24.7 Querying the books Database

Next, we present a `DisplayQueryResults` app that allows you to enter a SQL query and see its results. The GUI uses a combination of JavaFX and Swing controls. We display the query results in a Swing `JTable` (package `javax.swing`), which can be populated dynamically from a `ResultSet` via a `TableModel` (package `javax.swing.table`). A `TableModel` provides methods that a `JTable` can call to access a `ResultSet`'s data. Though JavaFX's `TableView` control provides data-binding capabilities (like those in Chapter 13), the combination of `JTable` and `TableModel` is more powerful for displaying `ResultSet` data.

24.7.1 `ResultSetTableModel` Class

Class `ResultSetTableModel` (Fig. 24.25) is a `TableModel` that performs the connection to the database and maintains the `ResultSet`. The class extends class `AbstractTableModel` (package

`javax.swing.table`), which implements interface `TableModel`. `ResultSetTableModel` overrides `TableModel` methods `getColumnClass`, `getColumnCount`, `getColumnName`, `getRowCount` and `getValueAt`, based on the current `ResultSet`. The default implementations of `TableModel` methods `isCellEditable` and `setValueAt` (provided by `AbstractTableModel`) are not overridden, because this example does not support editing the `JTable` cells. The default implementations of `TableModel` methods `addTableModel-Listener` and `removeTableModelListener` (provided by `AbstractTableModel`) are not overridden, because the `AbstractTableModel` implementations of these methods properly add and remove listeners for the events that occur when a `TableModel` changes.

```
   1   // Fig. 24.25: ResultSetTableModel.java
 2   // A TableModel that supplies ResultSet data to
       3   import java.sql.Connection;
       4   import java.sql.Statement;
     5   import java.sql.DriverManager;
       6   import java.sql.ResultSet;
      7   import java.sql.ResultSetMetaData;
     8   import java.sql.SQLException;
 9   import javax.swing.table.AbstractTableModel;
                    10
11   // ResultSet rows and columns are counted from 1
12   // rows and columns are counted from 0. When pro
13   // ResultSet rows or columns for use in a JTable
14   // necessary to add 1 to the row or column numbe
15   // the appropriate ResultSet column (i.e., JTabl
16   // ResultSet column 1 and JTable row 0 is Result
```

```
17   public class ResultSetTableModel extends Abstrac
18      private final Connection connection;
19      private final Statement statement;
20      private ResultSet resultSet;
21      private ResultSetMetaData metaData;
22      private int numberOfRows;
23
24      // keep track of database connection status
25      private boolean connectedToDatabase = false;
26
27      // constructor initializes resultSet and obta
28      // determines number of rows
29      public ResultSetTableModel(String url, String
30         String password, String query) throws SQLE
31         // connect to database
32         connection = DriverManager.getConnection(u
33
34         // create Statement to query database
35         statement = connection.createStatement(
36            ResultSet.TYPE_SCROLL_INSENSITIVE, Resu
37
38         // update database connection status
39         connectedToDatabase = true;
40
41         // set query and execute it
42         setQuery(query);
43      }
44
45      // get class that represents column type
46      public Class getColumnClass(int column) throw
47         // ensure database connection is available
48         if (!connectedToDatabase) {
49            throw new IllegalStateException("Not Co
50         }
51
52         // determine Java class of column
53         try {
54            String className = metaData.getColumnCl
55
56            // return Class object that represents
```

```
            return Class.forName(className);
        }
        catch (Exception exception) {
          exception.printStackTrace();
        }

    return Object.class; // if problems occur
    }

  // get number of columns in ResultSet
  public int getColumnCount() throws IllegalSta
    // ensure database connection is available
    if (!connectedToDatabase) {
      throw new IllegalStateException("Not Co
    }

    // determine number of columns
    try {
      return metaData.getColumnCount();
    }
    catch (SQLException sqlException) {
      sqlException.printStackTrace();
    }

    return 0; // if problems occur above, retu
    }

  // get name of a particular column in ResultS
  public String getColumnName(int column) throw
    // ensure database connection is available
    if (!connectedToDatabase) {
      throw new IllegalStateException("Not Co
    }

    // determine column name
    try {
      return metaData.getColumnName(column +
    }
    catch (SQLException sqlException) {
      sqlException.printStackTrace();
```

```
                97         }
                    98
99         return ""; // if problems, return empty st
                100      }
                    101
  102      // return number of rows in ResultSet
103      public int getRowCount() throws IllegalState
104         // ensure database connection is availabl
     105         if (!connectedToDatabase) {
106            throw new IllegalStateException("Not C
                107         }
                    108
         109         return numberOfRows;
                110      }
                    111
112      // obtain value in particular row and column
113      public Object getValueAt(int row, int column
    114         throws IllegalStateException {
                    115
116         // ensure database connection is availabl
     117         if (!connectedToDatabase) {
118            throw new IllegalStateException("Not C
                119         }
                    120
121         // obtain a value at specified ResultSet
                122         try {
   123            resultSet.absolute(row + 1);
124            return resultSet.getObject(column + 1)
                125         }
 126         catch (SQLException sqlException) {
 127            sqlException.printStackTrace();
                128         }
                    129
130         return ""; // if problems, return empty s
                131      }
                    132
    133      // set new database query string
   134      public void setQuery(String query)
135         throws SQLException, IllegalStateExceptio
                    136
```

```
137          // ensure database connection is availabl
     138          if (!connectedToDatabase) {
139             throw new IllegalStateException("Not C
                    140          }
                         141
   142          // specify query and execute it
143          resultSet = statement.executeQuery(query)
                         144
   145          // obtain metadata for ResultSet
 146          metaData = resultSet.getMetaData();
                         147
148          // determine number of rows in ResultSet
149          resultSet.last(); // move to last row
150          numberOfRows = resultSet.getRow(); // get
                         151
                         152
153          fireTableStructureChanged(); // notify JT
                        154       }
                         155
   156       // close Statement and Connection
 157       public void disconnectFromDatabase() {
       158       if (connectedToDatabase) {
 159          // close Statement and Connection
                  160          try {
       161             resultSet.close();
       162             statement.close();
      163             connection.close();
                  164          }
 165          catch (SQLException sqlException) {
 166             sqlException.printStackTrace();
                    167          }
168             finally { // update database connectio
 169                connectedToDatabase = false;
                  170             }
                    171          }
                    172       }
                        173    }
```

