
                                  <blank line>
Delete "Beatles, I Saw Her Standing There (Meet the Beatles)"? (Y/N) ->
```

```
                                Delete Audio File
                                  <blank line>
                                  <blank line>
Delete "Weird Al Yankovich, Some Really Bizarre Song by Weird Al..."? (Y/N) ->
```

If the user enters “Y” or “y”, the specified **`AudioFile`** will be removed from the **`AudioList`** (see below, under “Non-functional Requirements”). If the user enters “N” or “n”, the record will be left untouched. If the user enters anything other than “Y/y” or “N/n”, the program should print an error message and re-prompt the user for the correct input as follows:

```
                                <blank line>
Must be 'Y' or 'N'.
Delete "Beatles, I Saw Her Standing There (Meet the Beatles)"? (Y/N) ->
```

Following a correct entry by the user, the program should then print 25 blank lines to the screen (to clear the original screen) and return to the opening screen.

If there are no records to delete (if the list is empty), the program should print 25 blank lines to the screen (to clear the original screen) and then print the following:

```
                            Delete Publication
                               <blank line>
                               <blank line>
Nothing to delete.
Press <ENTER/RETURN> to continue . . .
```

The cursor should pause after the “**`Press <ENTER/RETURN>`**” line. Once the user presses the **`<ENTER/RETURN>`** key, the program should print 25 blank lines to the screen (to clear the original screen) and return to the opening screen. The deleted audio file should no longer be present.

9. If the user chooses “S” or “s” (for “Search”), the program should print 25 blank lines to the screen (to clear the original screen) and then print the following:

```
                            Search Audio Files
                               <blank line>
                               <blank line>
This function is not currently available.
Press <ENTER/RETURN> to continue . . .
```

The cursor should pause after the “**`Press <ENTER/RETURN>`**” line. Once the user presses the **`<ENTER/RETURN>`** key, the program should print 25 blank lines to the screen (to clear the original screen) and return to the opening screen.

10. If the user chooses “P” or “p” (for “Play”), the program should print 25 blank lines to the screen (to clear the original screen) and then print the following:

```
                              Play Audio File
                                <blank line>
                                <blank line>
This function is not currently available.
Press <ENTER> to continue . . .
```

The cursor should pause after the "**Press <ENTER/RETURN>**" line. Once the user presses the **<ENTER/RETURN>** key, the program should print 25 blank lines to the screen (to clear the original screen) and return to the opening screen.

11. If the user chooses "N" or "n" (for "Next"), the program will bring 25 blank lines to the screen (to clear the original screen) and advance to the next page of audio files (items 17 - 32 for page 2, items 33 - 48 for page 3, etc.). If there are no further pages of audio files, then this command will have no effect.

12. If the user chooses "B" or "b" (for "Back"), the program will bring 25 blank lines to the screen (to clear the original screen) and move to the previous page of titles. If the program is displaying the first page, then this command will have no effect.

13. If the user chooses "Q" or "q" (for "Quit"), the program will bring 25 blank lines to the screen (to clear the original screen) and terminate.

3. Non-functional Requirements (Version 2)

3.1. Development Requirements

1. **Classes**: Your program will be built using a Model/View/Controller (MVC) architecture.

   `Model:`

   `AudioFile`: This class represents a single audio file. It will need instance variables to store the artist (**`String`**), title (**`String`**), album (**`String`**), and track (**`int`**) information. It will need methods to set and get these values. It will also need a **`toString()`** method that returns the **`String`** required for the display list. This class will require the following public methods:

   - **`public AudioFile( String artist, String title, String album, int track )`** — This constructor will set the values of the incoming parameters to the respective instance variables. You should use the various **`set...`** methods below to handle the assignments.
   - **`public boolean canAdd()`** — This method will return **`true`** if the **`AudioFile`** object is correctly configured and **`false`** otherwise. An **`AudioFile`** object will be correctly configured if both the **`artist`** and **`title`** attributes contain values (**`!null && length > 0`**) and the track is valid (between 1 and 99, or -1 if not entered).
   - **`public boolean equals( Object obj )`** — This method returns true if **`obj`** is "equal" to the current object. This is required by the **`contains()`** method of the **`ArrayList`**. To make this work properly, you will need to cast **`obj`** to the appropriate type. In this case, the first line of the method would include something like:

     ```
     AudioFile file = (AudioFile) obj;
     ```

     Two **`AudioFile`** objects are equal if all of their public attributes have the same values (**`artist == artist`**, **`title == title`**, etc.).

   - **`public String getAlbum()`** — This method will return the album **`String`**.
   - **`public String getArtist()`** — This method will return the artist **`String.`**
   - **`public String getTitle()`** — This method will return the title **`String.`**
   - **`public int getTrack()`** — This method will return the track number (as an **`int`**).

- **`public void setAlbum( String album )`** — This method will set the value of the **`album`** attribute. If the incoming parameter is **`null`**, this should be set to an empty **`String`**.
- **`public boolean setArtist( String artist )`** — This method will set the value of the **`artist`** attribute. The method will return **`true`** if the value of artist is valid (**`!null && trimmed length > 0`**). If the incoming parameter is either **`null`** or an empty **`String`**, then the method should assign an empty **`String`** to **`artist`** and return **`false`**.
- **`public boolean setTitle( String title )`** — This method will set the value of the **`title`** attribute. The method will return **`true`** if the value of **`title`** is valid (**`!null && trimmed length > 0`**). If the incoming parameter is either **`null`** or an empty **`String`**, then the method should assign an empty **`String`** (**`""`**) to **`title`** and return **`false`**.
- **`public boolean setTrack( int track )`** — This method sets the value of the **`track`** attribute. The method will return **`true`** if the value of **`track`** is valid (**`>= 1 && <= 99,`** OR **`-1`** if no track was entered) and **`false`** otherwise. Note: this method should accept and maintain whatever value is sent. If the track is invalid then the method should return **`false`** and the **`canAdd()`** method should return **`false`**.
- **`public String toString()`** — this method returns the **`String`** required for the display list (see above).

`AudioList`: This class represents the collection of audio files used by your program. It will need an instance variable to represent the collection — this will be built around an **`ArrayList`** of **`AudioFile`** objects. This program will function as a Set. It will require the standard methods for a collection (**`add()`**, **`get()`**, **`remove()`**, and **`size()`**) as well as methods for handling the standard set operations (**`intersection()`, `difference()`**, **`union()`**, and **`subset()`**). You may also want to pass through other useful methods from the **`ArrayList`** class. This class will require the following public methods:

- **`public boolean add( AudioFile pub )`** — This method adds or inserts a **`AudioFile`** alphabetically into the **`AudioList`**. This method will return **`true`** if the **`AudioFile`** can be added successfully. An **`AudioFile`** can be added successfully if the **`AudioFile`** does not already exist in the **`AudioList`** and if its **`canAdd()`** method returns **`true`**. If the **`AudioFile`** cannot be added (if its **`canAdd()`** method returns **`false`**), then the method should return **`false`**.

- ~~**`public int capacity()`** — This method returns the current size (capacity) of the array.~~ *No longer needed.*
- ~~**`public boolean deleteLast()`** — This method removes from the array the last **`AudioFile`** object that was added. You will need to decrement the value of **`last`** whenever an **`AudioFile`** is removed.~~ *No longer needed.*
- **`public boolean contains( AudioFile file )`** — This method returns true if file is contained by the underlying **`ArrayList`** of the **`AudioList`**.
- **`public boolean delete( int whichOne )`** — This method removes from the **`ArrayList`** the **`AudioFile`** found at the position indicated by **`whichOne`**. If the value of **`whichOne`** is invalid, then the method should return **`false`**. If the value is valid, then the method should return **`true`**.
- **`public AudioList difference( AudioList other )`** — This method will return a new **`AudioList`** that contains the difference between the current **`AudioList`** and the other **`AudioList`**.
- **`public AudioFile get( int whichOne )`** — This method will return the **`AudioFile`** object found at the position indicated by **`whichOne`**. If the value of **`whichOne`** is invalid, then the method should return **`null`**.
- **`public AudioList intersection( AudioList other )`** — This method will return a new **`AudioList`** that contains the intersection of the current **`AudioList`** and the other **`AudioList`**.
- **`public int size()`** — This method returns the size of the collection (the number of **`AudioFile`** objects currently stored).
- **`public boolean subset( AudioList other )`** — This method will return true if every item in the other **`AudioList`** is also in the current **`AudioList`**.
- **`public AudioList union( AudioList other )`** — This method will return a new **`AudioList`** that contains the union of the current **`AudioList`** and the other **`AudioList`**.

`View:`

`AudioView:` All interaction between the user and the program will be here. This is the only class that can contain **`System.out.print/println()`** statements. This is also the only class that can use a **`BufferedReader`** object. You should **NOT** user **`Scanner`**. Use only **`BufferedReader`**! This class should have the following methods:

- **`public void display( String str )`** — This method will display the value of **`str`** to the screen. You will need to use **`System.out.print()`** rather than **`System.out.println()`** here.
- **`public String getInput()`** — This method will get input from the user and return what the user enters as a **`String`**.
- **`public void pause( String message )`** — This method will print the incoming **`message`** and wait for the user to press the **`<ENTER/RETURN>`** key.
- **`public void clearScreen()`** — This method will print 25 blank lines to the screen.

- **`AudioFileIO:`** This class will handle all file reading and writing operations. This means that no references to **`BufferedReader`**, **`BufferedWriter`**, **`PrintWriter`**, or **`File`** should be in any other class. This class will require access to:

  - a file reader (**`BufferedReader`**, for example),
  - a file writer (**`BufferedWriter`** or **`PrintWriter`**, for example),
  - a **`File`** object (representing the file to read and write, in this case "AudioList.txt").

  Which classes you use for these I will leave up to you. In addition, you will need two finals to distinguish between reading and writing operations:

  - **`public static final int READER = 0`**
  - **`public static final int WRITER = 1`**

  This class will require the following public methods:

  - **`public AudioFileIO()`** — default constructor, creates a **`File`** object for "AudioList.txt."
  - **`public AudioFileIO( String fileName )`** — explicit value constructor, creates a **`File`** object for the **`String`** sent via the parameter.
  - **`public void close( int which )`** — this method will close the specified **`READER / WRITER`**.
  - **`public boolean open( int which )`** — This method will open the specified **`READER / WRITER`** for the file represented by the **`File`** object. This should return **`true`** if the method was able to open the file and **`false`** otherwise (file to read does not exist, cannot read file, file to write cannot be deleted, etc). Note: you will need to be careful with this, as the file writer will delete the file from which you need to read. So the sequence of operations is important here!!!

- **`public String readLine()`** — This method will read a single line from the file represented by the **`File`** object.
- **`public void write( String line )`** — This method will write a **`String`** to the file represented by the **`File`** object.

`Controller:`

`AudioControl:` This class will control the action of the program as a whole. It will use the View and Model objects to accomplish its tasks. This program will have a method that begins the program (called **`start()`**). You should think here about decomposing the problems with which you are dealing. For example, the opening screen displays information to the user, gets a response from the user, and then acts on that response. These could be 3 methods that are are called from the initiating method (**`start()`**). See the solution to lab 2 for an example of how such a control class can work.

In addition, your program will have a class called **`JMUTunes`** that contains the **`main`** method. This method will instantiate the **`AudioController`** and invoke its starting method (**`start()`**).

2. **Data Store**: The **`AudioFile`** objects will be stored in an **`ArrayList`** within an **`AudioList`** object. The **`AudioList`** will function as a Set. That is, it will not allow duplicate entries, and will implement the standard set functions (intersection, difference, and union). The **`ArrayList`** will store items in alphabetical order, with Artist, Album, track, and Title serving as the sort elements. That is, entries should be listed according to artist and album, with each title listed in track number order. For example, side "A" of the Beatles' album "Meet the Beatles" contains 6 songs in the following order:

```
Beatles, I Want To Hold Your Hand (Meet the Beatles)
Beatles, I Saw Her Standing There (Meet the Beatles)
Beatles, This Boy (Meet The Beatles)
Beatles, It Won't Be Long (Meet the Beatles)
Beatles, All I've Got to Do (Meet the Beatles)
Beatles, All My Loving (Meet the Beatles)
```

`AudioFile` objects should be entered into their correct position in the **`AudioList`** as they are added. Do not sort these after the fact (in other words, do not use **`Collections.sort()`** or any other sorting algorithm, including your own). If a **`AudioFile`** is edited so that its position in the list will change, then the edited **`AudioFile`** should be moved to its new (correct) position.

All access to the **`ArrayList`** inside of the **`AudioList`** object must be handled by the **`AudioList`** object itself. You should not allow any other class/object to have direct access to the underlying **`ArrayList`**.

3. **File Reading/Writing**: When the program begins, and before the first screen is displayed, the program will populate the **`AudioList`** with **`AudioFile`**s read from a data file. If the file does not exist, you will begin with an empty **`AudioList`**. In other words, the program should not fail if there is no file to read. When exiting the program, the program will populate the file with information contained within the **`AudioList`**. The file should be called **`AudioFile.txt`** and should contain the following information, with each **`AudioFile`** on a single line and with each data element separated by the pipe symbol (**`|`**):

   `Artist|Title|Album|Track|` (note the trailing "|")

   For example:

