Java Programming (JOOSE2)

Laboratory Sheet 9

The deadline for submission is Thursday 1 December at 17:00. The submission site will be opened on Monday 28 November at 13:00.

Aims and objectives

- Gain experience with writing JUnit test cases of your own and using test-driven design
- Gain experience writing components of a simple GUI application in Swing
- Design and write your own extended code without a specification

Set up

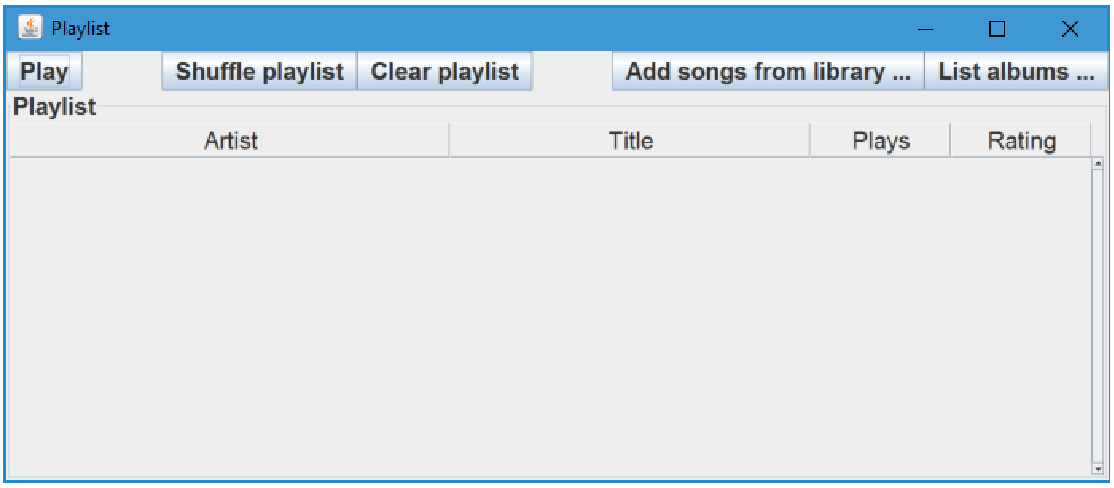
1. Download Laboratory9.zip from Moodle and launch Eclipse.
2. In Eclipse, select File → Import … to launch the import wizard – this will be used to import the starter projects into your workspace. Then select General → Existing Projects into Workspace and click Next.
3. Choose Select archive file and then browse to the location where you downloaded Laboratory9.zip. Select Laboratory9.zip and click Open.
4. You should now see a new project in the Projects window: Submission9_1. Ensure that the checkbox beside this project is selected, and then press Finish.
5. In the *Package Explorer view* (on the left of the Eclipse window), you should now see an icon representing the new project (in addition to any projects that were there before).

Please refer to the lab sheet from Laboratory 1 for additional details on how to open and run the code in the new project. Note that, as in previous labs, you will be creating some new classes as part of the assignment; refer to the lab sheet for Laboratory 4 for details on creating classes in Eclipse if necessary.

Instructions

Remember the playlist and song classes we used in lab 7? We are going to revisit those classes to develop an interactive, GUI-based system. Since this is a more complex exercise, you will have two lab sessions (weeks 10 and 11) to work on this code. You do not have to submit your code until Thursday of week 11, after your last lab session of the semester.

The code you have been given is the start of a Swing music player application. If you run the main method of the **musicplayer.Main** class, you will see a window pop up similar to the following (this is an instance of the **musicplayer.MainWindow** class):



Note that when you click on the buttons, nothing happens at the moment. Your job is to write the back-end code to turn this window into an interactive GUI application.

