Project 2

Computer Science 2334

Spring 2016

This project is group work. Group composition will be determined later this week. Each student should complete and submit the draft design individually.

User Request:

"Create a sortable and searchable data system for movies and broadcast series."

Milestones:

1.	Use keyboard input to get information from the user.	*5 points*
2.	Use text file I/O to read and write text files.	*10 points*
3.	Create classes to store data on movies and broadcast series. Note that you should create any additional classes (abstract and/or concrete) and/or interfaces you deem necessary to arrive at a good design.	*10 points*
4.	Implement both the **Comparable** and **Comparator** interfaces to compare one movie or broadcast series to another.	*10 points*
5.	Use a **List** to store, retrieve, and display data related to movies and broadcast series as described below.	*15 points*
6.	Use the `sort()` and `binarySearch()` methods from the **Collections** class to sort and search for data related to movies and broadcast series as described below.	*20 points*
►	Develop and use a proper design. (See Milestone 3, above.)	*15 points*
►	Use proper documentation and formatting.	*15 points*

Description:

Movies have certain basic features, such as title, release year, and venue (movie theater, TV, or video). Similarly, broadcast series[1] have basic features such as title and starting year. However, series are more complex in that they have episodes that have their own release years and may have episode titles and/or season and episode numbers. For this project, you will create a system that models these objects.

As with Project 1, you will put together several techniques and concepts learned in CS 1323 and some new techniques to make an application. This application will allow users to search through data on movies and series. As with Project 1, we will call this application *MDb*. However, since we are now modeling two types of media, you can now think of the "M" as standing for "media" rather than "movie." Note that much of the code you write for this program could be reused in more complex applications, as we will see in later assignments.

MDb will first ask the user for a file in which movie data is stored. This should be done using the technique described in the section below entitled Reading Input from the Keyboard. MDb will then read in the specified data file and store the data. Each line in this data file describes a particular movie. The format of this file is the same as for Project 1. Note that while there may be more than one movie in this database with the same title, the title together with the information in the first set of parentheses after the title (release year and possible Roman numeral, as described in Project 1) will be unique.

1 Hereafter we will use the term "series" to include TV series and other episodic broadcast media, such as web shows.

MDb will then ask the user for a file in which series data is stored. This should also be done by reading input from the keyboard. MDb will then read the specified file and store the data. The series data file format is described below in the section entitled Input Format.

Once both data files are loaded, MDb will enter a loop where it asks the user questions. As with the file name, answers to these questions should come from the keyboard as described in the section entitled Reading Input from the Keyboard. Depending on each answer given, MDb will proceed through a series of questions before displaying data to the user and starting over with the first question again.

The first question all users will be asked is "Search (m)ovies, (s)eries, or (b)oth?" The second question is "Search (t)itle, (y)ear, or (b)oth?" If the user answered "t" or "b" to the second question, MDb will ask, "Search for (e)xact or (p)artial matches?" If the user has answered either "s" or "b" to the first question and either "t" or "b" to the second question, MDb will ask, "Include episode titles in search and output (y/n)?" If the user answered "t" or "b" to the second question, MDb will ask "Title?" If the user answered (y) or (b) to the second question, MDb will ask "Year(s)?" Finally, MDb will ask, sort by (t)itle or (y)ear?"

For most of the questions, MDb should only accept the one letter responses indicated as possibilities within parentheses for each question. For "Title?" any string is a valid response (but may not match with any titles in the data). For "Year(s)?" the user may enter a single year, multiple years separated by commas, or a range of years by entering start year-end year. (For example, the user could enter "1984" to indicate just the one given year, "1984, 1976, 2014, 2005" to indicate the four years listed, or "1998-2002" to indicate all years in the range from 1998 to 2002 inclusive.)