```
Beatles|I Want to Hold Your Hand|Meet the Beatles|1|
Beatles|I Saw Her Standing There|Meet the Beatles|2|
```

   Malformed Lines. If a line in the **`AudioFile.txt`** file is malformed, that is, it either has an insufficient number of elements (as defined by the pipe symbols) or is missing a required element (artist or title), then this line should not be added to the **`AudioList`**. Instead, you should write the malformed line to a file called **`malformed.err`**. For example, in the following example, lines 2 and 4 are malformed:

```
Beatles|I Want to Hold Your Hand|Meet the Beatles|1|
Beatles||Meet the Beatles|2|
Beatles|This Boy|Meet The Beatles|3|
Beatles|It Won't Be Long|4|
Beatles|All I've Got to Do||5|
Beatles|All My Loving|Meet the Beatles||
```

   Line 2 is missing a required element (title), while line 4 is missing one data element (missing publisher). In this example, lines 1, 3, 5 and 6 would be added to the **`AudioList`**. Lines 2 and 4 would be written to the **`malformed.err`** file. Note: empty elements are defined by adjacent pipe symbols. The following shows an **`AudioFile`** that is missing the album and track:

```
Beatles|Love Me Do|||
```

3.2.Testing Requirements

1. You will need to provide a **`JUnit4`** test class (not **`JUnit5`**!) for each of the model classes that tests each method within each model class. The test classes should named according to the class that it tests followed by the word "Test". So, for example, the **`JUnit`** test class for **`AudioFile`** should be named **`AudioFileTest`**. In addition, you will need to write tests for the **`AudioFileIO`** class. You will find it easiest to allow Eclipse to generate the shells of your **`JUnit`** test program (see the lab on this).

3.3.Stylistic Requirements

1. Your source code (excluding your **`JUnit`** tests) must adhere to the Checkstyle requirements as outlined in the *Style Guide*. **Assignments missing the acknowledgement statement specified in the *Style Guide* will receive a 50% deduction**. This statement must appear in the class header for **each class** in your assignment.

3.4. Submission Requirements

1. Your source code files should be in the default package that Eclipse creates. Do NOT include these in another package. Your source code files and **`JUnit`** test files should be combined into a single "zip" file. The name of the zip file should include the name of the assignment followed by the last name of the author: for example: **`PA2Norton.zip`**. The zip file should contain only the .java files and not the directories in which they are contained. You should submit this single "zip" file to the appropriate submission slot in **`AutoLab`**.

For PA2, you are allowed to use the professor's solution to PA1 as a base for your solution (recommended!). You are free also you begin with your own solution for PA1 or some combination your's and the professor's solutions. If you combine your solution with the professor's solution, be sure to clearly indicate which portions of code are "borrowed." If you begin with the professor's solution, be sure to clearly indicate what has been added and/or changed.

PA2 UML Class Diagram (JMUTunes)