Hacettepe University

Computer Engineering Department

Report of Programming Assignment 2

Programming Languages: Java

Subject: Inheritance, Access Modifiers

Name-Surname: **Buse Orak**

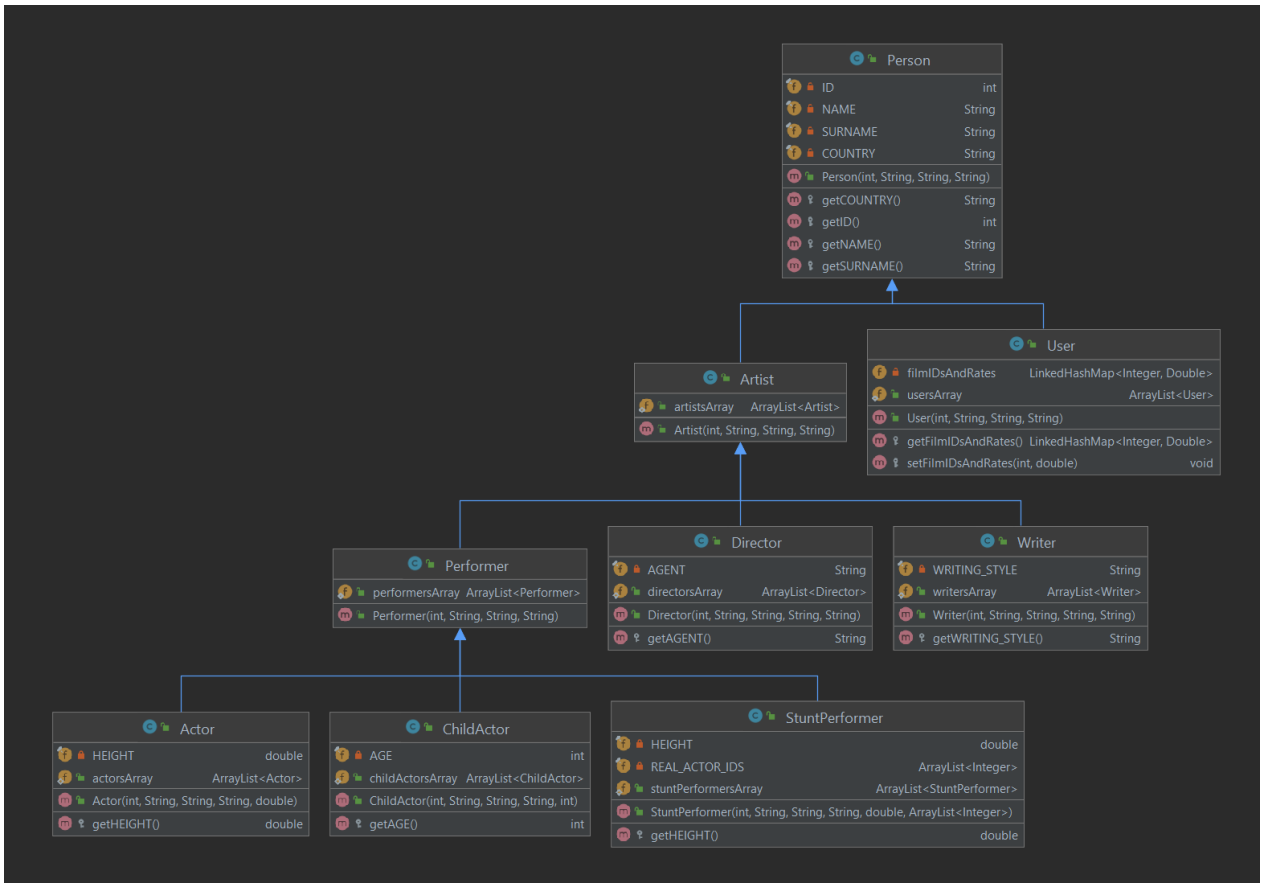
Student number: **2200356005**

BBM104: Introduction to Programming Laboratory II (Spring 2021)

