

***JMUTunes*: A Digital Audio Player**
Software Requirements Specification
Version 1.032

1. Introduction

1.1. Objectives

1. This assignment has six (6) objectives:
 - To give you practice creating an application with multiple classes;
 - To give you practice using a Model/View/Controller architecture;
 - To give you practice using and manipulating an array;
 - To give you practice in controlling user input and handling errors and exceptions;
 - To give you practice in creating an attractive character-based user interface;
 - To take the first steps in creating a full-blown Digital Audio Manager and Player.

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 audio file information in an **AudioFile** object within a one-dimensional array of **AudioFile** objects.
4. The program will allow the user to delete the last audio title entered.

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:

```

                                JMTunes 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)delete, (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 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 to 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: Love me do
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: Love me do
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.

```
JMTunes Audio Player
CS159 (Fall 2018)
<blank line>
<blank line>
1. Beatles, Love me do (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)etele, (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, Love me do

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. If the user chooses “E” or “e” (for “Edit”), 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>
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 the program should print 25 blank lines to the screen (to clear the original screen) and return to the opening screen.

8. If the user chooses “D” or “d” (for “Delete”), the program should print 25 blank lines to the screen (to clear the original screen) and ask the user if they wish to delete the last record entered. 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, Love me do (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 last record entered 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, Love me do (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 (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 Audio File
<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 1)

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 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 array of **AudioFile** objects — as well as an **int** variable called **last** that indicates the position in the array for the last element added. As a collection, this class will require methods to add an **AudioFile** to the collection, delete an **AudioFile** from the collection, and get (return) an **AudioFile** from the collection. You should also provide a method that returns the size of the collection (the number of **AudioFiles** currently stored, not the size of the array). This class will require the following public methods:

- **public boolean add(AudioFile audio)** — This method adds an **AudioFile** to the end of the list. This method will return **true** if the **AudioFile** can be added successfully. An **AudioFile** can be added successfully if its **canAdd()** method returns **true**. If the **AudioFile** cannot be added (if its **canAdd()** method returns **false**), then the method should return **false**. Note: this method will need to check the size of the array to ensure that there is room to add the **AudioFile**. If the array is full, the method will need to double the size of the array as indicated below. You will need to increment the value of **last** whenever an **AudioFile** is added.
- **public int capacity()** — This method returns the current size (capacity) of the array.

- **public boolean deleteLast()** — This method removes from the array the last **AudioFile** object that was added. This method should return **true** if the deletion was successful, and **false** if nothing was deleted. You will need to decrement the value of **last** whenever a **AudioFile** is removed.
- **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 will return **null**.
- **public int size()** — This method returns the size of the collection (the number of **AudioFile** objects currently stored). This is not the same as the **capacity** of the array (unless, of course, the array is full).

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** use **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.

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 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 a one-dimensional array of **AudioFile** objects stored within an **AudioList** object. The array will initially contain 16 elements. **AudioFile** objects will be stored in the order they are added. If more elements are needed (if the array is full), then the **AudioList** should double the size of the array. You will need to create a new array with twice the number of elements contained by the current array, copy each cell from the current array to the new array, and then rename the new array to that of the current array.

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 be 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**. 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: **PA1Norton.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**.

PA1 UML Class Diagram (JMUtunes)

