Module: Core Java

Session 21: Generics and Legacy classes Practice

* This is a practice session; you will work on Collections assignments
* You can discuss your doubts with the trainer

**Assignments:**

**Assignment 1 – Collections Assignment**

**Objectives:**

* Learn to use the Java Collections Framework
* Implement a system according to specifications that are provided by someone else.
* Become familiar with standard java doc documentation.
* Use UML diagrams to help you understand how the different components of a system should interact.

**Introduction**

In this assignment you will implement a music library for organizing and playing locally stored music files such as MP3s, according to the provided specification. You are only asked to write a small number of classes, but you will most likely find that the task of maintaining consistency as changes are made to the library is a difficult one.

**Required Reading**

MusicLibrary API, specifically the **library** package (You can ask your instructor to provide this API) This API is available as MusicLibrary-javadoc.zip file.

**Setup**

1. Create the directory MusicLibrary in your local system (For example either on desktop or some drive like C: or D: drive)
2. Your instructor will provide you provided-classes.zip file that you have to unzip it to the directory created above.
3. Your instructor will provide you libraries.zip file that you have to unzip it to the directory created above.
4. Create a new project in Eclipse called MusicLibrary. Choose Create project from existing source and fill in the directory field with MusicLibrary (Directory that you created above). Click Next and then click on the Libraries tab. If you don't see the three JAR files that you extracted from the ZIP file (jawin.jar, JID3.jar, and MPStubs.jar) listed there, click Add JARs and add them to the build path. Click Finish.
5. Create a package named library.

**The Music Player**

The portion of the program that actually plays music has been provided for you. It will operate on Linux machines provided that XMMS is installed and that you have our modified version of xmmsctrl, and on Windows machines provided that Windows Media Player 6.4 or higher is installed.

**Problem specification**

You will produce a music library with functionality similar to programs such as Microsoft Windows Media Player and Apple iTunes. Your program will allow the user to:

* View a list of all the songs in the library and see information about each song, including the album on which it appears and the artist(s) who performed it.
* View a list of all the artists in the library.
* View a list of all the albums in the library.
* Edit any information in the library. This includes changing the title of songs and albums and changing the names of artists.
* View a list of all the songs by a particular artist.
* View a list of all the songs on a particular album.
* Add songs to and remove them from the library.
* Add artists to and remove them from songs.
* Add songs to and remove them from albums.
* Play songs using the provided player

**Organization**

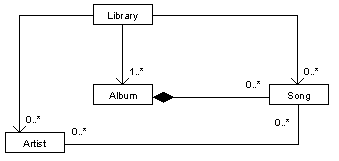
Your project will contain 4 packages:

* player: this package (and it's sub-packages) are provided for you and you will use them to play songs
* util: this package contains a class you can use to load descriptive information stored in MP3 files
* ui: this package contains a test driver you can use to test your library
* library: this is the package you will implement

For this assignment, you will not have to use the player or util packages directly as they will be used by the test driver we provide.

**Classes You Will Implement**

You will implement the 4 classes in the library package as they are specified in the javadoc. The following class diagram gives an overview of the package:



* Each instance of the Song class will correspond to a unique file on disk and will store the following pieces of information:
  + The path to the file on disk
  + The title of the song
  + The duration of the song
  + The track number
* Each instance of the Artist class will store the name of a person (eg. "Lucinda Williams") or group (eg. "Ben Harper & the Innocent Criminals", "Radiohead") that performs Songs. There will be no distinction between groups and individual people.
* Each instance of the Album class will represent an album - a collection of Songs ordered by track number (where each song on an album has a different track number) - and will store the album title. In addition, there will be at least one album in the system at all times (a "default" album) and every song will be associated with exactly one album as shown above. Any song for which the user has not entered an album will belong to the default album.
* There will be a single instance of the Library class in the system which will maintain collections of the other three classes and provide methods to create and destroy them that ensure consistency between the library and the songs, artists, and albums.
* Each class will also implement the associations depicted above.

You must implement all the methods described in the javadoc and will probably need to add some private methods of your own. Note that you are **not** allowed to modify the classes in the ui package. The code we provided **must** work with your implementation of the library package.

**Library as a Singleton**

As stated in the Javadoc, you must implement the Library class as a singleton. A singleton class has the following features:

* Only a single instance of the class may be created and this instance can be accessed from anywhere. This is appropriate for the Library class because it would make no sense (in this context) to have multiple libraries, and because many classes will want to manipulate the library.
* To ensure that there is exactly one instance of the class, the constructor is **private** and is called once from within the class to initialize the single instance, which will be kept as a static field of the class.
* To allow access to the single instance, the class defines a static method called getInstance.

