Movie Database Project – Student Assignment

Course: Object Oriented Programming (POO 1)
Class: ING 2 Section A
Instructor: Djeraoui Mouna
Due Date:
Total Points: 15 Points

1. **Project Overview**

- Objective:

Students are tasked with creating an interactive Media Database application where users can:

  - Add different types of media (movies, TV series, documentaries) to their collection
  - Search and filter their collection
  - Rate media and mark them as watched
  - View statistics about their collection
- Learning Objectives:
  - Apply Object-Oriented Programming (OOP) concepts
  - Practice inheritance and polymorphism
  - Implement method overloading
  - Create an interactive console application
  - Use proper encapsulation techniques

2. **System Architecture**

The system is organized into the following conceptual layers:

a. Models Layer: Contains the classes representing different types of media.
b. Services Layer: Provides management functionality for the media collection.
c. UI Layer: Handles user interaction via a console interface.

Class Relationships:

a. Media is the base class.
b. Movie, Series, and Documentary inherit from Media, demonstrating

Polymorphism.

a. Series contains a list of Episode objects representing individual episodes.
b. MediaDatabase manages a collection of Media objects and provides search, filter, and statistics functionality.
c. Application interacts with the user, calling methods in MediaDatabase and displaying results.

3. **Class Specifications**

a. **Media.java (Base Class)**

Purpose: Represents a basic media entry.

Attributes (all private):

- title (String)
- director (String)
- releaseYear (int)
- genre (String)
- rating (double)
- isWatched (boolean)

Constructors (Overloaded):

- Media(title, director, releaseYear)
- Media(title, director, releaseYear, genre)

Methods:

- Getters and setters for all attributes
- markAsWatched()
- getMediaType()
- displayInfo()
- displayInfo(boolean showRating)

b. **Movie.java (Subclass of Media)**

Purpose: Represents a movie.
Constructor:

- Movie(title, director, releaseYear, genre)

Methods:

- Override getMediaType() -> 'Movie'
- displayInfo() and displayInfo(boolean showRating)

c. **Series.java (Subclass of Media)**

Purpose: Represents a TV series.
Additional Attributes:

- episodes (List<Episode>) – List of episodes in the series
- nbOfSeasons

Constructor:

- Series(title, director, releaseYear, genre, episodes, nbOfSeasons)

Methods:

- Override getMediaType() -> 'TV Series'
- getTotalEpisodes()
- displaySeriesInfo()
- displaySeriesInfo(boolean showTotalEpisodes)

d. **Episode.java (Associated with Series)**

Purpose: Represents a single episode in a TV series.
Attributes (all private):

- title (String)
- duration (int) – in minutes
- episodeNumber (int)
- seasonNumber (int)

Constructor:

- Episode(title, duration, episodeNumber, seasonNumber)

Methods:

- Getters and setters
- displayEpisodeInfo()

e. **Documentary.java (Subclass of Media)**

Purpose: Represents a documentary.

Additional Attributes:

- topic (String)
- isEducational (boolean)

Constructor:

- Documentary(title, director, releaseYear, topic, isEducational)

Methods:

- Override getMediaType() -> 'Documentary'
- getEducationalValue()

f. **MediaDatabase.java (Service Class)**

Purpose: Manages the collection of media.

Attributes:

- mediaList (ArrayList<Media>)

Key Methods:

Add Methods (Overloaded):

- addMedia(Media media)
- addMedia(title, director, year) – creates and adds media

Search Methods (Overloaded):

- searchByTitle(String title)
- searchByDirector(String director)
- searchByGenre(String genre)

Management Methods:

- markAsWatched(String title)
- rateMedia(String title, double rating)
- getUnwatchedMedia()
- getWatchedMedia()
- displayAllMedia()

Statistics Methods:

- getAverageRating() :

Calculates and returns the average rating for all media in the collection. This method iterates through the list of media, adds up all ratings, divides by the total number of media, and returns the average.

- getMediaCount()

Simply returns the total number of media items stored in the database. This is equivalent to mediaList.size().

- getMediaByType(String type)

Returns a list of all media of a specific type (e.g. ‘Film’, ‘TV Series’, ‘Documentary’). This method scans the collection, checks the type of each media item (via getMediaType()), and collects those that match the requested type.

g. **Application.java (UI/Interactive Class)**

Purpose: Provides a console interface for the user.

