Overview/

In this assignment, I was tasked with creating a relational database to model Actors, Movies, Directors, and Awards. This involved creating a schema as well as data to be inserted into this database. Furthermore, I had to create multiple Java programs to perform these operations: creating the tables, inserting the data, and running queries based on arguments entered by the user. This made use of the JDBC (Java Database Connectivity) API to make connections to the database and execute queries. As well as this, I had to create an entity relationship diagram to model by database including names of tables and their attributes. I used skills like choosing appropriate classes and methods in the JDBC API, testing and debugging, developing a robust solution to the problem, and developing my skills at ER modelling.

Design and Implementation/

Within my solution folder, I have divided parts of my program into their respective directories:

- src: for my java programs
- tables: for my DDL schema for creating tables
- libs: for my jar files that get added to the classpath
- databases: for my database files
- data: for the csv files containing data to add to the database
- testing: for my junit testing files
- docs: for my built javadoc documentation

Additionally in my project folder I have my ER diagram as a PNG file as well as a README file containing information on how to run my program.

Throughout my program in my Java files, I have created informative javadoc comments that improve readability and informs the reader on what the class and method does.

ER diagram/

My ER diagram is comprised of 7 tables. The Actor and Director tables feature similar attributes but with different primary key names. The Movie table has the most attributes and is linked to Director through the DirectorID foreign key. However, to link this table to actor initially required a many to many relationship, I resolved this by adding in a weak table of ‘Cast’ which is comprised of a compound key of two foreign keys and an additional character field. This removes the many to many relationship. For awards, I initially had one Awards table, but this would require multiple foreign keys which was not possible, so I split it up into MovieAward, ActorAward, DirectorAward.

InitialiseDB/

This program creates and initialises a database in SQLite. It deletes any existing database file and recreates it using the schema files provided in tables/. It then checks to make sure the tables are properly created.

The class has 2 variables containing the relative file path to the database and to the schema, ensuring that this program will work no matter where the project folder is on the computer.

The main method starts by attempting to create a file object with the database file path and then deleting it if it managed to create it.

A database connection is then established, creating an empty database. The next line makes sure that foreign key constraints are enabled. This is something I only realised was necessary quite far into development and only through extensive research.

A method is then called to create the tables in the database, which loops through all the files in the tables/ directory and executes a prepared statement with the entire of each file.

After this, another method checks that the tables were created by querying all the table names and matching them against a list of tables stored in an array.

All methods in this class throw an SQLException that is handled properly in try-catch blocks, ensuring that any errors are printed out to the user in standard output.

PopulateDB/

This program reads in data from a list of CSV files and inserts the contents into the SQLite database that was initialised in the program above. The program maps each csv file to an already created database table and ensures correct data entry.

The main method starts by creating an array of files in the data/ directory. It then loops through each file and reads in the data line by line, where a method to insert the data to the database is called for every line.

An integer variable is used to track what number file the program is currently reading, which is then mapped to a string array of table names, so it can concatenate the table name in the query.

The insert row method first maps this integer variable to the table name and then retrieves the column names for this table. This is so the first column (the primary key) can be skipped as it is auto incremented as defined in the schema. This is the case for all tables apart from the ‘Cast’ table which has data for all columns to be inserted.

The program then uses string builders for column names and placeholder values (question marks separated by commas) to build the query.

From there, the data is inserted into the database using a prepared statement.

Using one single method to insert data where it generates the column names means that robustness is improved and this solution will work when more tables are added, instead of having a method to insert data for each table.

All methods in this class throw an SQLException that is handled properly in try-catch blocks, ensuring that any errors are printed out to the user in standard output.

QueryDB/

This program connects to the database and runs searching queries based on a user-entered arguments. The program accepts a query number (1-6), and any further additional arguments.

The main method processes the command line arguments by first checking if there are any arguments at all, throwing an error if not. A switch case statement is then used to handle each integer from 1-6.

Some integers (2,3,4), require further parameters, so within those cases the program must check that there are the correct number of parameters or else throw an error.

There is a generic args error message:

- "Usage: {integer 1-6} {optional params each in single quotes}"

As well as a method of the same name that takes in 2 parameters, the query number and the customised message. For query 2, it looks like this:

- Usage: {2} {'movie title'}

For queries 5 and 6, it was my choice of what they were. Information about these queries can be found in the README file.

If the number of arguments is correct, the program will pass the query into the executeQuery method, which creates a prepared statement with the query and then executes it. The ResultSet is then passed into a method that prints it to the terminal.

