CS-141 Introduction to Programming
Creating a Simple Database

A database is a program that consists of an organized collection of data together with a user interface that enables one to read, write, and access the data in multiple ways. One familiar example is The Internet Movie Database (IMDb). There are also many online music databases.

This document provides suggestions for how to go about writing a Java database program. The example, which will be used for illustration purposes, is a Movie database.

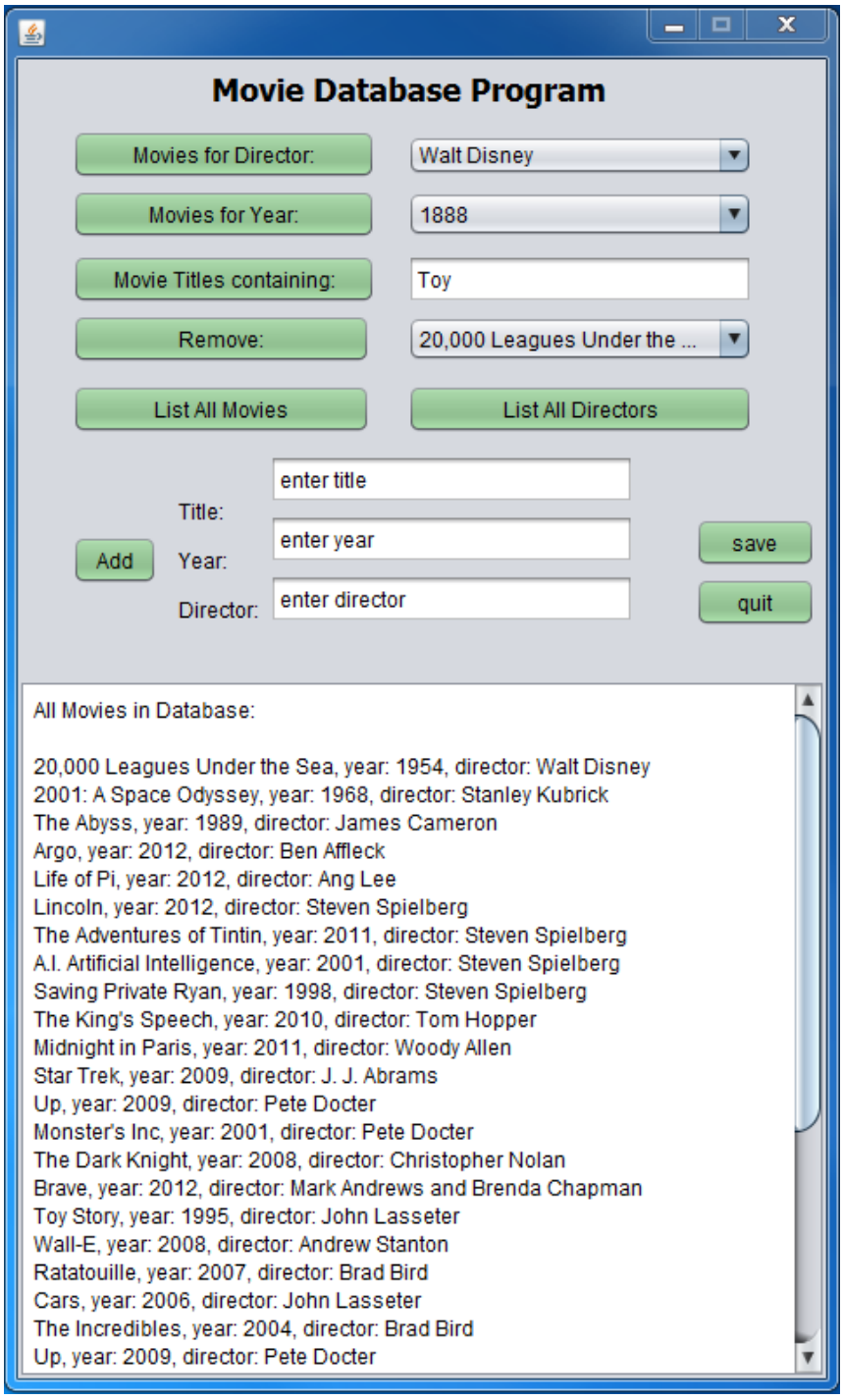
Suggested steps for creating your program:

- Get together with your partner and pick a subject. (E.g. movies)
- Identify the specific data you want to store, i.e. what data you need to put in your data file. (E.g. movie name, year, director, genre, …)
- Decide what kinds of queries or actions you want the user to be able to do.
- Design the look of the GUI interface based on your queries and actions (see example below).
- Design the UML Class Diagram which describes the program structure (this is a must!). Your UML diagram should include classes to store the data and methods to process your desired actions. Make sure the methods are placed in the appropriate classes.
- Decide the best order of implementation of classes and features. *Don't try do it all at once.* Start with the small classes first. Remember to apply the process of ***Stepwise Refinement***.
- Based on the order of implementation, divide the implementation between you and your partner so that each of you always have something to work on. It is not ok to say both of you will work on everything together; each person should do part of the program on their own.
- Decide with your partner how you will keep track of the most current version of the code. Note, programmers generally use version control software (e.g. git, RCS, etc) to manage things.
- As each person implements and tests their classes, the completed classes are added to the "most current version".
- As classes are added to the "most current version" of the project, the project should be run and tested as a whole.
- Once you feel you have a completed the program, recruit a friend who is unfamiliar with the program to test it. Fix parts that they find confusing or broken.

Below is an elaboration of a few of the above items.

Sample Graphical User Interface for a Movie Database

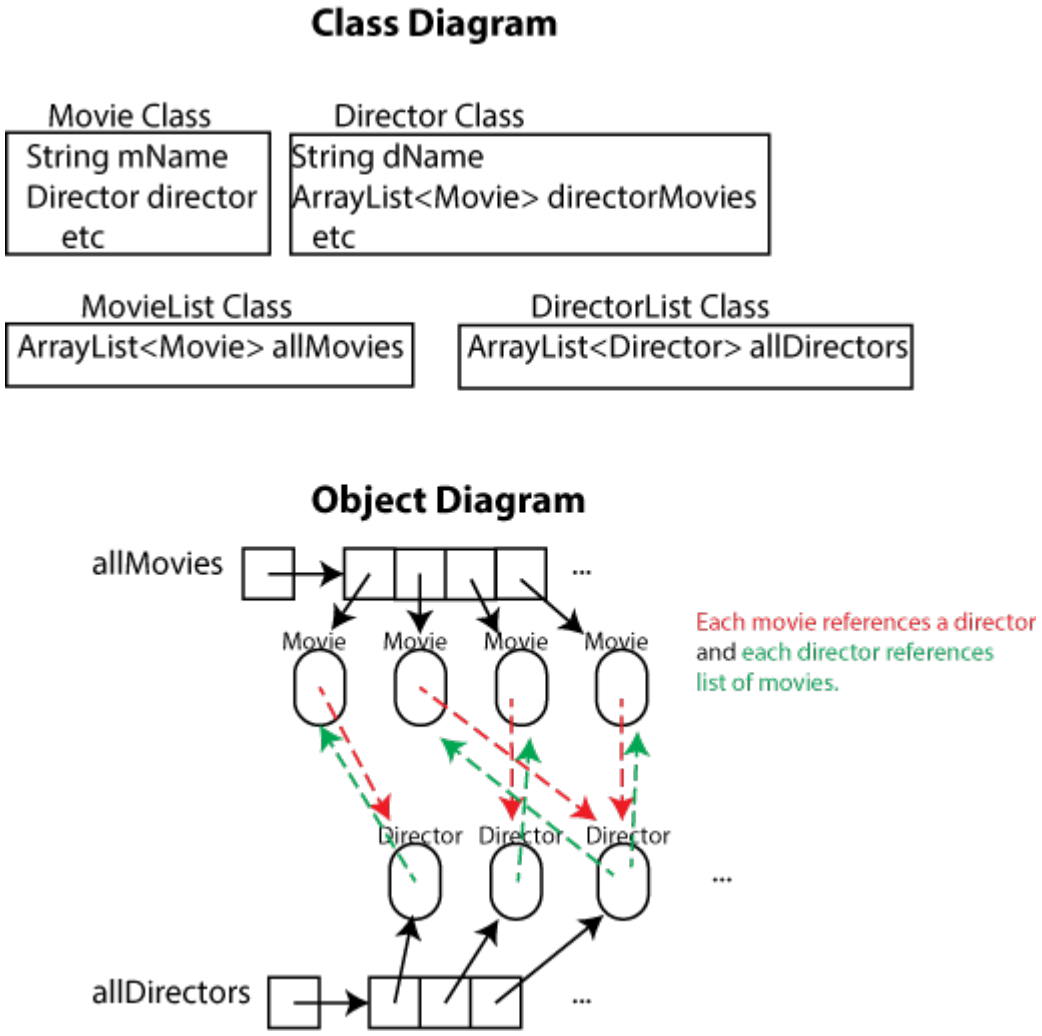
Carefully think about the user inteface. For a movie database, you might have something like what is shown on the following page. When the program is closed, an updated data file should be written out and should reflect items that the user entered or removed.