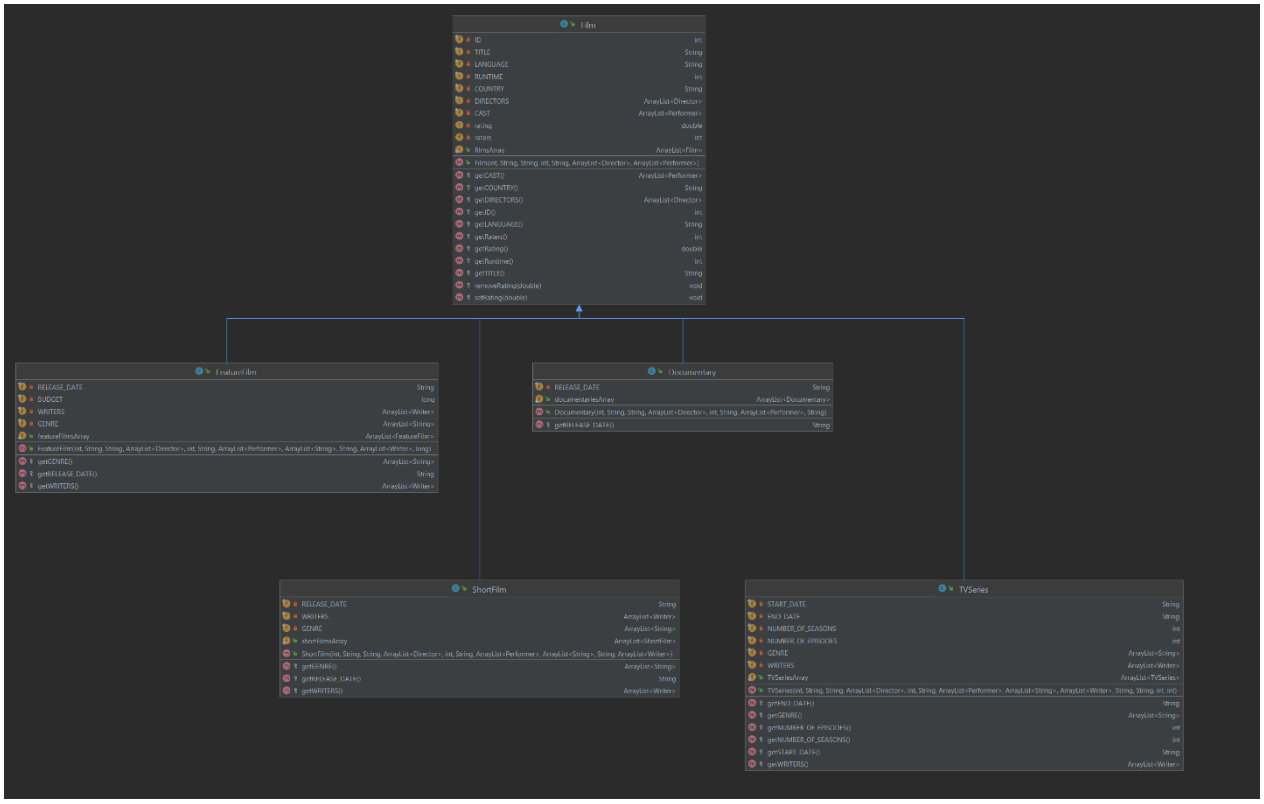
Bahar Gezici and Merve Özdeş

Due Date: 16.04.2021 (23:59)

Person Class: UML Diagram Showing Inheritance Hierarchy



Film Class: UML Diagram Showing Inheritance Hierarchy



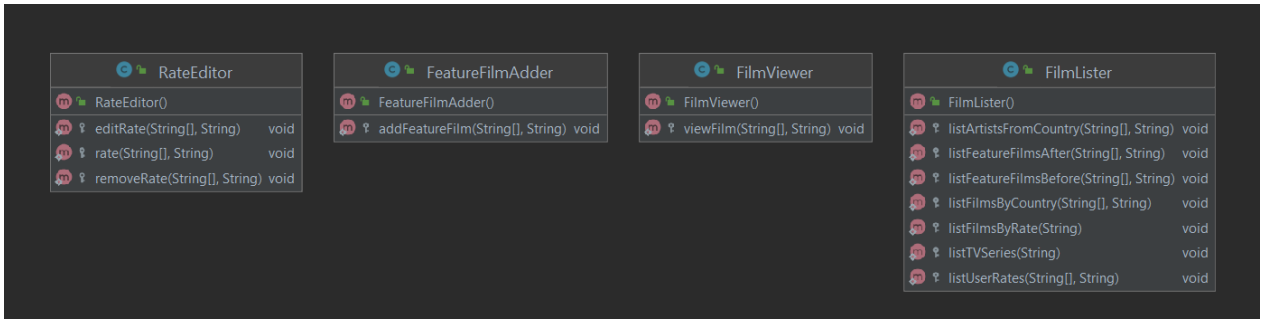
Other Classes That Do Not Have Inheritance Relationships:

UML Diagrams of the Classes MyFileIOOperations, PeopleFileReader, FilmsFileReader

(For File I/O Operations)