Internal structure of MainWindow class

There is a lot of code in the **MainWindow** class, much of it involved in actually laying out the user interface. Here is a summary of the overall structure:

- Each graphical component on the screen is an instance field – for example, the **playButton** field represents the "Play" button
- The **displayed** playlist of songs is represented by the **JTable** called **songTable**
- The underlying list of songs that is displayed by **songTable** is stored in the field **songModel**, which is of type **SongTableModel**. This model can be accessed from outside the **MainWindow** class through the public **getSongModel()** method.
  - Note: **SongTableModel** is a subclass of **javax.swing.table.AbstractTableModel**, which is in turn an implementation of **javax.swing.table.TableModel** – more documentation of these base classes can be found at http://docs.oracle.com/javase/tutorial/uiswing/components/table.html.
  - The **SongTableModel** stores a list of **Song** objects in an internal **List<Song>**, and the methods at the top of the class provide access to this list. The overridden methods at the bottom of the class control the rendering of the table on screen – you do not need to read those methods unless you are curious about the details.
- The **MainWindow** constructor creates all of the components and lays them out on the screen, and also sets up the behaviour of the top-level window – e.g., it ensures that the program exists when the window is closed.
- **MainWindow** also implements **ActionListener**, which means that it provides an **actionPerformed()** method. This method is registered with all of the on-screen buttons, which ensures that whenever any of the buttons is pressed, the **actionPerformed()** implementation is called.

Before working on the GUI side of the application, your first task is to complete the classes in the **musiclibrary** package – this set of classes will be used to store and manage the songs that are used by the music player.

Implementing the Song interface

The interface **musiclibrary.Song** defines the methods and behaviours for a simplified representation of a song. Your first task is to provide a concrete implementation of this interface – a suggested name for this class would be **musiclibrary.SongImpl**. The Javadoc comments in the **Song** interface should provide most of the information that you will need to carry out this implementation; here are some additional notes.

Every song must have an **artist**, a **title**, an **album title**, and a **year**. Depending on the song, it may or may not also have a separately defined **album artist** – this property is only necessary in the case of (a) compilation albums where the album artist is probably "Various", or (b) guest appearances by one artist on another artist's album.

The **Album** class represents the properties of an album – i.e., the title, year, and artist. The **Song.getAlbum()** method should create and return an Album instance based on the properties of the current song: in particular, it should use the album artist if it is defined, or else use the song artist if no album artist is given.

A Song must also have a **play count** (initially zero when the Song is constructed) and a **rating** (between 0 and 5 – also initialised to zero). The getPlayCount(), increasePlayCount(), getRating(), and setRating() methods should be used to manage these two fields – see the Javadoc for the specification.

Writing test cases for **SongImpl**

As part of the process of implementing **SongImpl**, you should also write a set of JUnit test cases to verify the behaviour of your class. You should write a test suite in the class **test.TestSongImpl** – try writing the test cases and the body of the class at the same time (i.e., Test-Driven Design). You do not need to test the behaviour of simple methods such as getter and setters; concentrate on the more complex and potentially error-prone functionality.

Completing MusicLibrary

The **MusicLibrary** class is mostly complete, but it is missing implementations of the two methods **getAlbums()** and **getAlbumSongs()**. You should implement these two methods based on the documentation.

