

Project 1
Computer Science 2334
Spring 2016

This project is individual work. Each student must complete this assignment independently.

User Request:

“Create a simple movie data system.”

Milestones:

- | | |
|--|-----------|
| 1. Use program arguments to specify a file name. | 10 points |
| 2. Use simple File I/O to read a file. | 10 points |
| 3. Create an abstract data type (ADT) to store information on a single movie. | 15 points |
| 4. Create an ADT that <i>abstracts</i> the use of an array (or list) of movies. | 15 points |
| 5. Implement a program that allows the user to search the list of movies as described below. | 20 points |
|
 | |
| ▶ Develop and use a proper design. | 15 points |
| ▶ Use proper documentation and formatting. | 15 points |

Description:

For this project, you will put together several techniques and concepts you have learned in CS 1323 (or from a similar background) and some new techniques to make an application that searches a collection of data on movies. This application will be called *Mdb* and will allow users to enter the names of movies and see basic data (such as release year) of each movie according to the data in the database.

One of the positive aspects of this project is that it will use an arbitrary amount of data. *Mdb* must be capable of handling data on hundreds or thousands of movies. To the surprise of no one, the best approach to this problem is to decompose the problem into separate classes that can be gradually built up. Note that much of the code you write for this program could be reused in more complex applications, such as the Internet Movie Database (IMDb), which, by the way, is the source of the data we will use in our projects this semester.

Operational Issues:

Data file and format:

Mdb will read the data file (a text file) as specified by a file name. The file name will be given as a program argument. (See below for information on how to read program arguments). Each line of the file contains information on a single movie. The line will contain the name of the movie, followed by a space, followed by one or more sets of parentheses containing information about the movie, followed by some amount of whitespace (meaning tabs or spaces), followed by the year the movie was released. For example:

```
Highlander (1986)           1986
```

This indicates that the name of the movie is “Highlander” and the 1986 at the end of the line indicates that the movie was released in 1986. The first 1986 (in parentheses) may appear to be redundant

information. However, in some cases this set of parentheses may contain obviously useful information. For example, in 1921 there were two movies released that both had the title “Black Beauty.” To distinguish between these in the database, one is marked as 1921/I and the other as 1921/II, as shown:

```
Black Beauty (1921/I)           1921
Black Beauty (1921/II)         1921
```

This tells us that the 1986 in parentheses for “Highlander” is not actually redundant information. Instead, the fact that the year at the end of the line matches the year in parentheses tells us that there was only one movie titled “Highlander” that was released in 1986¹. In some cases, additional information is included in additional sets of parentheses. For example:

```
The Star Wars Holiday Special (1978) (TV)      1978
```

Here the name of the movie is “The Star Wars Holiday Special,” it was released in 1978, and it was the only movie with that name released in 1978. In addition, (TV) indicates that this was a TV movie, rather than one released for viewing in movie theaters. The other parenthetical information that might be included for a movie is (V), which indicates that this movie went straight to video.

Note that there may be other sets of parentheses within the name of the movie itself! For example:

```
'Weird Al' Yankovic: (There's No) Going Home (1996) (TV)      1996
```

Here, the title is “Weird Al' Yankovic: (There's No) Going Home.” To deal with this complexity, you may want to parse the line from right to left, knowing that once you have found the set of parentheses that match the year of release (with possible Roman numerals for multiple movies with the same name), everything remaining to the left must be the title, regardless of what it is.

Finally, note that the release data may be indeterminate. In that case, it is given as a set of four question marks, rather than a numeric value. For example:

```
Starlight (????)                ????
```

This indicates that there is a movie entitled “Starlight” for which a release date has not yet been determined.

You will need to store the data on each movie as an object and the collection of all movies will be stored as a list of these objects. In addition, you may create and use objects of other types to give your system a logical design and the functionality required by the program specifications.

User interaction with MDb:

Once the list of movies has been read into MDb and stored, MDb will use a **JOptionPane** to display to the user a dialog box requesting the name of a movie.

When the user enters a movie name into the dialog, MDb will check to see if the name is associated with any movie in the database. If so, MDb will use another dialog to display to the user the data of the movie in the database with that name. In particular, MDb will show the name of the movie, the year it was released, and whether it was released in theaters, on TV, or on video. If the movie name is *not* associated with any movie in the list, MDb will use a dialog to inform the user of that fact.