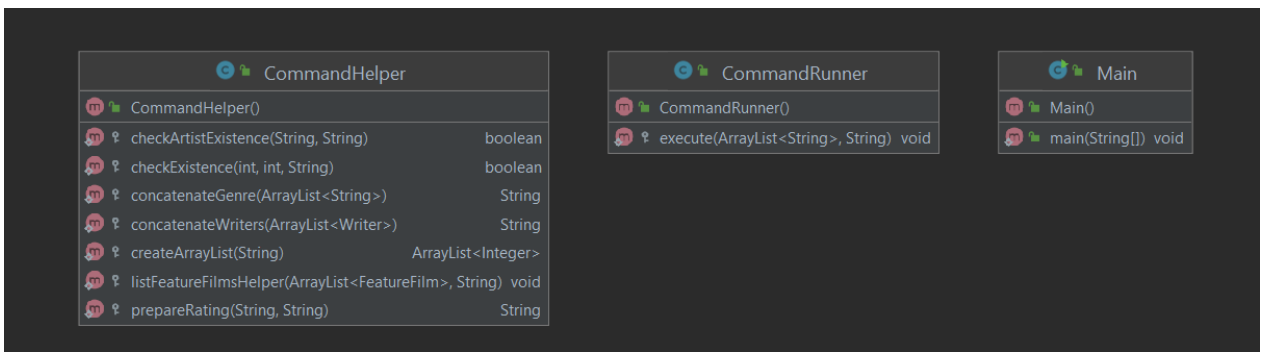
UML Diagrams of the Classes RateEditor, FeatureFilmAdder, FilmViewer, FilmLister

(For Commands)



UML Diagrams of the Classes CommandHelper, CommandRunner, Main

(For Execution of Commands and Entrance to Program)



My Solution

Class Structures:

- Person class hierarchy:
    - Person
        - User
        - Artist
            - Performer
                - Actor
                - ChildActor
                - StuntPerformer
            - Director
            - Writer
- Film class hierarchy:
    - Film
        - FeatureFilm
        - ShortFilm
        - Documentary
        - TVSeries
- Other classes that do not have inheritance relationships and their explanations:
    - MyFileIOOperations
        - contains 3 methods about file I/O operations.
    - PeopleFileReader
        - contains the method that reads the people file and creates the Person objects with respect to the inheritance hierarchy.
    - FilmsFileReader
        - contains 4 helper methods and the method that reads the films file and creates the Film objects with respect to the inheritance hierarchy.
    - RateEditor
        - contains 3 methods that are about rating films (rating, editing, and removing).
    - FeatureFilmAdder
        - contains the method that adds a new FeatureFilm object to the system.
    - FilmViewer
        - contains the method that displays Film object's information with respect to the inheritance hierarchy.
    - FilmLister
        - contains 7 methods that are about listing films according to the commands (listing artists from a specified country, listing feature films after and before a specified year, listing films according to the country information,

listing films by their rating information, listing TV series, and listing films with respect to the user's ratings).

- CommandHelper
  - contains 7 helper methods that are used by methods that perform the necessary operations for the commands.
- CommandRunner
  - contains the method that calls the other methods that execute commands according to the command file,
  - provides the chain of method calls as containing the first method invoked from the *main* method for executing commands.
- Main
  - contains the *main* method,
  - serves only as an entrance to the program by
    - initializing the program by making calls on the methods about file inputs,
    - calling the method *execution* to initiate chain of reactions in other methods in other classes.

Explanation of My Approach to The Problem:

- Design of Classes (Inheritance Mechanisms):

  Firstly, I have created super classes for people and films. Then, for a better and more efficient software practice that enables reusability and specification of code in each layer, I used inheritance mechanisms. I connected the other classes that fall under either people or films.

  Secondly, I have created the MyFileIOOperations class for general file reading and writing. Then, I have created two classes for reading the people and films file separately so that I can manage my whole project in a more organized manner.

  Thirdly, in order to execute the commands and perform necessary operations, I have created the classes RateEditor, FeatureFilmAdder, FilmViewer, and FilmLister. For the helper methods needed here to reduce copying the same code, I have created a separate class called CommandHelper. Throughout the project, this practice has provided me a better management and maintenance for command operations. Additionally, I have created a separate file called CommandRunner for, again, better maintenance and code organization reasons in which I call the necessary methods according to the commands in the commands file.

  Fourthly, I have created the Main class with the *main* method in it that provides an entrance to the program by reading input files and making the call to the method *execute*.

While designing the classes and their members, I have obeyed the Java naming conventions throughout the project by

- writing the names of fields in lower Camel case,
- writing the names of `final` instance variables in uppercase (if the identifier is more than one word, I used underscores between words.),
- writing the names of methods in lower Camel case, with the first word being a verb.