In all cases, once the user has answered all questions, data will be displayed to the console as described in the section entitled Output Format. Once data is displayed to the user, the user will be asked "Save (y/n)?" If the user chooses "y," MDb will prompt for a file name to which the output should be saved, then save the data in a file with the given name in the same format as it was displayed to the user. After saving or skipping, MDb will ask the user "Continue (y/n)?" If the user enters "y," MDb will return to the first question ("Search (m)ovies, TV (s)hows, or (b)oth?"). If the user enters "n," MDb will thank the user for using MDb and exit gracefully.

If the user response to any question is incorrect (that is, does not allow MDb to proceed to the next step), MDb should present the user with a polite message stating that and asking the user to give new input.

Learning Objectives:

Sorting and Searching:

Sorting data can be useful to users because the output may be organized in a way that makes it easier to use. It can also be useful to software developers because it can improve the efficiency of their software. Consider finding information about a person based on name. If the data structure holding the data is unsorted, you need to do a linear search through it to find an entry. However, if the data structure is sorted based on name, you can do a binary search instead. A binary search will, in most cases, take far fewer comparisons to find the desired entry than a linear search. You should ensure that your program uses a binary search when searching for an exact name. (Why only for exact names? Answer this question in your writeup on the project.)

Note that a collection in Java can have at most one *natural ordering*. You should determine an appropriate natural ordering for movies and for series (which must implement **Comparable**) and define the `compareTo()` method(s) to use that ordering. The other sort option will need to be implemented using the `compare()` method that comes from implementing **Comparator**.

Input/Output Formats:

Input Format

The input file format for movies is the same as the input file format from Project 1. See that project for details. The input file format for series consists of one line for the general series information followed by zero or more lines, one per episode. Each series line consists of the series title in quotation marks followed by a space followed by the series start year in parentheses followed by one or more tabs followed by the range of years the series was produced as start year-end year. For example:

```
"Doctor Who" (1963)                1963-1989
```

This shows that the series name is "Doctor Who," it started in 1963, and it ran from 1963 to 1989. Note that as with movie titles, the series title may not be unique. However, the series title together with the start year will be unique. For example, the data file might also contain the following line:

```
"Doctor Who" (2005)                2005-????
```

Here, the series title is the same but the start year in parentheses is different, indicating that this is a different series from the previous series, although the title is the same. Also note that, as with the movie data, an indeterminate year may be indicated by a series of four question marks. (Here, the ending year is indeterminate because the series is still in production and we don't know when it will end because we don't have a time machine.)

Each episode line generally consists of the series title followed by a space followed by the series start year followed by a space followed by an opening curly brace followed by the episode title (if any) followed by the episode season and episode numbers (if any) in parentheses followed by a closing curly brace followed by one or more tabs followed by the year the episode first aired. The format for season and episode numbers is a number sign followed by the season number followed by a period followed by the episode number. For example:

```
"Doctor Who" (1963) {A Bargain of Necessity (#1.41)}    1964
```

This shows that this is an episode from the "Doctor Who" series that started in 1963, that the episode's title is "A Bargain of Necessity," that this episode is episode 41 of season 1, and that this episode first aired in 1964. Note that the combination of episode name and number will be unique within a series.

Note that not all TV show episodes have episode names available. For example:

```
"Doctor Who" (2005) {(#10.1)}              2016
```

This shows that the title of episode 1 of season 10 of the 2005 "Doctor Who" series does not yet have its title available in the data file. Similarly, not all episodes have episode names. For example:

```
"Doctor Who" (2005) {A Ghost Story for Christmas}   2009
```

This shows that the episode entitled "A Ghost Story for Christmas" from the 2005 "Doctor Who" series is, for some reason, lacking a season and episode number.

Finally, it is possible for an episode to be suspended and never (or not yet) finished. In that case, the word "SUSPENDED" will appear inside two sets of curly braces after the curly braces enclosing the episode name and/or season and episode numbers. Also the year it was aired will be listed as four question marks. For example:

```
"Doctor Who" (1963) {Blink} {{SUSPENDED}}       ????
```

This shows that there was an episode from the original 1963 "Doctor Who" series that was suspended and never completed. That episode was entitled "Blink."