After checking whether the name is associated with any movie in the list and displaying information to the user one way or the other, MDb will again use a dialog to request another movie name. It will continue in this loop until the user clicks on cancel, at which time MDb should gracefully exit.

1. There can be only one.

Implementation Issues:

There are two Java elements in this project that may be new to some students: reading from a file and program arguments. These Java features are summarized below.

Reading from a file:

We will discuss File I/O in more depth later in the class; this project is just designed to give you a brief introduction to the technique. Reading files is accomplished in Java using a collection of classes in the **java.io** package. To use the classes you must import the following package:

```
import java.io.IOException;
```

The first action is to open the file. This associates a variable with the name of the file sitting on the disk.

```
String fileName = "movie.txt";  
FileReader fr = new FileReader(fileName);
```

(Note that the lines given above will work if your data file is called “movie.txt.” However, you should *not* “hardcode” this file name into your source code. Instead, you should get the name of the file from a program argument when your program is run. You will, therefore, need to modify the code provided above to use the variable in which you have stored the program argument.)

Next the **FileReader** is wrapped with a **BufferedReader**. A **BufferedReader** is more efficient than a **FileReader** for working with groups of characters (as opposed to individual characters). Another advantage of using a **BufferedReader** is that there is a command to read an entire line of the file, instead of a single character at a time. This feature comes in particularly handy for this project.

```
BufferedReader br = new BufferedReader(fr);
```

The **BufferedReader** can now read in Strings.

```
String nextline;  
nextline = br.readLine();
```

Look at the Java API listing for **BufferedReader** and find out what `readLine()` returns when it reaches the end of the file (stream). Have your code process each line, putting the data into objects and variables while also looking for this special return value. When you are finished with the **BufferedReader**, the file should be closed. This informs the operating system that you’re finished using the file.

```
br.close();
```

Closing the **BufferedReader** also closes the **FileReader**.

Any method which performs I/O will have to throw or catch an **IOException**. If it is not caught, then it must be passed to the the calling method. The syntax is given below:

```
public void myMethod(int argument) throws IOException {  
    //method body here  
}
```

Program Arguments:

Sometimes it is handy to be able to give a program some input when it first starts executing. Program arguments can fulfill this need. Program arguments in Eclipse are equivalent to MS-DOS, Mac, or Unix command line arguments. Program arguments are handled in Java using a **String** array that is traditionally called `args` (the name is actually irrelevant). See the “Lab 2” slides (this year provided for Lab 1) for how to supply program arguments in Eclipse.

The program below will print out the program arguments.

```

public static void main(String[] args) {
    System.out.println(args.length + " program arguments:");
    for (int i=0; i< args.length; i++)
        System.out.println("args[" + i + "] = " + args[i]);
}

```

(Note that your program should not print the arguments but, instead, use the appropriate argument as the filename from which to read the data.)

Milestones:

A milestone is a “significant point in development.” Milestones serve as guides in the development of your project. Listed below are a set of milestones for this project along with a brief description of each.

Milestone 1. Use program arguments to specify a file name.

The name of the file that stores the list of data on movies will be passed to the program using program arguments as discussed above. Type in the sample program given in the section on program arguments and make sure that you understand how the program arguments you provide affect the `String[] args` parameter that is passed into the `main` method of the program. Then, write a `main` method for your program that reads in the name of the data file from the program arguments.

Milestone 2. Use simple File I/O to read a file.

Before you can allow the user to search the list of movies, you must first be able to read a text file. Examine the section above on reading from a file. A good start to the program is to be able to read in the name of a file from the program arguments, read each line from the file, one at a time, and print each line to the console using `System.out.println()`. Later, you will want to remove the code that prints out each line read in from the file, since the project requirements do not specify that the file is to be written out to the console as it is read.

Milestone 3. Create an abstract data type (ADT) to store data on a single movie.

You must create an ADT that holds the data for a single movie from the data file before you can store that data. Think about what data is associated with each movie and how to most efficiently store the data. Also, think about any methods that may help you to manage and compare the data by abstracting operations to be performed on individual entries in the list. Such methods may be used by other classes.