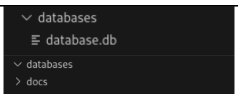
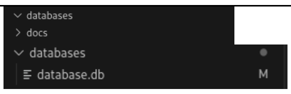
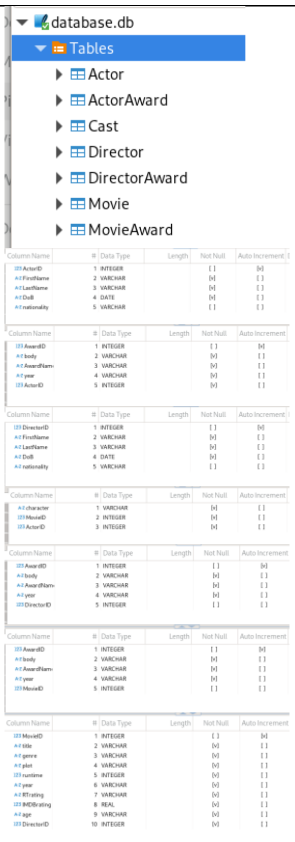
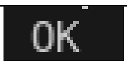
The print method takes the ResultSet and pretty prints it to standard output, showing the row number and name of each column before the value.

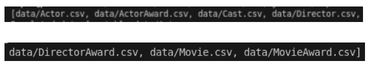
All methods in this class throw an SQLException that is handled properly in try-catch blocks, ensuring that any errors are printed out to the user in standard output.

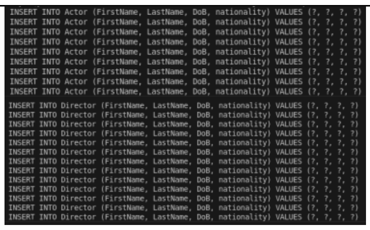
Testing/

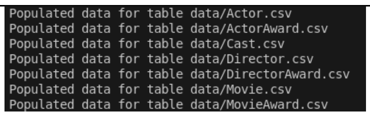
My program features a small JUnit test suite that tests each of my 3 programs with sample test data that is smaller than the main data. Instructions on how to run this can be found in the README file.

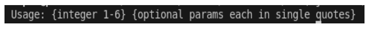
Some further testing and examples can be found below.

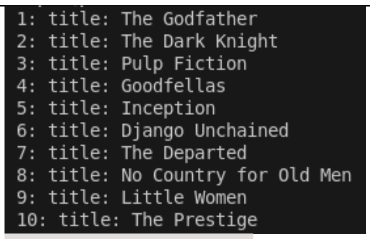
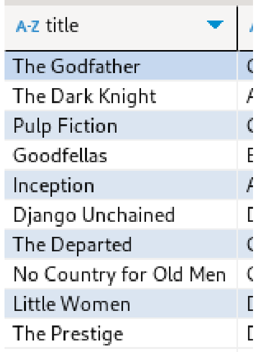
What is being tested	Conditions	Expected Output	Actual Output	Pass/Fail
InitialiseDB: Existing database table gets deleted	Existing DB file Further code is commented out	Database file gets deleted	 databases database.db databases docs Database file is deleted	P
InitialiseDB: New database file is created	No existing DB file Further code is commented out	A new file called 'database.db' is created	 databases docs databases database.db M Database file is created	P
InitialiseDB: New database has the correct tables	Using DBeaver, inspecting the tables to make sure they match the schema	Tables should be in Dbeaver	 database.db Tables Actor ActorAward Cast Director DirectorAward Movie MovieAward Column Name # Data Type Length Not Null Auto Increment ActorID 1 INTEGER [ ] [v] FirstName 2 VARCHAR [v] [ ] LastName 3 VARCHAR [v] [ ] DoB 4 DATE [v] [ ] nationality 5 VARCHAR [ ] [ ] Column Name # Data Type Length Not Null Auto Increment AwardID 1 INTEGER [ ] [v] body 2 VARCHAR [v] [ ] AwardNam 3 VARCHAR [v] [ ] year 4 VARCHAR [v] [ ] ActorID 5 INTEGER [v] [ ] Column Name # Data Type Length Not Null Auto Increment DirectorID 1 INTEGER [ ] [v] FirstName 2 VARCHAR [v] [ ] LastName 3 VARCHAR [v] [ ] DoB 4 DATE [v] [ ] nationality 5 VARCHAR [ ] [ ] Column Name # Data Type Length Not Null Auto Increment character 1 VARCHAR [v] [ ] MovieID 2 INTEGER [v] [ ] ActorID 3 INTEGER [v] [ ] Column Name # Data Type Length Not Null Auto Increment AwardID 1 INTEGER [ ] [v] body 2 VARCHAR [v] [ ] AwardNam 3 VARCHAR [v] [ ] year 4 VARCHAR [v] [ ] DirectorID 5 INTEGER [ ] [ ] Column Name # Data Type Length Not Null Auto Increment AwardID 1 INTEGER [ ] [v] body 2 VARCHAR [v] [ ] AwardNam 3 VARCHAR [v] [ ] year 4 VARCHAR [v] [ ] MovieID 5 INTEGER [ ] [ ] Column Name # Data Type Length Not Null Auto Increment MovieID 1 INTEGER [ ] [v] title 2 VARCHAR [v] [ ] genre 3 VARCHAR [v] [ ] plot 4 VARCHAR [v] [ ] runtime 5 INTEGER [v] [ ] year 6 VARCHAR [v] [ ] RTrating 7 VARCHAR [v] [ ] IMDBrating 8 REAL [v] [ ] age 9 VARCHAR [v] [ ] DirectorID 10 INTEGER [v] [ ]	P
Running InitialiseDB program	The rest of the program	Should print OK in the terminal	 OK	P