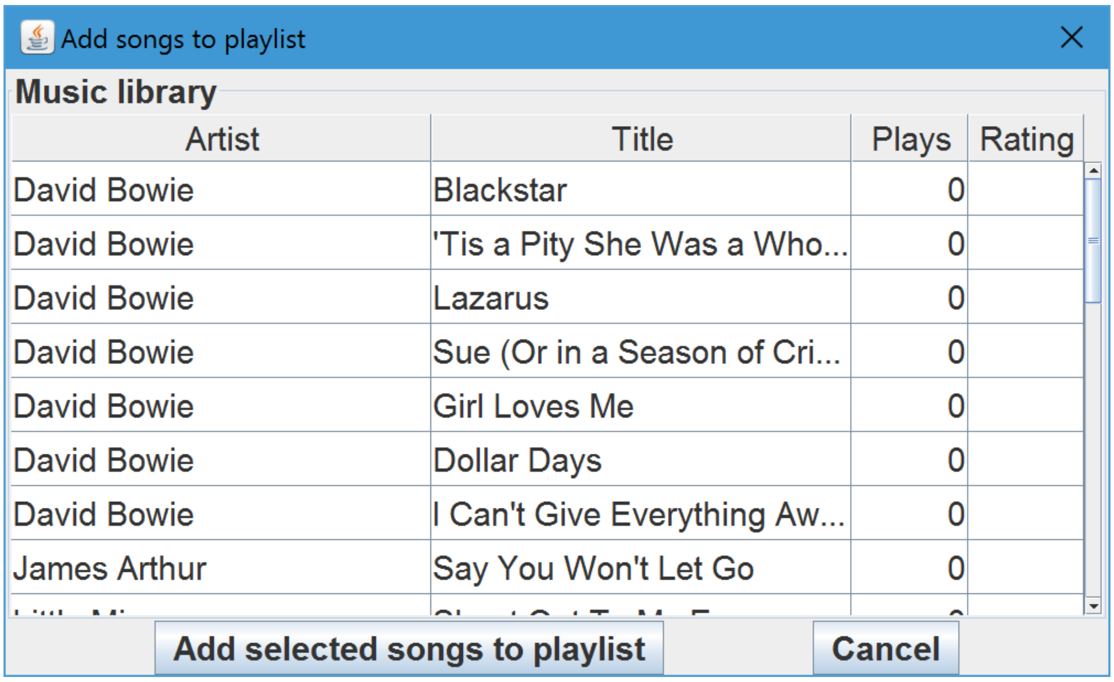
Once you have completed the music library, you must write action handlers for all of the GUI components. The behaviour for the **Clear** and **Shuffle** buttons is already implemented (in **MainWindow.actionPerformed()**) – the following is the required behaviour for the other buttons. I will give more suggestions on implementations below.

- **Add songs from library**: when this is clicked, you should create a new instance of the provided **ChooseSongDialog** class and call its **setVisible()** method; the required behaviour of this dialog will be specified below.
- **List albums**: when this is clicked, you should create a new instance of the provided **ChooseAlbumDialog** class and call its **setVisible()** method; the required behaviour of this dialog will be specified below.
- **Play**: when this button is clicked, the method should:
    - Determine which row in the playlist is selected by calling **songTable.getSelectedRow()** – this will return -1 if no row is selected and the selected index otherwise
    - For each song in the list, from the selected row to the end (of from index 0 if nothing is selected), it should increase the play count using the **increasePlayCount()** method – use **songModel.getSong()** to access the individual songs from the model.
    - After all play counts have been updated, call **songModel.fireTableDataChanged()** to ensure that the display is updated.

All of the above behaviour should be implemented inside the **MainWindow.actionPerformed()** method. Inside that method, you can use **event.getSource()** to find out which of the buttons was actually pressed so that you can implement the required behaviour.

Choosing songs from the list

As mentioned above, the process of adding songs to the list should be handled in the **ChooseSongDialog** – when the "Add songs" button is pressed, the **MainWindow** should just create a new **ChooseSongDialog** with appropriate parameters and make it visible through **setVisible(true)**. Here is what it looks like on screen:



Artist	Title	Plays	Rating
David Bowie	Blackstar	0	
David Bowie	'Tis a Pity She Was a Who...	0	
David Bowie	Lazarus	0	
David Bowie	Sue (Or in a Season of Cri...	0	
David Bowie	Girl Loves Me	0	
David Bowie	Dollar Days	0	
David Bowie	I Can't Give Everything Aw...	0	
James Arthur	Say You Won't Let Go	0	

The following is the structure of **ChooseSongDialog**:

- The parent **MainWindow** is stored in the field **mainWindow**, which will allow **ChooseSongDialog** to access and modify the playlist in the top-level frame.
- **ChooseSongDialog** also contains a tabular list of songs similar to the one in the top-level playlist, in the fields **songTable** and **songModel**.
- **ChooseSongDialog** implements **ActionListener**, which means that it also has an **actionPerformed** method. This method is called whenever the "Add selected songs" and "Cancel" buttons are pressed.

A skeleton **actionPerformed** method has been created for you, and I have already implemented the behaviour for the "Cancel" button (the dialog box is closed with by calling **dispose()**).