Milestone 4. Create an ADT that abstracts the use of a list of data about movies.

You are to store the object representing each movie into a list of objects. However, it is not necessary for the portions of the program that will carry out user actions to directly operate on this list as they would if you simply used an array of movie objects. Instead, you should create a class that abstracts and encapsulates this list and allows for the addition of new movies and also supports other required operations on it.

This ADT will represent the collection of information associated with the program. Think about the operations that this ADT needs to support and how it will use the ADT created for Milestone 3. At this point, you should be able to read in the input file and create an object for each movie in the file, and store that object in the list. Note that the data file used for grading may be larger (or smaller) than the data file provided for testing.

Milestone 5. Implement a program that allows the user to search the list of movies as described below.

This is where the entire program starts to take on its final form and come together. Here you will create the input and output dialogs and the menu system. Start by creating the input dialogs and the output dialogs. Tie together the input dialogs, the ADT from Milestone 4, and the output dialogs to make this

search functional and test its functionality.

Finally, you are ready to create the main loop of the program that will take input and invoke the correct methods to create appropriate output.

Remember that when the user clicks on “cancel,” the program must gracefully exit. This can be accomplished by using `System.exit(0)`.

How to Complete this Project:

Preliminary Design:

1 During the lab session and in the week following, you should 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 systems 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 and methods 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. 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 the classes until after the design is completed.

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 TAs 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 stub 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 for 5 points of extra credit. You could enable a wider range of searches to be used, such as searching based on release year or on ranges of release years, or searching based on release type (theater, TV, or video), or by using regulars expressions or wild cards in names. Alternatively, think of ways to decompose the list of movies into logical groupings. You could also revise the user interface. If you revise the user interface, you **must** still read the file name from the program arguments and the list of movies from the text file.

To receive the full five points, your extended feature must be novel 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 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 novel, and the write-up must cite any outside resources used.

Due Dates and Notes:

Due dates:

Your preliminary design (list of nouns, verbs, and adjectives; UML; stub code; detailed Javadoc documentation; and unit tests) is due on **Friday, February 5th**. Submit the project archive in D2L following the steps given in the submission instructions **by 11:00pm**.

The final version of the project is due on **Monday, February 15th**. Submit the project archive in D2L following the steps given in the submission instructions **by 11:00pm**.

Submissions:

Note that both the preliminary design and the final project are to be submitted electronically. Paper copies will not be submitted. The list of nouns, verbs, and adjectives and their corresponding classes/variables, methods, and interfaces should be in PDF format.

Academic honesty:

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.

ADTs

Do not be confused by the term “abstract data type” (ADT). An ADT is **not** the same as an abstract class, even though they both contain the word “abstract” in them.

A data type is simply a description of how bits in a computer are grouped and interpreted. Maybe one set of 32 bits is interpreted as a character, whereas another set of 32 bits is interpreted as an integer, and a set of 64 other bits is interpreted as an integer that can hold larger magnitudes, etc. With *concrete* data types, implementation details matter, such as the number of bits, whether the bits are ordered from least to most significant, etc. If you try to mix implementations, you’ll screw things up.

With an abstract data type, you hide the implementation details of the data type from the user, so that what matters is how one *interacts* with instances of the type, not how they are *implemented* internally. So, if you add two integers whose internal representations differ, you should still get a sensible result.

This means that if you create a class using object-oriented techniques (such as making variables private and only accessible through methods, etc.), then even a *concrete* class is an *abstract* data type.

The reason this description doesn't just tell you to create a class to store each movie's data is because you don't have to use just one class. You could use two classes, or three, or more. You could arrange them in an inheritance hierarchy (where one is a subclass of another). You could use composition or aggregation (the types of has-a links we have discussed). All of these classes could be concrete or some of them could be concrete and some could be abstract. You could also include interfaces, if you saw a good reason to do so. All of these alternatives would count as creating an ADT. However, you should also strive for simplicity; don't make an inheritance hierarchy or a bunch of classes or interfaces just because you can – try to match your design to the requirements.

If the term 'ADT' is still confusing you, think of the assignment as saying “Create something appropriate that the computer can use to store each movie's data.” That is what it means.

Hint:

Consider using an **ArrayList** in conjunction with Milestone 4.