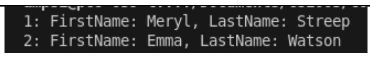
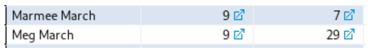
PopulateDB: File array contains the names of all the files	Print out dataFiles[]
	
	System.out.println(Arrays.toString(dataFiles));
		Should list all the names of the 7 files	
			`[data/Actor.csv, data/ActorAward.csv, data/Cast.csv, data/Director.csv,`
			`data/DirectorAward.csv, data/Movie.csv, data/MovieAward.csv]`
				P

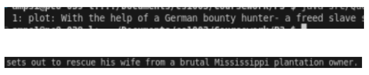
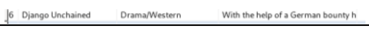
PopulateDB: Insert queries are being created properly	Print out the query before executing
	
	System.out.println(query);
		For each record, it should print the query	
			`INSERT INTO Actor (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Actor (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Actor (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Actor (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Actor (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Actor (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Actor (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Actor (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Actor (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
			`INSERT INTO Director (FirstName, LastName, DoB, nationality) VALUES (?, ?, ?, ?)`
				P

Running PopulateDB program	The rest of the program	Should print messages for each table	
			`Populated data for table data/Actor.csv`
			`Populated data for table data/ActorAward.csv`
			`Populated data for table data/Cast.csv`
			`Populated data for table data/Director.csv`
			`Populated data for table data/DirectorAward.csv`
			`Populated data for table data/Movie.csv`
			`Populated data for table data/MovieAward.csv`
				P

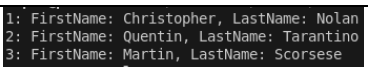
QueryDB: No arguments given	Program is ran with no command line arguments supplied	Should print the default noargs message	 `Usage: {integer 1-6} {optional params each in single quotes}`	P

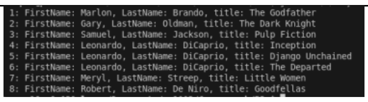
QueryDB: Run Query 1	Program is ran with args: {1}	Should print query 1, listing all movies in the database, and it should match DBeaver result	
			`1: title: The Godfather`
			`2: title: The Dark Knight`
			`3: title: Pulp Fiction`
			`4: title: Goodfellas`
			`5: title: Inception`
			`6: title: Django Unchained`
			`7: title: The Departed`
			`8: title: No Country for Old Men`
			`9: title: Little Women`
			`10: title: The Prestige`
			
			title
			The Godfather
			The Dark Knight
			Pulp Fiction
			Goodfellas
			Inception
			Django Unchained
			The Departed
			No Country for Old Men
			Little Women
			The Prestige
				P

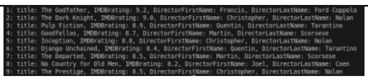
QueryDB: Run Query 2	Program is ran with args: {2} {Little} {Women}	Should print the actors in Little Women	
			`1: FirstName: Meryl, LastName: Streep`
			`2: FirstName: Emma, LastName: Watson`
			Little Women: MovieID 9
			Meryl Streep: ActorID 7
			Emma Watson: ActorID 29
			
			Marmee March 9 7
			Meg March 9 29
				P

QueryDB: Run Query 3	Program is ran with args: {3}	Should print the plot of	
			`1: plot: With the help of a German bounty hunter- a freed slave s`
			`sets out to rescue his wife from a brutal Mississippi plantation owner.`
			