Class Diagram and Data Structure

As a program becomes more complex, it is hard to keep track of how all the parts fit together. Thus, it is really important to draw a class diagram so that the high level structure can be clearly seen. This should be done *before* you begin coding.

For a database, it is recommended that you use an ArrayList data structure to store the data. However*, beware of the problem of duplication.* Each movie must reference a Director object. But, each director could reference multiple movies. Ideally, one wants to store a single director object and have each movie, with that director, reference that single director object. We get a picture something like the following for the (partial) class and object diagrams:



Notice that the objects are never duplicated but each object may have multiple references pointing to it.

File Input/Output

Instructions:

1. For each item in your database (e.g. Movie), you need to consider what data needs to be provided and how you want it listed in the asci data file. You could place all the information for a given item on one line separated by some delimiter (e.g. space, comma, or semi-colon), assuming there is a delimiter which will never appear as part of the data. For example, movie names are often multiple words so it would not be possible to use a space as the delimiter. However, a semi-color might work. For example:

   Movie name1; year1; director1
   Movie name2; year2; director2
   etc

   If no good delimiter exists, then you could put each component on a separate line, e.g.

   Movie name1
   year1
   director1
   Movie name2
   Year2
   Director2
   Etc

   Include enough data so that you can carefully test your program. Note, you can always add more data later.

2. To read in the data from your data file, create a FileIO class (recall example code from class, listed on lectures page). Below is an example of how to handle reading/writing files which have multiple items on a line separated by a delimiter (in this case ":") using the split() method in the String class. As already mentioned, be careful not to store duplicates. Each time a movie with a given director is read, you need to check to see if that director already exists. If not, create a new Director object and add it to the list of all directors and also to the movie's director. If yes, retrieve the existing Director object from the allDirectors list and also add the movie to the director's list of movies.

   Below is code for reading and writing data for a State Database Program.

Sample Code for File Input/Output for a State Database Application

```
 /**
   * Read state & population data from a file. Note: Assumes the
   * delimiter is a ":" The data file would look something like:
   *      Oregon :  3970239
   *      California : 38802500
   *          etc
 */
 public void readArrayFile(String inFileName, ArrayList<State> stateDataArray) {
 try {
    File infile = new File(inFileName);
    Scanner in = new Scanner(infile);
    while (in.hasNext()) {
          String line = in.nextLine();    // Read a line in the file
          String[] tokens = line.split(":"); // Split line using ":"

          // Remove extra whitespace from name
          String name = tokens[0].trim();

          // Convert the year (stored as String) to an integer:
          int population = Integer.parseInt(tokens[1].trim());

          // Create a State object with data
          State state = new State(name, population);
          stateDataArray.add(state);  // add to ArrayList
    }
    in.close();
 } catch (FileNotFoundException e) {
    System.out.println("Input File " + inFileName + " is not found. ");
    System.exit(0);
 }
}

 /** Write database out to File */
 // This works because the database's toString method prints the data as it
 // appears in the data file.  If this is not the case, you need to provide
 //   formatting code.
 public void writeArrayFile(String outFileName, StateDataBase stateDataArray) {
    try {
          System.out.println("Saving modified data to " + outFileName);
          PrintWriter out = new PrintWriter(outFileName);
          out.print(stateDataArray);  // works because of how toString is written
          out.close();
    } catch (FileNotFoundException e) {
          System.out.println("Error writing to " + outFileName);
          System.exit(0);
    }
 }
```