Attributes:

- database (MediaDatabase)
- scanner (Scanner)
- isRunning (boolean)

Responsibilities:

Present menu options

- Handle user input and validation
- Call appropriate methods in MediaDatabase
- Display results to the user

Interactive Flows:

- Adding media (Movie, Series, Documentary)
- Searching
- Marking as watched
- Rating media
- Viewing statistics, watched/unwatched lists

4. Implementation Requirements

Phase 1 – Core Classes:

- Implement Media with full encapsulation
- Implement Movie, Documentary, Series, Episode
- Test inheritance, polymorphism, and method overloading
- Implement MediaDatabase class

Phase 2 – Interactive Application:

- Create Application class with Scanner-based UI
- Implement all menu options
- Add input validation
- Test full user flow

Phase 3 – Enhancements (Optional):

- Add file persistence (save/load data)
- Add additional filters, sorting, and advanced statistic

Formula 1 Database Project – Student Assignment

Course: Object Oriented Programming (POO 1)
Class: ING 2 Section A
Instructor: Djeraoui Mouna
Due Date:
Total Points: 15 Points

1. **Project Overview**

- Objective:

Students are tasked with creating an interactive Formula 1 Database application where users can:S

  - Add different types of F1 entities (drivers, teams, races) to their collection
  - Search and filter entities by various criteria (e.g., driver name, team, race year)
  - Record race results and driver/team statistics
  - View statistics and rankings

- Learning Objectives:
  - Apply Object-Oriented Programming (OOP) concepts
  - Practice inheritance and polymorphism
  - Implement method overloading
  - Create an interactive console application
  - Use proper encapsulation techniques

2. **System Architecture**

The system is organized into the following conceptual layers:

a. Models Layer: Contains the classes representing different Formula 1 entities.
b. Services Layer: Provides management functionality for the F1 database.
c. UI Layer: Handles user interaction via a console interface.

Class Relationships:

a. F1Entity is the base class.
b. Driver, Team, and Race inherit from F1Entity, demonstrating polymorphism.
c. Driver contains a list of RaceResult objects.
d. Race contains a list of RaceResult objects.
e. F1Database manages collections of Drivers, Teams, and Races.
f. Application interacts with the user, calling methods in F1Database and displaying results.

3. **Class Specifications**

a. **F1Entity.java (Base Class)**

Purpose: Represents a generic Formula 1 entity.
Attributes (all private):

- name (String)
- country (String)
- activeYears (String)

Constructors (Overloaded):

- F1Entity(name, country)
- F1Entity(name, country, activeYears)

Methods:

- Getters and setters for all attributes
- getEntityType() -> Returns 'Generic F1 Entity' (to be overridden)
- displayInfo()

b. **Driver.java (Subclass of F1Entity)**

Purpose: Represents a Formula 1 driver.

Additional Attributes

- team (Team)
- number (int)
- raceResults (List<RaceResult>)

Methods:

- getEntityType()
- addRaceResult(RaceResult result)
- getTotalPoints()
- displayDriverInfo(boolean showResults)

c. **Team.java (Subclass of F1Entity)**

Purpose: Represents a Formula 1 team.

Additional Attributes:

- drivers (List<Driver>)
- championshipPoints (int)

Methods:

- getEntityType()
- addDriver(Driver driver)
- calculateTeamPoints()
- displayTeamInfo()

d. **Race.java (Subclass of F1Entity)**

Purpose: Represents a Formula 1 race.

Additional Attributes:

- year (int)
- location (String)
- raceResults (List<RaceResult>)

Methods:

- Override getEntityType() -> 'Race'
- addRaceResult(RaceResult result)
- displayRaceResults()

e. **RaceResult.java (Associated with Race and Driver)**

Purpose: Represents a driver's result in a race.

Attributes (all private):

- driver (Driver)
- team (Team)
- position (int)
- qualifyingPosition (int)
- points (int)

Methods:

- Getters and setters
- displayResult()

f. **F1Database.java (Service Class)**

Purpose: Manages collections of F1 entities.