Fig. 24.25

A `TableModel` that supplies `ResultSet` data to a `JTable`.

`ResultSetTableModel` Constructor

The `ResultSetTableModel` constructor (lines 29–43) accepts four `String` arguments—the URL of the database, the username, the password and the default query to perform. The constructor throws any exceptions that occur back to the application that created the `Result-SetTableModel` object, so that the application can determine how to handle the exception (e.g., report an error and terminate the application). Line 32 establishes a connection to the database. Lines 35–36 invoke `Connection` method `createStatement` to create a `Statement` object. This example uses a version of `createStatement` that takes two arguments—the result set type and the result set concurrency. The **result set type** (Fig. 24.26) specifies whether the `ResultSet`'s cursor is able to scroll in both directions or forward only and whether the `ResultSet` is sensitive to changes made to the underlying data.

Portability Tip 24.2

Some JDBC drivers do not support scrollable `ResultSet`*s. In such cases, the driver typically returns a* `ResultSet` *in which the cursor can move only forward. For more information, see your database driver documentation.*

Common Programming Error 24.8

Attempting to move the cursor backward through a `ResultSet` *when the database driver does not support backward scrolling causes a* `SQLFeatureNotSupportedException`*.*

`ResultSet` **constant**	**Description**
`TYPE_FORWARD_ONLY`	Specifies that a `ResultSet`'s cursor can move only in the forward direction (i.e., from the first to the last row in the `ResultSet`).
`TYPE_SCROLL_INSENSITIVE`	Specifies that a `ResultSet`'s cursor can scroll in either direction and that the changes made to the underlying data during `ResultSet` processing are not reflected in the `ResultSet` unless the program queries the database again.
`TYPE_SCROLL_SENSITIVE`	Specifies that a `ResultSet`'s cursor can scroll in either direction and that the changes made to the underlying data during `ResultSet` processing are

	reflected immediately in the `ResultSet`.

Fig. 24.26

`ResultSet` constants for specifying `ResultSet` type.

`ResultSet`s that are sensitive to changes reflect those changes immediately after they're made with methods of interface `ResultSet`. If a `ResultSet` is insensitive to changes, the query that produced the `ResultSet` must be executed again to reflect any changes made. The **result set concurrency** (Fig. 24.27) specifies whether the `ResultSet` can be updated with `ResultSet`'s update methods. This example uses a `ResultSet` that is scrollable, insensitive to changes and read-only. Line 42 (Fig. 24.25) invokes method `setQuery` (lines 134–154) to perform the default query.

