**Assignment 7**

**Due, Tuesday, August 1, 2016 for maximum 100**

**Wednesday, August 2, 2016 for maximum 90**

**Thursday, August 3, 2016 for maximum 80**

**Friday, August 4, 2016 for maximum 70**

**Deliverables**

To complete this assignment you must submit your **compressed Netbeans project** to Webcourses.

**Tasks and Rubric**

|  |  |
| --- | --- |
| Activity | |
| DatabaseJavaFx project | Create a new Java Application project named DatabaseJavaFx  allowing Netbeans IDE to create the main class called DatabaseJavaFx  Be sure to select the following on the Choose Project UI:   1. JavaFX 2. JavaFX Application |
| databasejavafx package | Automatically created by Netbeans IDE |
| DatabaseJavaFx.java | Automatically created by Netbeans IDE in package databasejavafx package  This class shall do the following: |
|  | Member variables:   1. Instantiate member variable of type Logger, set equal to the static method call Logger.getLogger(DatabaseJavaFx.class.getName()) 2. Instantiate member variable of type ObservableList<FilmDAO> set equal to static method call FXCollections.observableArrayList() |
|  | Methods:   1. The main method should call static method launch() passing args as an argument 2. Override the method start(), receiving parameter of type Stage; this method will do the following:    1. Instantiate an instance of class TableView calling the no argument constructor    2. On the TableView object, call method .setEditable() to false so the data cannot be edited in the display    3. Instantiate an instance of class Label passing as an argument to the constructor the explicit string “Films”    4. Create four instances of class TableColumn, one for each column of the database table we want to display by calling the constructor and passing as an argument the text we want displayed above the column on the UI       1. Title       2. Description       3. Rental Rate       4. Rating    5. For each instance of class TableColumn instantiated above do the following:       1. Call method .setMinWidth(), passing as an argument the number of pixels necessary to display all the data       2. Bind the data in the DAO to the UI by calling method setCellValueFactory() passing as an argument new PropertyValueFactory<FilmDAO, String>("filmName")) where:          1. FilmDAO holds the data to display          2. The second argument is the data type of the data to display          3. As an argument to the constructor PropertyVlaueFactory is the member variable name of the data to display.    6. Call method .getColumns().addAll(title, description, rate, rating) on the instance of the TableView object to add the columns to the UI… the arguments passed are indicative of the variable names used to create each TableColumn instance    7. Instantiate an instance of class Button calling the constructor with one argument passing explicit text "Fetch films from database"    8. Assign an EventHandler by calling method .setOnAction on the button instantiate passing as an argument a new EventHandler<ActionEvent >()       1. Define the EventHandler as follows          1. Override method handle(ActionEvent event) so that is calls method fetchData() passing as an argument the instance of class TableView    9. Instantiate an instance of class Scene calling the constructor with argument new Group() as the layout manager.    10. Instantiate an instance of class VBox calling the no argument constructore        1. Call method .setPrefHeight() on the instance of VBox to the desired pixels        2. Call method .setStyle() on the instance of VBox to create the look and feel you desire (GTS for your options)        3. Call method .getChildren().addAll(label, tableView), to add the FX UI controls to the parent UI class    11. Add to the scene the VBox components by calling ((Group) scene.getRoot()).getChildren().addAll(vbox, fetchData);    12. Call method .setTitle() on the instance of class Stage to put a title on the UI, pass as an argument explicit text "Films for Rent"    13. Call method .setScene() on the instance of class Stage passing as an argument the instance of class Scene    14. Call method .show() on the instance of class Stage to display the UI 3. Create method fetchData    1. return type void    2. parameter list of type TableView    3. Create a try block, in the parenthesis       1. Instantiate an instance of class Connection set equal to method call getConnection()          1. If successful, call method.setItems() on the instance of class TableView, passing as an argument method call fetchFilms() which passes as an argument the Connection variable    4. Create a catch block       1. Catches exceptions SQLException and ClassNotFoundException       2. If an exception is caught log it using the Logger instance 4. Create method getConnection    1. return type Connection    2. parameter list empty    3. throws exceptions ClassNotFoundException and SQLException    4. Download properties.xml from assignment folder on Webcourses.    