Output Format

The output format for the data depends on what data was requested and what was found.

The first several lines of the output will indicate the search and sort options requested, as follows. The first line will indicate the type of media search for, saying, “SEARCHED ” followed by a list of the media types searched (with the options being “MOVIES,” “SERIES,” and “EPISODES”). The second line will indicate the title search string given, if any, and whether the search was partial or exact, saying either “PARTIAL TITLE: ” or “EXACT TITLE: ” followed by the title search string given by the user. If the user did not search on title, MDb will indicate that by saying “TITLE: Any.” The third line will indicate the years searched for, if any, saying “YEARS: ” followed by the search years specified by the user or “YEARS: Any” if the user did not search on years. The fourth line will indicate the sort preference of the user, either “SORTED BY TITLE” or “SORTED BY YEAR” as appropriate. The fifth line will be a break between the header and the data found, consisting of 80 equals signs. The sixth and following lines will be the movies and TV shows found during the search. If no data matches the search, the sixth line will simply say, “NO MATCHES.”

Each movie (if any) will be given on a single line, formatted as follows: The word “MOVIE” (in all capitals), possibly followed by a space and a parenthetical note on format (“straight to video” or “TV”) as appropriate, followed by a colon and a space, followed by the movie name, followed by a space and the release year (and possible Roman numeral) in parentheses.

Each series (if any) will be given on a single line, formatted as follows: The word “SERIES” (in all capitals), followed by a colon and a space, followed by the series title, followed by the range of years broadcast given as start year-end year in parentheses.

Each episode (if any), will be given on a single line, formatted as follows: The word “EPISODE” (in all capitals), followed by a colon and a space, followed by the series title, followed by a colon and a space, followed by the episode title, followed by the broadcast year in parentheses. If the episode has no title, the season and episode numbers will be used instead, formatted as they are in the input file (parentheses and all). This is true for both the search and the output. However, if there is a title for the episode, the season and episode numbers will be omitted from the search and from the output.

The lines of the search result will either appear sorted first on title and then on year or first on year and then on title, depending on whether the user selected “t” or “y” for the sort option, respectively.

If any year is indeterminate, it will be given in the output as “UNSPECIFIED” and sorted to the end of the list. For ordering by years when a range is involved, MDb should use the start year of the range, regardless of the end year.

Examples:

If the user loaded the files “StarTrekMovies.txt” and “StarTrekTV.txt” and searched both movies and TV shows (including episodes) for the exact title “Star Trek,” left the year unspecified, and asked to have the output sorted by either title or year, the output would be:

```
SEARCHED MOVIES, TV SERIES, AND TV EPISODES
EXACT TITLE: STAR TREK
YEARS: Any
SORTED BY YEAR
================================================================================
SERIES: Star Trek (1966-1969)
MOVIE: Star Trek (2009)
```

If the user loaded the files “StarTrekMovies.txt” and “StarTrekTV.txt” and searched both movies and TV shows (including episodes) for the partial title “New,” left the year unspecified, and asked to have the output sorted by title, the output would be:

```
MOVIE (TV): Inside the New Adventure: Star Trek – Voyager(1995)
MOVIE (straight to video): Star Trek: A New Vision (2009)
EPISODE: Star Trek: Deep Space Nine: The Emperor's New Cloak (1999)
SERIES: Star Trek: New Voyages (2004-UNSPECIFIED)
EPISODE: Star Trek: New Voyages: Blood and Fire: Part One (2008)
EPISODE: Star Trek: New Voyages: Blood and Fire: Part Two (2009)
EPISODE: Star Trek: New Voyages: Bread and Savagery (UNSPECIFIED)
EPISODE: Star Trek: New Voyages: Center Seat (2006)
EPISODE: Star Trek: New Voyages: Come What May (2004)
EPISODE: Star Trek: New Voyages: Enemy: Starfleet! (2011)
EPISODE: Star Trek: New Voyages: Going Boldly (2012)
EPISODE: Star Trek: New Voyages: In Harms Way (2004)
EPISODE: Star Trek: New Voyages: Kitumba (2013)
EPISODE: Star Trek: New Voyages: Mind-Sifter (2014)
EPISODE: Star Trek: New Voyages: No Win Scenario (2011)
EPISODE: Star Trek: New Voyages: Origins (UNSPECIFIED)
EPISODE: Star Trek: New Voyages: Origins (UNSPECIFIED)
EPISODE: Star Trek: New Voyages: The Child (2012)
EPISODE: Star Trek: New Voyages: The Holiest Thing (2014)
EPISODE: Star Trek: New Voyages: The Protracted Man (UNSPECIFIED)
EPISODE: Star Trek: New Voyages: To Serve All My Days (2006)
EPISODE: Star Trek: New Voyages: Torment of Destiny (UNSPECIFIED)
EPISODE: Star Trek: New Voyages: World Enough and Time (2007)
```