`ResultSet` `static` **concurrency constant**	**Description**
`CONCUR_READ_ONLY`	Specifies that a `ResultSet` can't be updated—changes to the `ResultSet` contents cannot be reflected in the database with `ResultSet`'s update methods.
`CONCUR_UPDATABLE`	Specifies that a `ResultSet` can be updated (i.e., changes to its contents can be reflected in the database with `ResultSet`'s update methods).

Fig. 24.27

`ResultSet` constants for specifying result properties.

Portability Tip 24.3

Some JDBC drivers do not support updatable `ResultSet`*s. In such cases, the driver typically returns a read-only* `ResultSet`*. For more information, see your database driver documentation.*

Common Programming Error 24.9

Attempting to update a `ResultSet` *when the database driver does not support updatable* `ResultSet`*s causes* `SQLFeatureNotSupportedException`*s.*

`ResultSetTableModel` Method `getColumnClass`

Method `getColumnClass` (lines 46–64) returns a `Class` object that represents the superclass of all objects in a

particular column. The `JTable` uses this information to configure the default cell renderer and cell editor for that column in the `JTable`. Line 54 uses `ResultSetMetaData` method `getColumnClassName` to obtain the fully qualified class name for the specified column. Line 57 loads the class and returns the corresponding `Class` object. If an exception occurs, the `catch` in lines 59–61 prints a stack trace and line 63 returns `Object.class`—the `Class` instance that represents class `Object`—as the default type. [*Note:* Line 54 uses the argument `column + 1`. Like arrays, `JTable` row and column numbers are counted from 0. However, `ResultSet` row and column numbers are counted from 1. Thus, when processing `ResultSet` rows or columns for use in a `JTable`, it's necessary to add 1 to the row or column number to manipulate the appropriate `ResultSet` row or column.]

`ResultSetTableModel` Method `getColumnCount`

Method `getColumnCount` (lines 67–82) returns the number of columns in the model's underlying `ResultSet`. Line 75 uses `ResultSetMetaData` method `getColumnCount` to obtain the number of columns in the `ResultSet`. If an exception occurs, the `catch` in lines 77– 79 prints a stack trace and line 81 returns 0 as the default number of columns.

`ResultSetTableModel` Method `getColumnName`

Method `getColumnName` (lines 85–100) returns the name of the column in the model's underlying `ResultSet`. Line 93 uses `ResultSetMetaData` method `getColumnName` to obtain the column name from the `ResultSet`. If an exception occurs, the `catch` in lines 95–97 prints a stack trace and line 99 returns the empty string as the default column name.

`ResultSetTableModel` Method `getRowCount`

Method `getRowCount` (lines 103–110) returns the number of rows in the model's underlying `ResultSet`. When method `setQuery` (lines 134–154) performs a query, it stores the number of rows in variable `numberOfRows`.

`ResultSetTableModel` Method `getValueAt`

Method `getValueAt` (lines 113–131) returns the `Object` in a particular row and column of the model's underlying `ResultSet`. Line 123 uses `ResultSet` method

`absolute` to position the `ResultSet` cursor to a specific row. Line 124 uses `ResultSet` method `getObject` to obtain the `Object` in a specific column of the current row. If an exception occurs, the `catch` in lines 126–128 prints a stack trace and line 130 returns an empty string as the default value.

`ResultSetTableModel` Method `setQuery`

Method `setQuery` (lines 134–154) executes the query it receives as an argument to obtain a new `ResultSet` (line 143). Line 146 gets the `ResultSetMetaData` for the new `ResultSet`. Line 149 uses `ResultSet` method `last` to position the `ResultSet` cursor at the last row in the `ResultSet`. [*Note:* This can be slow if the table contains many rows.] Line 150 uses `ResultSet` method `getRow` to obtain the row number for the current row in the `ResultSet`. Line 153 invokes method `fireTableStructureChanged` (inherited from class `AbstractTableModel`) to notify any `JTable` using this `ResultSetTableModel` object as its model that the structure of the model has changed. This causes the `JTable` to repopulate its rows and columns with the new `ResultSet` data. Method `setQuery` throws any exceptions that occur in its body back to the application that invoked `setQuery`.