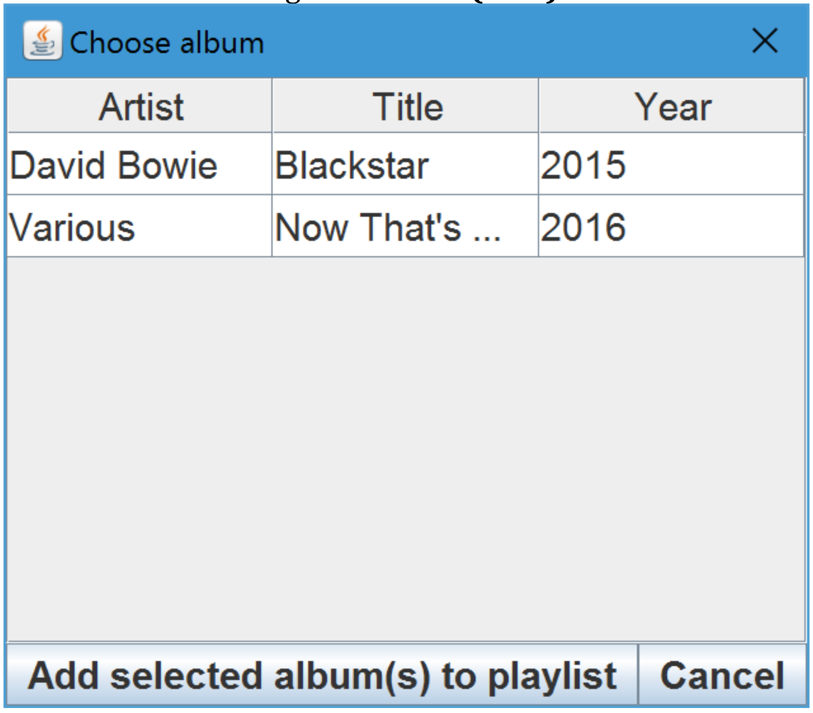
You should implement the following behaviour to respond to a press on the "Add songs" button

- Use **songTable.getSelectedRows()** to obtain the set of selected rows in the song table (as an int[]).
- For each selected row, use **songModel.getSong** to access the Song object associated with that row.
- Put all of the selected songs together into a List<Song> and call **songModel.addSongs()** on the resulting list – this will append the selected songs to the playlist.
- After the songs have been added, call **dispose()** to close the dialogue box.

Note that you should not worry about duplication – it is perfectly acceptable for the playlist to consist of several copies of the same song.

Adding all songs from an album

The process for adding songs from an album is similar to the process of adding individual songs. When the "List albums" button is pressed, the **MainWindow** should create a new **ChooseAlbumDialog** with the appropriate parameters and make it visible through **setVisible(true)**. Here is what it looks like on the screen:



Artist	Title	Year
David Bowie	Blackstar	2015
Various	Now That's ...	2016

The structure of **ChooseAlbumDialog** is very similar to **ChooseSongDialog**, except that the table shows a list of Albums rather than a list of Songs.

Again, you are to complete the **actionPerformed**() method – the behaviour for the "Cancel" button is already given for you, so you need to fill in appropriate behaviour if the event source is the "Add selected album(s)" button, as follows.

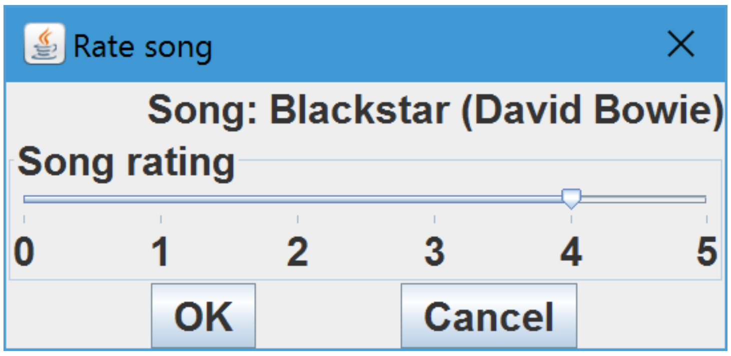
You should implement the following behaviour to respond to a press on the "Add selected albums" button

- Use **albumTable.getSelectedRows()** to obtain the set of selected rows in the song table (as an int[]).
- For each selected row, use **albumModel.getAlbum** to access the Album object associated with that row, and then use **musicLibrary.getAlbumSongs()** to access the songs from that album.
- Put all of the selected songs together into a List<Song> and call **songModel.addSongs()** on the resulting list – this will append the selected songs to the playlist.
- After the songs have been added, call **dispose()** to close the dialogue box.