- Object Oriented Programming Design Pattern Compatibility:

  Throughout the project, I tried to organize the classes and methods in such a way that fits the OOP principles and design patterns as good as possible. For example, I chose not to call the related methods for each command inside the *main* method; on the contrary, I have only made certain primary calls to other methods there and initiated a chain of method calls, which are independent of the *main* method which only serves as an entrance/initiator. This was the main reason why I decided to create the class CommandRunner, so that I could provide and entry to the program in *main* method, keep it distinctly short, and only call the `static` method *execute* in CommandRunner that calls the other methods according to the respective command.

- Encapsulation (Access Modifier Choice, Coherence&Cohesion):

  I have used `private` non-static variables and `protected` accessors and mutators throughout the project. `private` variables are for information hiding and cutting the field access/alteration from outside the class. For the accessors and mutators I have used `protected` access modifiers for better software practice and providing the methods only to the sub classes that inherit those fields and other classes inside the package.

  I have created the constructors as `public`. The `ArrayList`s that store the objects of their classes are `public` as well since there is no issue of information hiding and encapsulation there. Even if they were somehow accessed from outside the package or by a class that does not have an inheritance relationship with that class, the information would not be able to be retrieved nor changed since both the direct (due to keyword `private`) and indirect (due to keyword `protected`) access to the fields are not allowed.

- Data Structures Used and Technical Advantage/Disadvantage Comparison:

  In the methods that perform necessary operations for their related commands, my general approach was to use data structures `ArrayList` and `LinkedHashMap`. I have used `ArrayList` for storing the objects of each class (both super classes and sub classes, when needed). By this way, I had the ability to store the reference of the objects during their creation and then iterate through them and retrieve their data afterwards. I

have used `LinkedHashMap` when I needed to link two information together as one instance variable. Another reason why I chose to use `LinkedHashMap` was its time complexity. It is a lot less time-consuming for retrieving information, as I will discuss below in detail.

There are positive and negative consequences of using `ArrayList` and `LinkedHashMap`:

- Memory Complexity:

  `LinkedHashMap` takes up more memory than `ArrayList` in general since `ArrayList` stores only the value by managing the indexing internally for each element, but `LinkedHashMap` stores both the key and value for each entry. Since I used the `ArrayList`s more than `LinkedHashMap`s in my implementation, I have increased the efficiency and performance in terms of memory. The negative side of this is that I have 13 `ArrayList`s in total to store the objects of each class, which still takes up memory in a linear manner (O(n)).

- Time Complexity:

  As far as I have understood while doing this assignment, the time-consumption point of view needs algorithm-specific analysis. The retrieve of information in an `ArrayList` has time complexity of O(1) if the index of the element to be returned is known (method: *get(index)*). In my implementation, since the `ArrayList`s mostly consisted of objects, most of the time I did not know the index of the elements. Therefore, I had to iterate through the elements until I finally found the object I was looking for. This operation has O(n) time complexity, depending on in which index the object is located. To minimize this negative effect, I have named the outermost loop and used the `break` keyword after the operation is over so that the iteration does not continue and take much more time for no reason. However, adding new elements to the `ArrayList` has a time complexity of O(1) (method: *add(element)*) if it is appended and O(n) if it is inserted (method: *add(index, element)*). Since I have always appended the new elements to the end of the list but did not insert, I gained advantage of using `ArrayList`s.

  In `LinkedHashMap`, however, the key was known most of the time in my implementation, therefore the time complexity of retrieving the value is O(1) (method: *get(index)*). All of the time, appending a new entry to a `LinkedHashMap` (method: *put(key, value)*) has a time complexity of O(1) as well. These two properties of `LinkedHashMap`s were huge advantages in terms of time consumption for my implementation.

Comments:

This project has helped me get more familiar with inheritance mechanisms and the data structures `HashMap` and `LinkedHashMap`.

What I liked the most about this assignment is that it was close to being a real-life project, which kept me very motivated to not only write the code but read and learn more about the OOP principles, conventions, and design patterns to develop my skills in a professional way for the future.

What strained me the most was to come up with ways to improve my algorithm in terms of its efficiency and time and memory complexity, which I have discussed in detail above.

Works Cited

Java™ Platform, Standard Edition 8 API Specification. *Class HashMap<K,V>*. [online] Available at: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html [Accessed April 12, 2021].

Java™ Platform, Standard Edition 8 API Specification. *Class LinkedHashMap<K,V>*. [online] Available at: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html [Accessed April 12, 2021].

Java™ Platform, Standard Edition 7 API Specification. *Class ConcurrentModific*ationException. [online] Available at: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html [Accessed April 12, 2021].

Deitel, Paul, and Harvey Deitel. *Java - How to Program*. Pearson Education, 2012.

"Time Complexity of Java Collections." Baeldung, 19 July 2019, www.baeldung.com/java-collections-complexity. Accessed 12 Apr. 2021.