			6 Django Unchained Drama/Western With the help of a German bounty h
				P

	{‘Leonardo’}
	{‘DiCaprio’}
	{‘Quentin’}
	{‘Tarantino’}
		Django Unchained		

QueryDB: Run Query 4	Program is ran with args: {4}
	{‘Leonardo’}
	{‘DiCaprio’}
		Should list the director’s names of movies that he is in that they directed	
			`1: FirstName: Christopher, LastName: Nolan`
			`2: FirstName: Quentin, LastName: Tarantino`
			`3: FirstName: Martin, LastName: Scorsese`
			Leonardo DiCaprio
			ActorID is 10
			He is in movies MovieID 5,6,7
			
			MovieID 5 is directed by DirectorID 3
			DirectorID 3 is Christoper Nolan
			
			MovieID 6 is directed by DirectorID 4
			DirectorID 4 is Quentin Tarantino
			
			MovieID 7 is directed by DirectorID 5
			DirectorID 5 is Martin Scorsese
				P

QueryDB: Run Query 5	Program is ran with args {5}	Should list actors who have won an award and the movies they are in	
			`1: FirstName: Marlon, LastName: Brando, title: The Godfather`
			`2: FirstName: Gary, LastName: Oldman, title: The Dark Knight`
			`3: FirstName: Samuel, LastName: Jackson, title: Pulp Fiction`
			`4: FirstName: Leonardo, LastName: DiCaprio, title: Inception`
			`5: FirstName: Leonardo, LastName: DiCaprio, title: Django Unchained`
			`6: FirstName: Leonardo, LastName: DiCaprio, title: The Departed`
			`7: FirstName: Meryl, LastName: Streep, title: Little Women`
			`8: FirstName: Robert, LastName: De Niro, title: Goodfellas`
			SELECT DISTINCT used so no duplicate records
				P

QueryDB: Run Query 6	Program is ran with args {6}	Should list all films with IMDB rating >= 8.0 and the director's name	
			`1: title: The Godfather, IMDBrating: 9.2, DirectorFirstName: Francis, DirectorLastName: Ford Coppola`
			`2: title: The Dark Knight, IMDBrating: 9.0, DirectorFirstName: Christopher, DirectorLastName: Nolan`
			`3: title: Pulp Fiction, IMDBrating: 8.9, DirectorFirstName: Quentin, DirectorLastName: Tarantino`
			`4: title: Goodfellas, IMDBrating: 8.7, DirectorFirstName: Martin, DirectorLastName: Scorsese`
			`5: title: Inception, IMDBrating: 8.8, DirectorFirstName: Christopher, DirectorLastName: Nolan`
			`6: title: Django Unchained, IMDBrating: 8.4, DirectorFirstName: Quentin, DirectorLastName: Tarantino`
			`7: title: The Departed, IMDBrating: 8.5, DirectorFirstName: Martin, DirectorLastName: Scorsese`
			`8: title: No Country for Old Men, IMDBrating: 8.2, DirectorFirstName: Joel, DirectorLastName: Coen`
			`9: title: The Prestige, IMDBrating: 8.5, DirectorFirstName: Christopher, DirectorLastName: Nolan`
				P

Evaluation/

I believe I have developed a robust, efficient, and modular solution to the problem given in the specification. I believe I have achieved all the requirements to a very satisfactory degree and don't believe there are any issues with my solution.

I have created an entity relationship diagram that features tables and attributes as well as their relationships.

I have created a DDL schema that reflects the ER diagram from Requirement 1. This schema when ran properly creates tables in the database.

I have integrated Java with my database by running this DDL code via JDBC to create tables. My program checks for existing files, creates the new file, creates the tables, and makes sure the tables were created and prints “OK” afterwards.

I have created a Java program to populate the database using handmade CSV data which is stored in a directory in my project folder. I have used relative paths throughout all my code.

I have created a Java program to run queries against my database as outlined in the requirements, as well as two of my own that are explained in my README file. These queries are run via command line arguments that are validated, and the user is shown an error message if the arguments are not correct.

All my queries use prepared statements and none of them directly access files in the file system.

I have added Javadoc comments to all classes and methods and have built them into a viewable docs site that is found in docs/index.html.

I have created a suite of JUnit tests found in testing/, as well as my own testing table that runs sophisticated tests to ensure maximum functionality of my program.

Conclusion/

I thoroughly enjoyed this project, and it gave me a chance to refresh myself with skills and knowledge learned from school, as I did a lot of SQL there.

I found the SQL part of this not at all challenging but using JDBC was the new skill I learned, as well as Javadoc comments which I had never made prior to this assignment. If I had more time, I would implement more Junit tests.