

Lab 01: PlayListApp

In this lab, you will:

- *Work with a basic music playlist.*
- *Complete the partial implementation of the playlist, which aggregates a set of Track objects.*
- *Create a controller for the playlist, which controls various data-flow to a playlist instance, separating-out this responsibility from the playlist itself.*
-

... which will result in the following outcomes:

- *A reminder of the function of:*
 - *getter and setter methods and class constructors.*
 - *Use of Predicate.*
 - *Overridden methods.*
 - *Introduction to the basic use of a Controller class*
-

Table of Contents

1.1	Preliminary	3
1.2	Exercise 1	3
1.3	Exercise 2	5
1.4	Exercise 3	6
Figure 1: PlayListAppStarter project		3
Figure 2: Target output from the TrackTest class		4
Figure 3: Configuring Netbeans		5

1.1 Preliminary

- Download the lab1.zip file < **Lab1_PlaylistAppStarter.zip** > from GCULearn and unzip.
- Launch NetBeans 8 and open the **Lab1_PlaylistAppStarter** project.

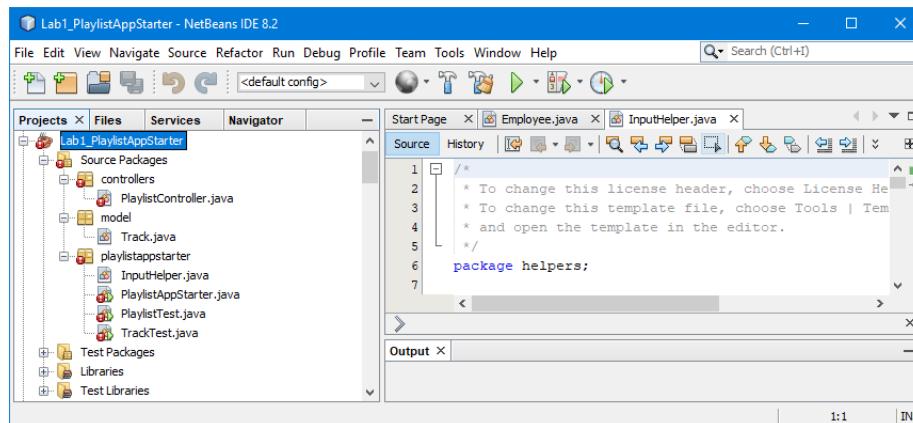


Figure 1: Lab1_PlayListAppStarter

Note the structure of the application with packages folders for model and controllers. Currently there are syntax errors in some of the classes which are addressed in the following exercises.

1.2 Exercise 1

Open the **Track** class, in the model folder, and note the attributes: **trackId**, **trackTitle**, **trackArtist** and **trackLength**.

- Add in the missing getters and setters for attributes.

Note the class variable – **lastAllocatedTrackId** – which is used to generate a value for the **trackId** attribute when a new **Track** object is created. You can see how this is used in the two constructor methods: in both cases, a new **Track** object is being created so a new value for **trackId** is required.

```
public Track(String trackTitle, String trackArtist, int trackLength)
{
    this.trackId = ++lastAllocatedTrackId;
    this.trackTitle = trackTitle;
    this.trackArtist = trackArtist;
    this.trackLength = trackLength;
}
```

However, as we shall see later, we may be creating a new **Track** object from data which is being retrieved from a data store. In this case, track information, including track id, has been set during a previous programme run so we shouldn't be creating a new value for the **trackId**.