Attributes:

- drivers (List<Driver>)
- teams (List<Team>)
- races (List<Race>)

Methods:

- Add methods (overloaded) for Driver, Team, Race
- Search methods by name, country, or year
- Get driver and team standings
- Display all entities
- Record race results and calculate statistics

g. **Application.java (UI/Interactive Class)**

Purpose: Provides a console interface for the user.

Attributes:

- database (F1Database)
- scanner (Scanner)
- isRunning (boolean)

Main Menu Options:

- Add a new driver/team/race
- Search entities
- View all drivers
- View all teams
- View all races
- Record race results
- View driver standings
- View team standings
- Exit

4. Implementation Requirements

Phase 1 – Core Classes:

- Implement F1Entity with full encapsulation
- Implement Driver, Team, Race, and RaceResult
- Test inheritance, polymorphism, and method overloading
- Implement F1Database class

Phase 2 – Interactive Application:

- Create Application class with Scanner-based UI
- Implement all menu options
- Add input validation
- Test full user flow

Phase 3 – Enhancements (Optional):

- Add file persistence (save/load data)
- Add additional filters, sorting, and advanced statistics

Sample User Flow

```
=== F1 DATABASE MENU ===
Enter choice: 1
What would you like to add?
1. Driver
2. Team
3. Race
Enter choice: 1
Enter driver name: Lewis Hamilton
Enter country: UK
Enter driver number: 44
Select team: Mercedes
Driver successfully added!
```

1. Java Lists: The Simple Explanation

Think of an **ArrayList** like a **resizable shopping list** in Java. It's a container that can hold multiple objects (like Driver, Team, or Race objects in your project), and it automatically grows when you add more items.

a. Basic Declaration:

```
java

import java.util.ArrayList;
import java.util.List;

// Create a list that holds Driver objects
List<Driver> drivers = new ArrayList<>();
```

Essential Methods:

```
add(element) → Add item to end: drivers.add(new Driver("Lewis Hamilton"));

get(index) → Get item by position (0=first): Driver d = drivers.get(0);

remove(index) → Remove by position: drivers.remove(0);

size() → Get number of items: int count = drivers.size();

isEmpty() → Check if empty: if (drivers.isEmpty()) { ... }
```

Simple Loop:

```
// View all drivers
for (Driver driver : drivers) {
   System.out.println(driver.getName());
}
```

1. Install Git on Your PC

Step 1: Download Git from https://git-scm.com/downloads
Step 2: Install Git

- Run the installer and select default options.
- Important: Choose "Use Git from the Windows Command Prompt".
- Keep VS Code as the default editor if available.

Step 3: Verify Installation
Open a terminal and run:

```
git –version
```

You should see a version number.

2. Create a GitHub Account

Step 1: Go to https://github.com and sign up.
Step 2: Enter email, username, and password. Verify your email.
Step 3: Optional: set up SSH keys (we will use HTTPS for simplicity).

3. Link GitHub to VS Code

Step 1: Install VS Code from https://code.visualstudio.com
Step 2: Install GitHub extension

- Open Extensions (Ctrl+Shift+X) and install "GitHub Pull Requests and Issues".

Step 3: Sign in to GitHub in VS Code

- Open Command Palette (Ctrl+Shift+P) → GitHub: Sign in → Follow prompts.

4. Configure Git Locally

```
git config --global user.name "Your Name"
git config --global user.email “your.email@example.com”
```

Check config:

```
git config –list
```

5. Create a Repository on GitHub

Step 1: Go to GitHub → New Repository

- Name: project-name
- Set project visibility to **Private**
- Optional: Initialize with README

Step 2: Copy HTTPS URL (example: https://github.com/username/project-name.git)

6. Connect Local Project to GitHub

Existing Project:

```
git init
git add .
git commit -m "Initial commit"
git remote add origin https://github.com/username/project-name.git
git branch -M main
git push -u origin main
```

7. Working on the Project

1. Make changes in VS Code.
2. Stage and commit changes:

```
git add .
git commit -m "Describe changes"
```

3. Push to GitHub:

```
git push origin main
```

4. Pull latest changes if collaborating:

```
git pull origin main
```

8. Work submission

1. Go to your repository settings
2. Collaborators
3. Add people, myself (https://github.com/MonaMo02) and Dr. Mekahlia (https://github.com/Fmekahlia) as collaborators.