Implementation Issues:

File I/O:

To perform output to a file, use the **FileWriter** class with the **BufferedWriter** class. An example follows. You will, of course, need to modify this example to use the name of the file specified by the user and to output the data in the format described above.

```
    FileWriter outfile = new FileWriter("output.txt");
    BufferedWriter bw = new BufferedWriter(outfile);
    bw.write("This is a test -- did it work?");
    bw.newLine();
    bw.close();
```

When you have finished writing to a file, you must remember to close it, or the file won't be saved. If you fail to close the file, it will be empty!

Remember to add '`throws IOException`' to the signature of any method that uses a **FileWriter** or **BufferedWriter** or that directly or indirectly calls a method that performs File I/O.

Reading Input from the Keyboard:

In order to get information from the user for this project, you need to read input from the keyboard. This can be done using the **InputStream** member of the **System** class, that is named “`in`”. When this input stream is wrapped with a **BufferedReader** object, the `readLine()` method of the **BufferedReader** class can be used to read and store all of the characters typed by the user into a **String**. Note that `readLine()`

will *block* until the user presses the Enter key, that is, the method call to `readLine()` will not return until the user presses the Enter key.

The following code shows how to wrap and read strings from `System.in` using an **InputStreamReader** and a **BufferedReader**.

```
BufferedReader inputReader = new BufferedReader(
                                    new InputStreamReader( System.in ) );
System.out.print( "Type some input here: " );
String input = inputReader.readLine();
System.out.println( "You typed: " + input );
```

You need to add '`throws IOException`' to the signature of any method that uses or that directly or indirectly calls a method that uses a **BufferedReader** or **InputStreamReader**.

How to Complete this Project:

Preliminary Design:

1 Before, during, and after the lab session, you should work with your partner to determine the classes, variables, and methods needed for this project and their relationship to one another. This will be your preliminary design for your software.

1.1 Be sure to look for nouns in the project description. More important nouns describing the items of interest to the "customer" should probably be incorporated into your project as classes and objects of those classes. Less important nouns should probably be incorporated as variables of the classes/objects just described.

1.2 Be sure to look for verbs in the project description. Verbs describing behaviors of the desired objects and the system as a whole should probably be incorporated into your project as methods.

1.3 Also look for adjectives, if any, in the project description. Adjectives often describe features of objects that could be incorporated into your project as interfaces to be instantiated by your classes.

1.4 Write down these nouns, verbs, and adjectives (if any), along with their corresponding classes/variables, methods, and interfaces (if any).

1.5 Next, use UML class diagrams as tools to help you establish proper relationships between your classes, variables, methods, and interfaces (if any).

2 Once you have completed your UML design, create Java "stub code" for the classes specified in your design. Stub code is the translation of UML class diagrams into code. It will contain code elements for class names (potentially including names for abstract classes and interfaces), variable names, and method names; relationships between classes such as inheritance, composition, and aggregation as appropriate; variable types; and method signatures. Stub code does not, however, contain method bodies (except for return statements for methods that return values or object references – these should return placeholders such as `null`). Because we are using object-oriented design, in which the inner workings of one class are largely irrelevant to the classes with which it interfaces (that is, we are using encapsulation), we do not need to complete the implementation of any class until the design is complete.