Here is an example:

public class Singleton

{

// this is the only time the constructor is ever called

private static Singleton instance = new Singleton();

// This is the only constructor and it is private

private Singleton()

{

// standard constructor code goes here

}

// Return the single instance of this class. Notice that

// every call to this method returns the same instance; no new

// instances are created.

public static Singleton getInstance()

{

return instance;

}

}

To access the singleton class, other classes would call Singleton.getInstance().

**Collections**

The "many" ends of the above associations must be implemented using members of the Java Collections Framework. You will be graded (in part) on which collections you choose to use. For each case, you should ask yourself the following questions:

* Which is more appropriate, a Collection or a Map?
* If I use a Collection, should I use one that permits duplicates?
* Should I use a collection that does or does not keep its elements in a specific order?
* Which implementation will be most efficient, based on the expected usage of the collection?

Here you should also consider whether the features of a particular collection are worth the cost. For example, you may want to use a Set to maintain a collection that is not allowed to contain duplicates, however, the restriction on modifying members of a Set might be inconvenient and you may not expect this collection to contain more than a few elements. In this case it may be much more efficient to use a List and manually ensure that you never insert duplicate elements.

**A Note On Terminology**

In this document, the word "collection" written in plain text refers to a member of the Java Collections Framework (ie. a class that implements either java.util.Collection or java.util.Map). The word "Collection," capitalized and written like this refers to java.util.Collection or one of its implementations. In other words, a Collection is a collection that is not a Map. This is the same convention used in the Java API and the Java Tutorial.

**Other Specific Requirements**

In addition to everything stated above (and below) this section you should keep the following in mind:

* When editing information in the library or adding songs, users of your program will specify Artists and Albums by entering their names/titles.
* We assume the path to songs will not change, that is, if the user renamed or moved a file they would need to remove the corresponding Song from the Library and create a new Song.
* The library will not be allowed to contain two Albums with the same title, two Artists with the same name, or two Songs representing the same file.
* Any method that returns a collection of Albums should return them in alphabetical order by title.
* Any method that returns a collection of Artists should return them in alphabetical order by name, with one exception: invoking getArtists() on a Song should return them in the order they were added to the Song.
* The order in which Songs are returned is irrelevant (since it is not clear by which attribute the user is most likely to want their songs ordered - we can leave this decision up to the user interface) with the exception that the getSongs() method of the Album class must return its Songs ordered by track number.

**Where to start**

1. You should first read and understand the javadoc for the library package in its entirety because it specifies in detail what you need to do for this assignment. Think about the interaction between the methods of different classes, remembering that a method that creates or destroys a bi-directional association between classes must ensure that the two ends of the association are consistent with each other. Note that you can add methods with default (package level) visibility which can only be called by classes within the library package - this restriction on how a method will be called relaxes the requirement that it maintain consistency provided the calling method takes on that obligation.
2. Take some time to review the Java Collections Framework and think about which class would be most appropriate for each of the associations.
3. Implement the system. A good place to start is the Album class as it is probably the simplest and least coupled class in the package.

**Testing Your Library**

You should use the TestDriver class in the ui package to test your library. It takes a single, optional command line argument: a directory containing MP3 files. Multiple directories can be specified by separating them with a colon (:) on UNIX systems or a semicolon (;) on Windows systems. You can use some MP3 files for testing purposes.

TestDriver will then create a Library containing Songs, Artists and Albums based on the information it reads from the ID3 tags of the files in the specified directory. You can use TestDriver to view the contents of the library, to play, add, and remove songs and to edit the information contained in the library. The interface provided is menu-based. In assignment 3, you will replace it with a much nicer-looking graphical user interface.

When you exit the test driver, it will save the library (including any changes you make) to a file called MusicLibrary.xml in your MusicLibrary directory. When the test driver is started again, it will first load the contents of the MusicLibrary.xml file and then add any other songs in the directories specified on the command line. To revert to the information stored in the MP3 files, simply delete the MusicLibrary.xml file and restart the test driver.

You should not assume that your classes are bug-free just because the test driver works correctly, but it will probably catch any major bugs. Still, you should test each method and each class as you write it; you do not want to find yourself in the position of having spent many hours writing a complete system which completely fails to work.

**Deliverables**

The library package you created, including the following:

* + Library.java
  + Artist.java
  + Song.java
  + Album.java

You should not have written or modified any code outside of these 4 files.