5. Create an instance of class XmlParser, passing “inputOutput/properties.xml” as an argument to the constructor    6. Declare a variable of class ConnectionData; instantiate it by setting it equal to the return variable from class XmlParser method getConnectionData()    7. Create an instance of class PostgreSQLConnect by passing the variable of class ConnectionData as an argument to the constructor    8. Call method getConnection() using the reference variable created above 5. Create method fetchFilms    1. return type ObservableList<FilmDAO>    2. parameter list of class Connection    3. throws SQLException    4. Instantiate an instance of class ObservableList<FilmDAO> set equal to static method call FXCollections.observableArrayList()    5. Create an instance of class String set equal to the SQL query "select title, rental\_rate, rating, description " +   "from film " +  "order by title;"   * 1. Create an instance of class Statement set equal to method call .createStatement() on the instance of class Connection   2. Create an instance of class ResultSet set equal to method call method call executeQuery() on the instance of class Statement passing as an argument the instance of class String with the SQL query   3. Create a while loop that evaluates if the instance of class ResultSet has more entries by calling method .next(); if true, do the following      1. Create an instance of class FilmDAO calling the no argument constructor      2. Call the setter method for each member variable of the class         1. Pass as an argument the .getString() or .getDouble() method on the instance of class ResultSet based on the data type of the data stored in the database   4. Add the instance of class FilmDAO to the ObservableList<FilmDAO> instance   5. return the instance of the ObservableList<FilmDAO> instance |
| dataModel package | 1. Create package dataModel |
| FilmDAO.java | 1. Create class FilmDAO in package dataModel |
|  | 1. Include member variables    1. Of type class String that stores the film’s title    2. Of type class String that stores the film’s rating    3. Of type class String that stores the film’s description    4. Of type class Double that stores the film’s rental price |
|  | 1. Include methods:    1. getters/setters for the member variables |
|  | 1. Include constructors:    1. A no-argument constructor    2. A constructor that receives the following parameters       1. String for the film name       2. String for the film rating       3. String for the film description       4. Double for the film rental price       5. The constructor must set the member variable values based on the parameters received |
| inputOutput package | Create package inputOutput |
| PostgreSQLConnect.java | 1. Include member variable Connection connect 2. Write a constructor to receive a parameter of class ConnectionData 3. Inside a try block    1. Call static method Class.forName() passing argument getType() method call on the instance of ConnectionData    2. Set variable **connect** equal to static method call DriverManager.getConnection() passing the following arguments       1. Method call toString() on class instance ConnectionData       2. Method call getLogin() on class instance ConnectionData       3. Method call getPassword() on class instance ConnectionData 4. Inside catch block    1. catch exception type Exception    2. output to the user that the database connection was not successful 5. If the code gets past the try/catch without throwing an exception then the database connection was successful, output to the user that the connection was successful |
| ConnectionData.java | Create class ConnectionData in package inputOutput; it should do the following:   1. Member variable    1. String type    2. String url    3. String ipaddress    4. String port    5. String database    6. String login    7. String password 2. Getters/setters for member variables 3. Method toString() should concatenate member variables url, ipaddress, port, and database so that it is in the format "jdbc:postgresql://localhost:5432/dvdrental" |
| XmlParser.java | Create class XmlParser in package inputOutput; it should do the following:   1. Member variable    1. ConnectionData connectionData    2. Document document 2. Getter for member variable connectionData 3. Constructor should do the following:    1. Defined one parameter of type String storing the name of the XML file to parse    2. Call method parseXmlFile() passing the parameter as an argument 4. Method parseXmlFile() should do the following:    1. return type void    2. Define one parameter of type String storing the name of the XML file to parse    3. Create an instance of class DocumentBuilderFactory    4. Create an instance of class DocumentBuilder    5. Instantiate member variable document to the parse method of class DocumentBuilder passing argument ClassLoader.getSystemResourceAsStream() passing the parameter for method parseXmlFile as an argument    6. Create an instance of class NodeList, instantiate it with method getDocumentElement().getChildNodes() from member variable document    7. Loop through the length of the NodeList instance       1. Create an instance of class Node, instantiate it to method item() of instance NodeList       2. Get each element from XML element <driver>       3. Instantiate member variable connectionData setting each of its member variables relative to the elements in the properties.xml file |
| properties.xml | Put file in the inputOutput package |
| Javadoc | 1. All files and all methods MUST include Javadoc comments in order to produce documentation on how the application runs 2. The project must have the Javadoc generated and included in the submission |
| Logger | 1. All files and all methods MUST include Logger statements so the user knows what is going on at any given point during application runtime. |
| DatabaseJavaFx application |  |
| Test Case 1 | Test Case 1 passes |
| Test Case 2 | Test Case 2 passes |
| Test Case 3 | Test Case 3 passes |
| Test Case 4 | Test Case 4 passes |
| Test Case 5 | Test Case 5 passes |
|  | Source compiles with no errors |
|  | Source runs with no errors |
|  | Source includes comments |
| Total |  |