`ResultSetTableModel` Method `disconnectFromDatabase`

Method `disconnectFromDatabase` (lines 157–172) implements an appropriate termination method for class `ResultSetTableModel`. A class designer should provide a `public` method that clients of the class must invoke explicitly to free resources that an object has used. In this case, method `disconnectFromDatabase` closes the `ResultSet`, `Statement` and `Connection` (lines 161–163). Clients of the `ResultSetTableModel` class should always invoke this method when `ResultSetTableModel` is no longer needed. Before the method releases resources, line 158 verifies whether the app is currently connected to the database. If not, the method returns. Method `disconnectFromDatabase` sets `connectedToDatabase` to `false` (line 169) to ensure that clients do not use an instance of `ResultSetTableModel` after that instance has already been terminated. The other methods in class `ResultSetTableModel` each throw an `IllegalStateException` if `connectedToDatabase` is `false`.

24.7.2 `DisplayQueryResults` App's GUI

Figure 24.28 shows the app's GUI (defined in `DisplayQueryResults.fxml`) labeled with its **fx:id**s. You've built many FXML GUIs in prior chapters, so we point out only the key elements and the event-handler methods implemented in class `DisplayQueryResults-Controller` (Fig. 24.29). For the complete layout details, open the file `DisplayQueryResults.fxml` in Scene Builder or view the FXML in a text editor. The GUI's primary layout is a `BorderPane` with the **fx:id** `borderPane`—we use this in the controller class to dynamically add a `SwingNode` containing the `JTable` to the `BorderPane`'s center (Section 24.7.3). The `BorderPane`'s top and bottom areas contain `GridPane`s with the app's other controls. The controller class defines two event-handling methods:

- `submitQueryButtonPressed` is called when the **Submit Query** `Button` is clicked.
- `applyFilterButtonPressed` is called when the the **Apply Filter** `Button` is clicked.

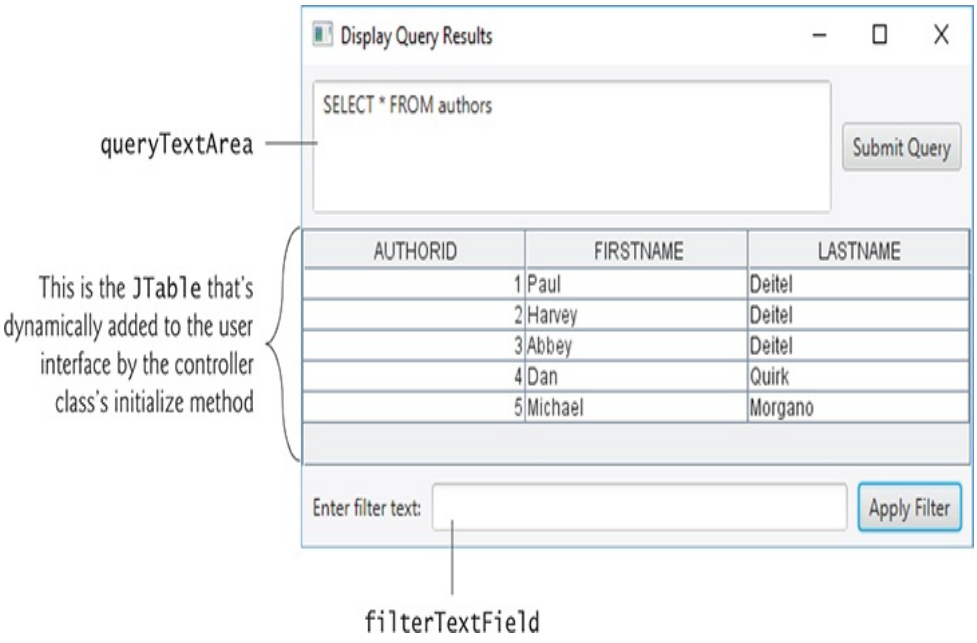


Fig. 24.28

`DisplayQueryResults` app's GUI.

