ExerciseSheet04

The Artist manager is satisfied with the ongoing implementation.

But again, she has some new requests :)

Now she asks you to create some wellformed output for the Tracks. Instead of using the oldfashioned way of always implementing the exact same thing while hoping you didn't forget anything, you will use the concept of Abstract Classes to create Formatters for your Tracks.

While working with Arrays of your Tracks, you additionally have the strong feeling it would be great to sort those arrays. Whilst you know already how to sort them basically, additionally it would be great to sort them by other criteria. As you know, to sort elements in a container, you first need a possibility to compare those elements. Therefore you make again good usage of abstract classes and inheritance to implement various comparators (Classes that allow to compare specific Datatypes).

After implementing those Formatters and Comparators and using them in a Main Method for displaying and sorting arrays of Tracks or Events, your manager has another idea

She wants to have an overview over the `Album` and `MusicVideo` Releases, she takes care of.

After some requirements engineering, you recognize, that both have several fields and methods in common, therefore you decide to implement an abstract class `Release` and inherit those information to `Album` and `MusicVideo`.

The most important fact for `Album`is, that it should refer to some kind of single linked list without using any collections you know so far.

As last tsak before full testing, you notice that it's not that cool to always hardcode Tracks in your main method. Therefore you decide to give the manager the chance to enter some basic information for a `Track` from the console. As this may be a good feature for other classes as well, you decide to implement this functionality by an interface called `ConsoleScanable`.

As all these tasks are not that easy to fulfill, make sure you perform good testing on your own by implementing a main method which makes use of all this stuff.

> Be aware that it may happen that you may have questions during implementation. Don't hesitate to ask the "managers" (your lecturers and colleagues) by posting your question to the forum!

Tasks

- Implement the Track Formatters (have a look at the given classes `MyTrackFormatter` `MyShortTrackFormatter` for a Hint how to do this)
- Implement the Track and Event Comparators (have a look at the given classes `MyTrackComparator` `MyDurationComparator` for a Hint how to do this)
- Implement the new classes `Album` and `MusicVideo` derived from the (also new) class `Release`.
- Create the interface `ConsoleScanable` itself and implement it for class `Track`. (see hints)
- Implement a class `DemoApp` with a main method. In the main method, create objects of the previously implemented classes. We highly recommend to create some Collections holding your Tracks respectively your Events and try to apply the Formatter and Comparators to these collections (e.g. by sorting them). Additionally try to use all methods of the `Album` class to see the full power of your implementation (e.g. adding some Tracks, trying to add / remove null values and so on) (recommended)