**Perform the following test cases**

|  |  |  |
| --- | --- | --- |
| Test Cases | | |
|  | **Action** | **Expected outcome** |
| Test Case 1 | **Display JavaFX UI no data** | JavaFX UI displays without data, see Figure 1 |
| Test Case 2 | **Display JavaFX UI with data from database** | UI updated with data from the PostgreSQL database, see Figure 2 |
| Test Case 3 | **Project view same as image** | Ensure project view looks like the image provided below, see Figure 3 |
| Test Case 4 | **Javadoc generated** | In the Files view of the project, under the dist folder the Javadoc should be accessible, see Figure 4 |
| Text Case 5 | **Logger output** | In the output UI of Netbeans there should be significant output from the Logger class as the application runs, see Figure 5 |

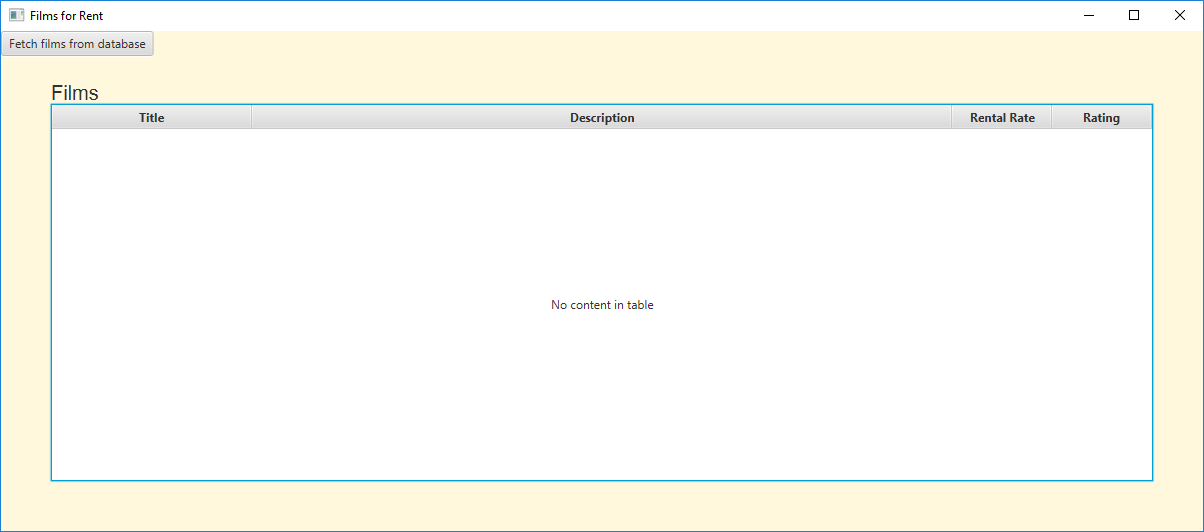


Figure 1 Initial display



Figure 2 After clicking Fetch button

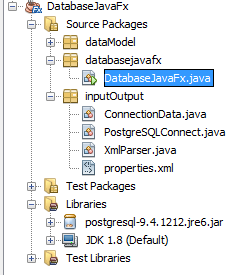


Figure 3 Project view

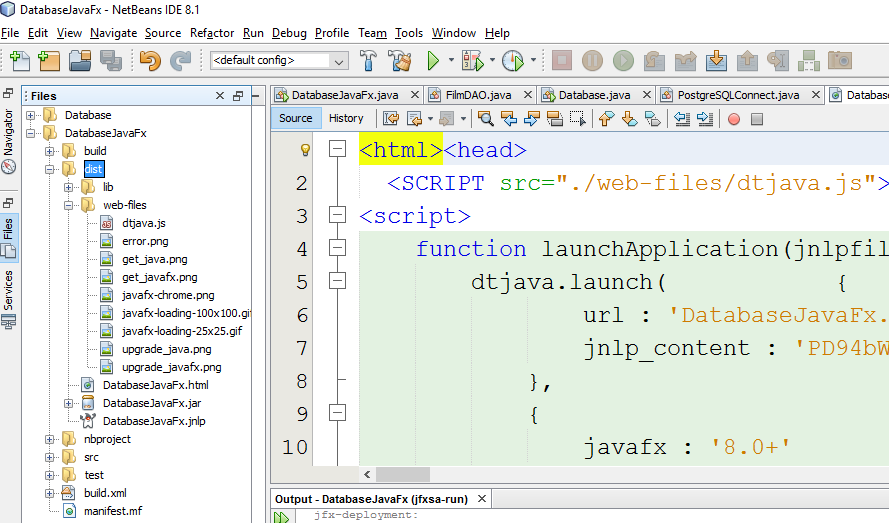


Figure 4 Javadoc

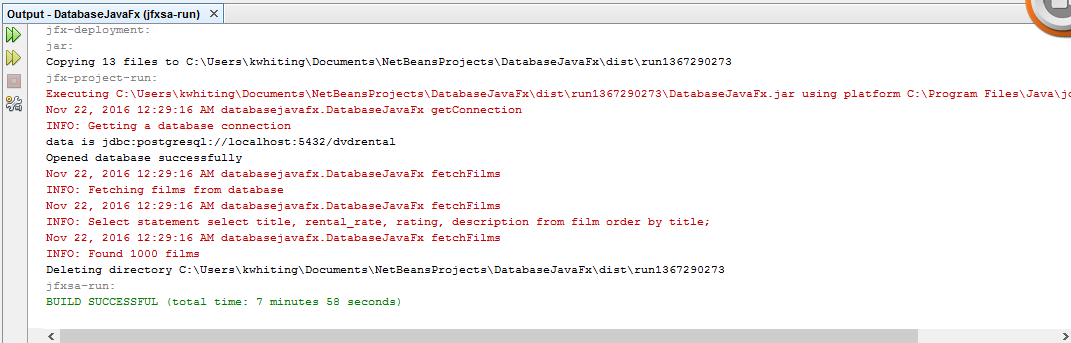


Figure 5 Logger output