Note that, as in the case of adding individual songs, you should not worry about duplication – it is perfectly acceptable for the playlist to consist of several copies of the same song.

Rating a song

I have also added code to the **MainWindow** class to allow the user to update the rating of a song by double clicking on the song in the main playlist window. In particular, double clicking on a song will bring up a **SongRatingDialog**, which looks like this:



You must also implement the **actionPerformed()** method of this dialogue box – if the cancel button is pressed, the dialogue should be close with **dispose()** (as with the other dialogue boxes); if the **OK** button is pressed, you should do the following:

- Get the current value of the rating slider with **slider.getValue()**
- Call **song.setRating()** to set the rating
- Call **mainWindow.getSongModel().fireTableDataChanged()** to ensure that the playlist is updated with the new rating data.
- Close the dialogue box with **dispose()**

Extended Behaviour

If you implement all of the above behaviour correctly, the maximum you can receive is a B. **To get an A on this assignment, you also need to add some extended functionality to the system.** What you add is up to you – it could involve improving the interactive behaviour of the GUI interface, or it could involve improving and/or restructuring the underlying code. Here are some possible examples of things you might try – but if you want to try something else, you can do that too.

- Enabling/disabling the Play, Clear, and Shuffle buttons depending on whether the playlist is empty (look up the documentation for **TableModelListener**).
- Making sure that the playlist never contains any duplicate songs, e.g., by removing duplicate copies whenever songs are added to the list
- Popping up a confirmation dialog before clearing the playlist (look up the documentation for **JOptionPane**)
- Adding the ability to load/save playlists through the interface – look up the documentation of **JFileChooser** for selecting the file name. You can use and/or modify the load/save code in **MusicLibrary** as a starting point.

In addition to implementing the extended behaviour, you should write a short document (as a text file, Word document, or similar) describing what you implemented and how you did it – this should allow the tutors to mark your assignment properly.

Submission

You should submit your work before the deadline no matter whether the programs are fully working or not. **Be sure to remove all the // TODO notes inside the source files.**

When you are ready to submit, go to the JOOSE2 moodle site. Click on Laboratory 9 Submission. Click 'Add Submission'. Open Windows Explorer and browse to the folder that contains your Java source code and drag **any of the Java source files that you modified or created** into the drag-n-drop area on the moodle submission page. **Your markers only want to read your java files, not your class files.** Please also upload **the document where you describe the extended functionality** that you implemented. Then click the blue save changes button. Check the files are uploaded to the system. Then click submit assignment and fill in the non-plagiarism declaration. Your tutor will inspect your files and return feedback to you via moodle.

Outline Mark Scheme

Below is an illustrative mark scheme. This is a large and challenging project so tutors will use their discretion. Note that this is a double-submission worth 5% of the final course grade.

A: Correct code with fully correct behaviour. Extension is well-motivated, correctly implemented, and clearly described. Code has sensible structure and is well commented. JUnit test cases are comprehensive.

B: Correct implementation of all **actionPerformed** methods, but no extension developed (or extension not working correctly). Mostly correct code. Code has reasonable structure and some comments. Test cases provide reasonable coverage.

C: Code compiles and behaves fairly reasonably. At least one **actionPerformed** method implemented correctly, and at least some test cases provided. Code has fair structure.

D: Code may not compile. Reasonable attempt at one **actionPerformed** method and at least one test case.

E: Code does not compile. Some evidence of attempting to write **actionPerformed** methods and/or test cases.

Within these general guidelines, the tutors are also checking to see whether:

- you use sensible variable names and method names
- you write appropriate comments (Javadoc or similar) for classes and 'interesting' methods – **be sure to remove all // TODO comments and the like**
- you write clean and efficient control flow structures (e.g. for each loops) in your code.
- you include an @author tag in each source code file
- you are using packages correctly