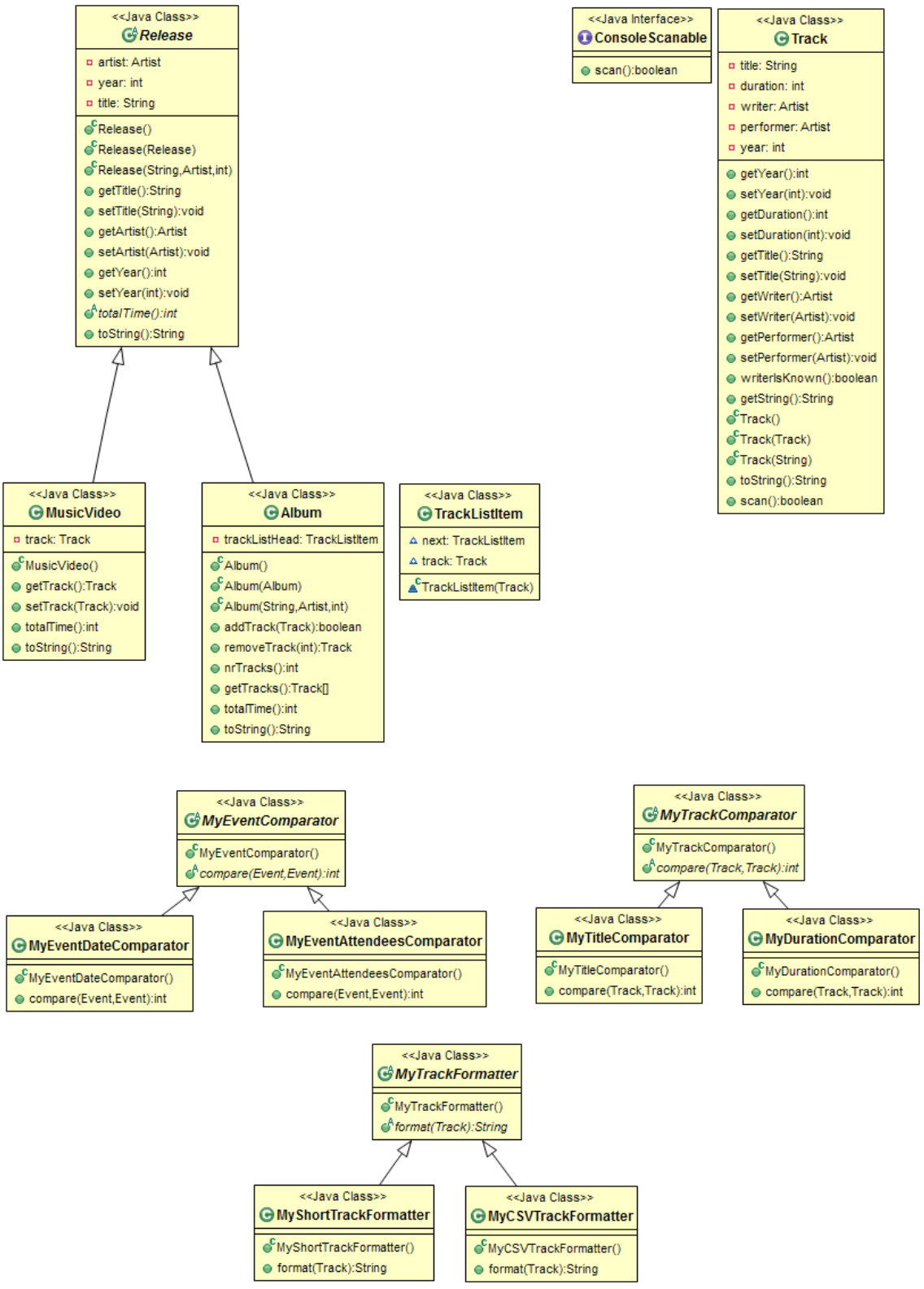

Figure 1: Classdiagram ES04 (only new classes)

For a detailed description of the fields and methods, please refer to the respective javadoc. Remember: You find new Methods introduced in the actual ExerciseSheet by checking the *Introduced in:* Tag.

Hint

This time, we provide some finished or partially implemented code as a starting point and support.

The provided finished classes can be found in the ES04_provided.zip and should be imported to your ExerciseSheet.

Below find some starting point for the `scan` Method.

You can use the following scan method as a starting point for your implementation of `ConsoleScanable` in `Track`. Simply add code for scanning/editing field `duration`.

```
@Override
    public boolean scan() {
        boolean  fieldChanged = false, objectChanged = false;
        String input;

        // scanning title
        do {
            TextIO.putf("current title: %s\n", this.title);
            TextIO.putf("enter new title (leave empty to keep):");
            input = TextIO.getlnString();

            if (input.length() == 0) { // keep old value?
                fieldChanged = false;
                break;
            }

            /*
            if (!validateTitle(title)) {
                TextIO.putf("not a valid title (%s).\n", title);
                continue;
            }
             */
            fieldChanged = true;
            break;
        } while (true);

        if (fieldChanged) {
            setTitle(input);
        }

        objectChanged = objectChanged || fieldChanged;

        fieldChanged = false; // set up for next field

        // scan next field(s)


        return objectChanged;
    }
```

Information in javadoc

You are asked to implement all Formatters, Comparators and Interfaces as well as the abstract class `Release` and the derived classes `Album` and `MusicVideo`

As in the exercise before, you will need the classes `Date` and `Venue` (you should have imported them to the MusicLandscape folder).

For some methods/fields/classes, you may find a headline **Hint** which gives you implementation hints.

Tests

The following amount of tested methods should be shown in your testng overview:

Classes	Methods
MyCSVTrackFormatter	2
MyTitleComparator	3
MyEventComparator	1
MyEventAttendeesComparator	6
MyEventDateComparator	7
Release	16
Album	11
MusicVideo	6
ConsoleScanable	5
sum	**57**

ATTENTION

- Make sure you have implemented all methods and fields with the predefined names!
- If you are not able to implement some of them because of the requirements, generate at least the methods themselves, else the tests won't compile and therefore the ExerciseSheet won't be graded.
- Never change the testfiles themselves (except the neccessary changes for package name and imports), adjustments in testfiles lead to 0 points of the exercise (as grading will be done with the original ones).
- Please note that 100 % passed tests don't mean 100% correct implementation