Description

24.7.3 `DisplayQueryResultsController` Class

Class `DisplayQueryResultsController` (Fig. 24.29) completes the GUI, interacts with the `ResultSetTableModel` via a `JTable` object and responds to the GUI's events. We do not show the JavaFX

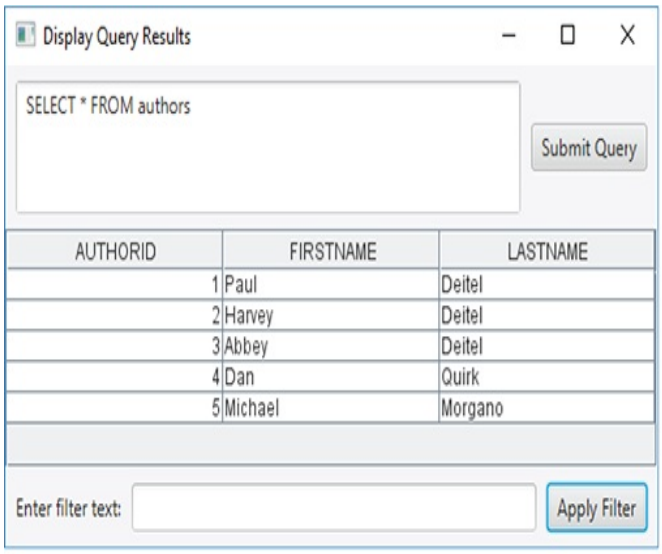
`Application` subclass here (located in `DisplayQueryResults.java`), because it performs the same tasks you've seen previously to load the app's FXML GUI and initialize the controller.

```
 1   // Fig. 24.29: DisplayQueryResultsController.jav
 2   // Controller for the DisplayQueryResults app
 3   import java.sql.SQLException;
 4   import java.util.regex.PatternSyntaxException;
 5
 6   import javafx.embed.swing.SwingNode;
 7   import javafx.event.ActionEvent;
 8   import javafx.fxml.FXML;
 9   import javafx.scene.control.Alert;
10   import javafx.scene.control.Alert.AlertType;
11   import javafx.scene.control.TextArea;
12   import javafx.scene.control.TextField;
13   import javafx.scene.layout.BorderPane;
14
15   import javax.swing.JScrollPane;
16   import javax.swing.JTable;
17   import javax.swing.RowFilter;
18   import javax.swing.table.TableModel;
19   import javax.swing.table.TableRowSorter;
20
21   public class DisplayQueryResultsController {
22      @FXML private BorderPane borderPane;
23      @FXML private TextArea queryTextArea;
24      @FXML private TextField filterTextField;
25
26      // database URL, username and password
27      private static final String DATABASE_URL = "j
28      private static final String USERNAME = "deite
29      private static final String PASSWORD = "deite
30
31      // default query retrieves all data from Auth
32      private static final String DEFAULT_QUERY = "
33
```