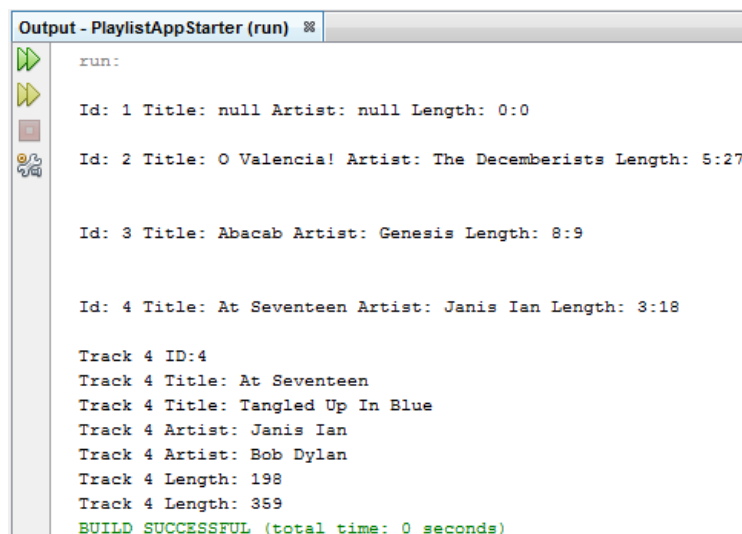
- Create a new constructor method which accepts parameter values for all attributes.

```
public Track(int trackId, String trackTitle,
```

```
String trackArtist, int trackLength) { ... }
```

Note the private methods: `getTrackLengthMinutes()` and `getTrackLengthSeconds()` which split the `trackLength` value into separate values for minutes and seconds. These methods are called from the `toString()` method which provides a `String` version of the current values of attributes for the object. This method is preceded by the `@Override` annotation to indicate that a `toString()` method is inherited from the `Object` class and we are overriding this to provide a customised version.

- c. Run the `TrackTest` class and ensure that you can create, update and display tracks.



```
run:
Id: 1 Title: null Artist: null Length: 0:0
Id: 2 Title: O Valencia! Artist: The Decemberists Length: 5:27
Id: 3 Title: Abacab Artist: Genesis Length: 8:9
Id: 4 Title: At Seventeen Artist: Janis Ian Length: 3:18
Track 4 ID:4
Track 4 Title: At Seventeen
Track 4 Title: Tangled Up In Blue
Track 4 Artist: Janis Ian
Track 4 Artist: Bob Dylan
Track 4 Length: 198
Track 4 Length: 359
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 2: Target output from the `TrackTest` class

Note you may have to change the configuration of the project to specify which class contains the `main()` method you wish to run.

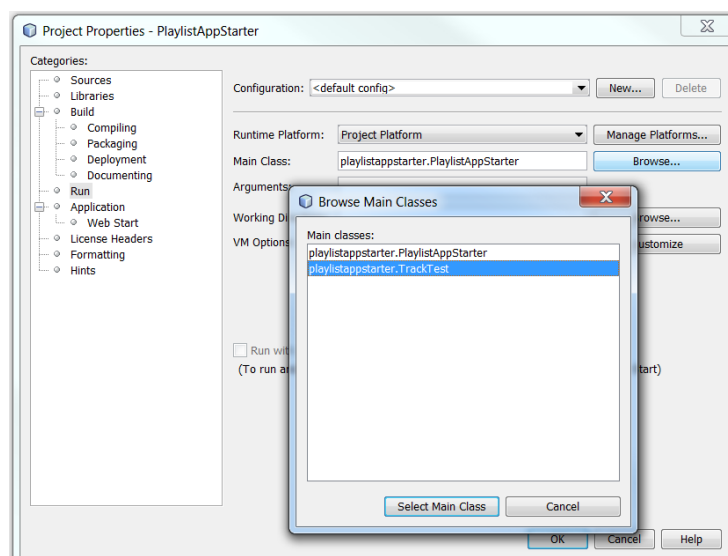


Figure 3: Configuring Netbeans

1.3 Exercise 2

We are working towards a basic Playlist app so next we have to create a **Playlist** class, in the model folder, which aggregates a collection of **Tracks**.

- a. Add a new class to the model folder called **Playlist** and declare attributes as follows:

```
playlistName - String
playlistAuthor - String
playlistTracks - ArrayList of Track
```

- b. Three constructors are required: a parameterless constructor detailed below; a constructor which accepts name and author; and, a third constructor which accepts values for all three attributes.