3 Add comments to your stubbed code as specified in the documentation requirements posted on the class website. Run your commented stubbed code through Javadoc as described in the Lab 2 slides. This will create a set of HTML files in a directory named "docs" under your project directory.

4 Create unit tests using JUnit for all of the non-trivial units (methods) specified in your design. There should be at least one test per non-trivial unit and units with many possible categories of input and output should test each category. (For example, if you have a method that takes an argument of type `int` and behaves differently based on the value of that `int`, you might consider testing it with a large positive `int`, and small positive `int`, zero, a small negative `int`, and a large negative `int` as these are all likely to test different aspects of the method.)

5 At the end of the first week, you will turn in your preliminary design documents (see ***Due Dates and Notes***, below), which the TA will grade and return to you with helpful feedback on your preliminary design. **Please note:** You are encouraged to work with the instructor and the TAs during their office hours during the design week to get feedback throughout the design process as needed.

Final Design and Completed Project

6 Using feedback from the instructor and TAs as well as your own (continually improving) understanding of OO design, revise your preliminary UML design.

7 Make corresponding changes to your code, including its comments.

8 Make corresponding changes to your unit tests.

9 Create a new set of Javadoc files using Eclipse and inspect them to make sure your final design is properly documented in your source code.

10 Implement the design you have developed by coding each method you have defined. A good approach to the implementation of your project is to follow the project's milestones in the order they have been supplied. If you find that your design does not allow for the implementation of all methods, repeat steps 6, 7, 8, and 9.

11 Test each unit as it is implemented and fix any bugs.

12 Test the overall program and fix any bugs.

13 Submit your project (see ***Due Dates and Notes,*** below).

Extra Credit Features:

You may extend this project with more search features for an extra 5 points of credit. Think of ways to enable a wider range of searches to be used, such as searching based on regular expressions or wild cards. You could also revise user interface elements. If you revise the user interface, you **must** still read the file name from the keyboard and the data from the text files.

To receive the full five points of extra credit, your extended feature must be novel (unique) and it must involve effort in the design and integration of the feature into the project and the actual coding of the feature. Also, you must indicate, on your final UML design, the portions of the design that support the extra feature(s); and you must include a write-up of the feature(s) in your milestones file. The write-up must indicate what the feature is, how it works, how it is unique, and the write-up must cite any outside resources used. If you create any non-trivial units in your extra credit work, you must create appropriate unit tests for them.

Due Dates and Notes:

Note that both the preliminary design and the final project are to be submitted electronically. Paper copies will not be submitted. The UML should preferably be in PDF format, although high resolution PNG or JPG would be acceptable alternatives. The list of nouns, verbs, and adjectives and their corresponding classes/variables, methods, and interfaces should be in PDF format.

Your preliminary design (list of nouns, verbs, and adjectives; UML; stub code; detailed Javadoc documentation; and unit tests) is due on **Friday, 19 February 2016**. Submit the project archive following the steps given in the submission instructions **by 11:00 pm**.

The final version of your project including final design (UML, Javadoc, unit tests) and final implementation is due on **Friday, 29 February 2016**. Submit the project archive following the steps given in the submission instructions **by 11:00pm**.

You may write your program from scratch or may start from programs for which the source code is freely available on the web or through other sources (such as friends or student organizations). If you do not start from scratch, you must give a complete and accurate accounting of where all of your code came from and indicate which parts are original or changed, and which you got from which other source. Failure to give credit where credit is due is academic fraud and will be dealt with accordingly.

The preliminary design will be completed by each student independently. Each student should turn in their own copy of the preliminary design. The implementation of the program, however, will be completed as a group. The group should turn in only one (1) copy of the final, completed assignment. This should contain the names and student ID numbers of the group members on the cover sheet.

Each group member is required to contribute equally to each project, as far as is possible. You must thoroughly document which group members were involved in each part of the project.