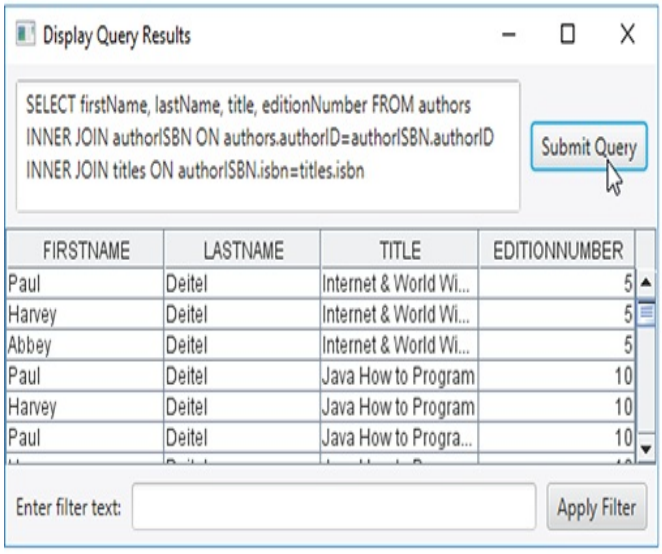
```
34      // used for configuring JTable to display and
 35      private ResultSetTableModel tableModel;
36      private TableRowSorter<TableModel> sorter;
                    37
        38      public void initialize() {
 39         queryTextArea.setText(DEFAULT_QUERY);
                    40
41         // create ResultSetTableModel and display
              42         try {
43            // create TableModel for results of DEF
44            tableModel = new ResultSetTableModel(DA
45               USERNAME, PASSWORD, DEFAULT_QUERY);
                    46
47            // create JTable based on the tableMode
48            JTable resultTable = new JTable(tableMo
                    49
 50            // set up row sorting for JTable
51            sorter = new TableRowSorter<TableModel>
 52            resultTable.setRowSorter(sorter);
                    53
54            // configure SwingNode to display JTabl
55            SwingNode swingNode = new SwingNode();
56            swingNode.setContent(new JScrollPane(re
 57            borderPane.setCenter(swingNode);
                 58         }
 59         catch (SQLException sqlException) {
60            displayAlert(AlertType.ERROR, "Database
   61               sqlException.getMessage());
62            tableModel.disconnectFromDatabase(); //
63            System.exit(1); // terminate applicatio
                 64         }
                   65      }
                    66
67      // query the database and display results in
                68      @FXML
69      void submitQueryButtonPressed(ActionEvent eve
        70         // perform a new query
              71         try {
72            tableModel.setQuery(queryTextArea.getTe
                 73         }
```

```
 74          catch (SQLException sqlException) {
75             displayAlert(AlertType.ERROR, "Database
   76                sqlException.getMessage());
                     77
78             // try to recover from invalid user que
   79              // by executing default query
                80             try {
81                tableModel.setQuery(DEFAULT_QUERY);
82                queryTextArea.setText(DEFAULT_QUERY)
                  83             }
84             catch (SQLException sqlException2) {
85                displayAlert(AlertType.ERROR, "Datab
 86                   sqlException2.getMessage());
87                tableModel.disconnectFromDatabase();
88                System.exit(1); // terminate applica
                  89             }
                    90          }
                      91       }
                          92
   93       // apply specified filter to results
                    94       @FXML
95       void applyFilterButtonPressed(ActionEvent eve
96           String text = filterTextField.getText();
                          97
      98           if (text.length() == 0) {
    99               sorter.setRowFilter(null);
                     100        }
                  101        else {
                 102            try {
103              sorter.setRowFilter(RowFilter.regexFi
                   104           }
105           catch (PatternSyntaxException pse) {
106              displayAlert(AlertType.ERROR, "Regex
     107                  "Bad regex pattern");
                   108           }
                     109        }
                       110     }
                          111
        112     // display an Alert dialog
       113       private void displayAlert(
```

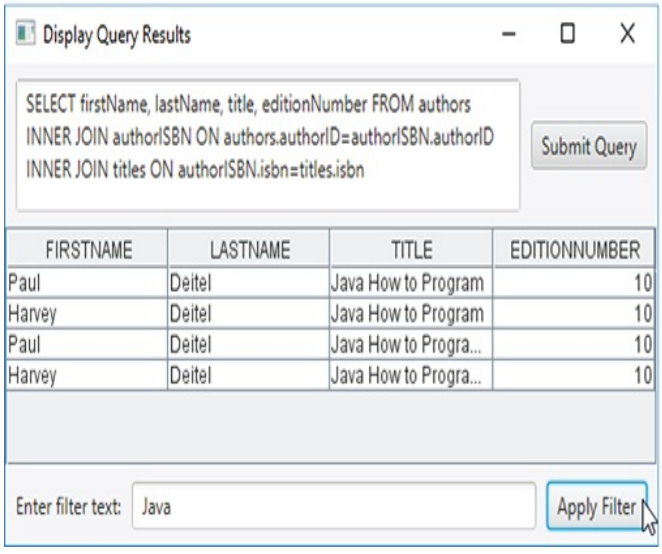
```
114         AlertType type, String title, String mess
115         Alert alert = new Alert(type);
116         alert.setTitle(title);
117         alert.setContentText(message);
118         alert.showAndWait();
119     }
120 }
```



a) Displaying all authors from the `Authors` table.



b) Displaying the authors' first and last names joined with the titles and edition numbers of the books they've authored.



c) Filtering the results of the previous query to show only the books with Java in the title.

Fig. 24.29

Controller for the `DisplayQueryResults` app.

Description

`static` Fields

Lines 27–29 and 32 declare the URL, username, password and default query that are passed to the `ResultSetTableModel` constructor to make the initial connection to the database and perform the default query.