```
public Playlist() {
    this.playlistName = "New Playlist";
    this.playlistAuthor = "Unknown";
    this.playlistTracks = new ArrayList<Track>();
}
```

Create the other two constructors.

- c. Add getters and setters for each of the attributes.
- d. Add a **toString()** method which uses the private method **getNumberedTracks()** detailed below:

```
private String getNumberedTracks() {
    String numberedTracks = "\n";
    int number = 1;
    for (Track track : this.playlistTracks) {
        numberedTracks += Integer.toString(number++) + ":
" +
                                track.toString() + "\n";
    }
    return numberedTracks;
}
```

- e. We also require methods to add and remove tracks from the playlist. The **addTrackToPlaylist()** method is straightforward and simply involves accepting a track as a parameter and adding it to the **playlistTracks** attribute:

```
public void addTrackToPlaylist(Track track) {
    this.playlistTracks.add(track);
}
```

The removal process is more complex. As a taste of things to come I have implemented this using a **Predicate** object which will search the list for a track that matches a supplied **trackId** and then removes it from the list. Later, we will see how this works:

```
public void removeTrackFromPlaylist(int trackId) {
    Predicate<Track> trackPredicate =
        t-> t.getTrackId() == trackId;
    this.playlistTracks.removeIf(trackPredicate);
}
```

Add these methods.

- f. Run the **PlaylistTest** class to create a playlist and add and remove tracks.

1.4 Exercise 3

We are now going to introduce the concept of a controller class which will interpret user requests and, through interacting with model classes, generate suitable responses. Now the app simply has to create a controller object and then run it; from then on the controller object carries out all the work.

- a. Open the **PlaylistAppStarter** class and note its use of the **PlaylistController** object:

```
public static void main(String[] args) {
    PlaylistAppStarter playlistApp = new
    PlaylistAppStarter();
    playlistAppStarter.run();
}

public static void run() {
    System.out.println("PlaylistApp\n=====\\n\\n");

    PlaylistController playlistController =
        new PlaylistController();

    playlistController.run();

    System.out.println("Thank you for using
    PlaylistApp.\\n");
}
```

- b. Open the **PlaylistController** class and note it has a **Playlist** attribute.

```
private final Playlist playlist;
```

Note the **PlaylistController** constructor uses an **InputHelper** object to ask for a playlist name and author; using these values in creating the **Playlist** object:

```
public PlaylistController() {
    InputHelper inputHelper = new InputHelper();
    String playlistName =
        inputHelper.readString("Enter Playlist
Name");
    String playlistAuthor =
        inputHelper.readString("Enter Playlist
Author");

    playlist = new Playlist(playlistName,
playlistAuthor);
}
```

Examine the **InputHelper** class which contains methods for reading strings and numbers from the standard input stream.

- c. Examine the **run()** method; note, it uses a private method **displayPlaylistMenu()** to display options for adding and removing tracks.

```
public void run() {
    boolean finished = false;
    do {
        System.out.println(playlist);
        char choice = displayPlaylistMenu();
        switch (choice) {
            case 'A':
                addTrack();
                break;
            case 'B':
                deleteTrack();
                break;
            case 'F':
                finished = true;
        }
    } while (!finished);
}

private char displayPlaylistMenu() {
    InputHelper inputHelper = new InputHelper();
    System.out.print("\nA. Add a new track");
    System.out.print("\tB. Delete a track");
    System.out.print("\tF. Finish\n");
    return inputHelper.readCharacter("Enter choice",
"ABF");
}
```

- d. Examine the **addTrack()** method which is used to get new track details from the user – using an **InputHelper** object – and then executes the **addTrackToPlaylist()** method on the **Playlist** object.

```
private void addTrack() {
    InputHelper inputHelper = new InputHelper();
    String trackTitle =
        inputHelper.readString("Enter Track Title");
    String trackArtist =
        inputHelper.readString("Enter Track
Artist");
    int trackMinutes =
        inputHelper.readInt("Enter Track Minutes", 60,
0);
    int trackSeconds =
        inputHelper.readInt("Enter Track Seconds", 60,
0);
    Track newTrack = new Track(trackTitle, trackArtist,
        trackMinutes * 60 +
trackSeconds);
    playlist.addTrackToPlaylist(newTrack);
}
```

- e. Implement the **deleteTrack()** method:
- Create an **InputHelper** object
 - Read an integer between 1 and 100 representing the track id to delete
 - Call the appropriate **Playlist** method using the track id
- f. Run the **PlaylistAppStarter** class and ensure that you can create a playlist, add and remove tracks.