Method `initialize`

When the `FXMLLoader` calls the controller's `initialize` method, lines 44–45 create a `ResultSetTableModel` object, assign it to instance variable `tableModel` (declared at line 35) and complete the GUI. Line 48 creates the `JTable` object that will display the `ResultSetTableModel`'s `ResultSet`. Here we use the `JTable` constructor that receives a `TableModel` object. This constructor registers the `JTable` as a listener for `TableModelEvent`s generated by the `ResultSetTableModel`. When such events occur—for example, when you enter a new query and press **Submit Query**—the `JTable` automatically updates itself, based on the `ResultSetTableModel`'s current `ResultSet`. If an

exception occurs when the `ResultSetTableModel` attempts to perform the default query, lines 59–64 catch the exception, display an `Alert` dialog (by calling method `displayAlert` in lines 113–119), close the connection and terminate the app.

`JTable`s allow users to sort rows by the data in a specific column. Line 51 creates a `TableRowSorter` (from package `javax.swing.table`) and assigns it to instance variable `sorter` (declared at line 36). The `TableRowSorter` uses our `ResultSetTableModel` to sort rows in the `JTable`. When the user clicks a particular `JTable` column's title, the `Table-RowSorter` interacts with the underlying `TableModel` to reorder the rows based on that column's data. Line 52 uses `JTable` method `setRowSorter` to specify the `TableRowSorter` for `resultTable`.

8

Lines 55–57 create and configure a `SwingNode`. This Java SE 8 class enables you to embed Swing GUI controls in JavaFX GUIs. The argument to `SwingNode` method `set-Content` is a `JComponent`—the superclass of all Swing GUI controls. In this case, we pass a new `JScrollPane` object—a subclass of `JComponent`—that we initialize with the `JTable`. A `JScrollPane` provides scrollbars for Swing GUI components that have more content to display than can fit in their area on the screen. For a `JTable`, depending on its number of rows and columns, the `JScrollPane` automatically provides vertical and horizontal scrollbars as

necessary. Line 57 attaches the `SwingNode` to the `BoderPane`'s center area.

 Software Engineering Observation 24.6

Class `SwingNode` *enables you to reuse existing Swing GUIs or specific Swing controls by embedding them in new JavaFX apps.*

Method `submitQueryButtonPressed`

When the user clicks the **Submit Query** `Button`, method `submitQueryButtonPressed` (lines 68–91) invokes `ResultSetTableModel` method `setQuery` (line 72) to execute the new query. If the user's query fails (for example, because of a syntax error in the user's input), lines 81–82 execute the default query. If the default query also fails, there could be a more serious error, so line 87 ensures that the database connection is closed and line 88 terminates the program. The screen captures in Fig. 24.29 show the results of two queries. Figure 24.29(a) shows the default query that retrieves all the data from table `Authors` of database `books`.

Figure 24.29(b) shows a query that selects each author's first name and last name from the `Authors` table and combines that information with the titles and edition numbers of all that author's books from the `Titles` table. Try entering your own queries in the text area and clicking the **Submit Query** button to execute the query.

Method `applyFilterButtonPressed`

`JTable`s can show subsets of the data from the underlying `TableModel`—this is known as filtering the data. When the user enters text in the `filterTextField` and presses the **Apply Filter** `Button`, method `applyFilterButtonPressed` (lines 94–110) executes. Line 96 obtains the filter text. If the user did not specify filter text, line 99 uses `JTable` method `setRowFilter` to remove any prior filter by setting the filter to `null`. Otherwise, line 103 uses `setRowFilter` to specify a `RowFilter` (from package `javax.swing`) based on the user's input. Class `RowFilter` provides several methods for creating filters. The `static` method `regexFilter` receives a `String` containing a regular expression pattern as its argument and an optional set of indices that specify which columns to filter. If no indices are specified, then all the columns are searched. In this example, the regular expression pattern is the text the user typed. Once the filter is set, the data

displayed in the `JTable` is updated based on the filtered `TableModel`. Figure 24.29(c) shows the results of the query in Fig. 24.29(b) filtered to show only records that contain the word `"Java"`.

Method `displayAlert`

When an exception occurs, the app calls method `displayAlert` (lines 113–119) to create and display an `Alert` dialog (package `javafx.scene.control`) containing a message. Line 115 creates the dialog, passing its `AlertType` to the constructor. The `AlertType.ERROR` constant displays an error-message dialog with a red icon containing an **X** to indicate an error. Line 116 sets the title that appears in the dialog's title bar. Line 117 sets the message that appears inside the dialog. Finally, line 118 calls `Alert` method `showAndWait`, which makes this a modal dialog. The user must close the dialog before interacting with the rest of the app.