# Criterion C: Development

Techniques Used:

**a.** GUI
**b.** try-catch blocks
**c.** SQL / throws
**d.** Encapsulation
**e.** static variables and methods
**f.** Arrays and ArrayLists

## a. Graphical User Interface

**Main Frame**

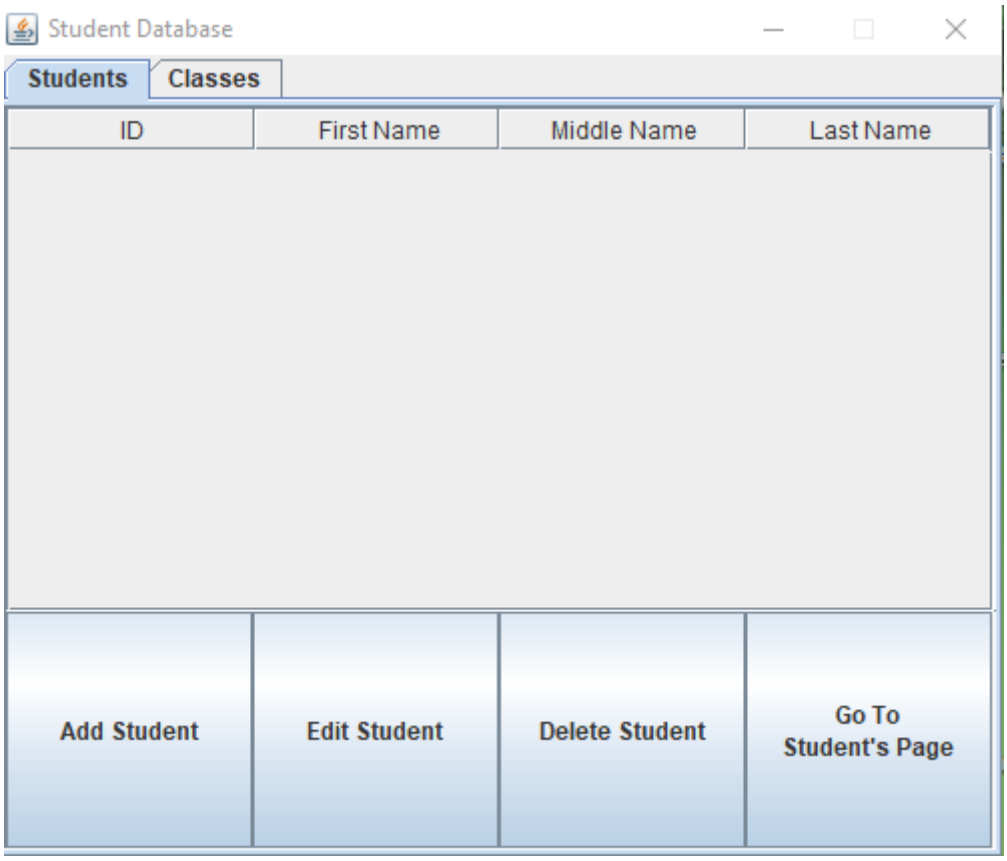


Figure 1. Panel of student table and corresponding buttons

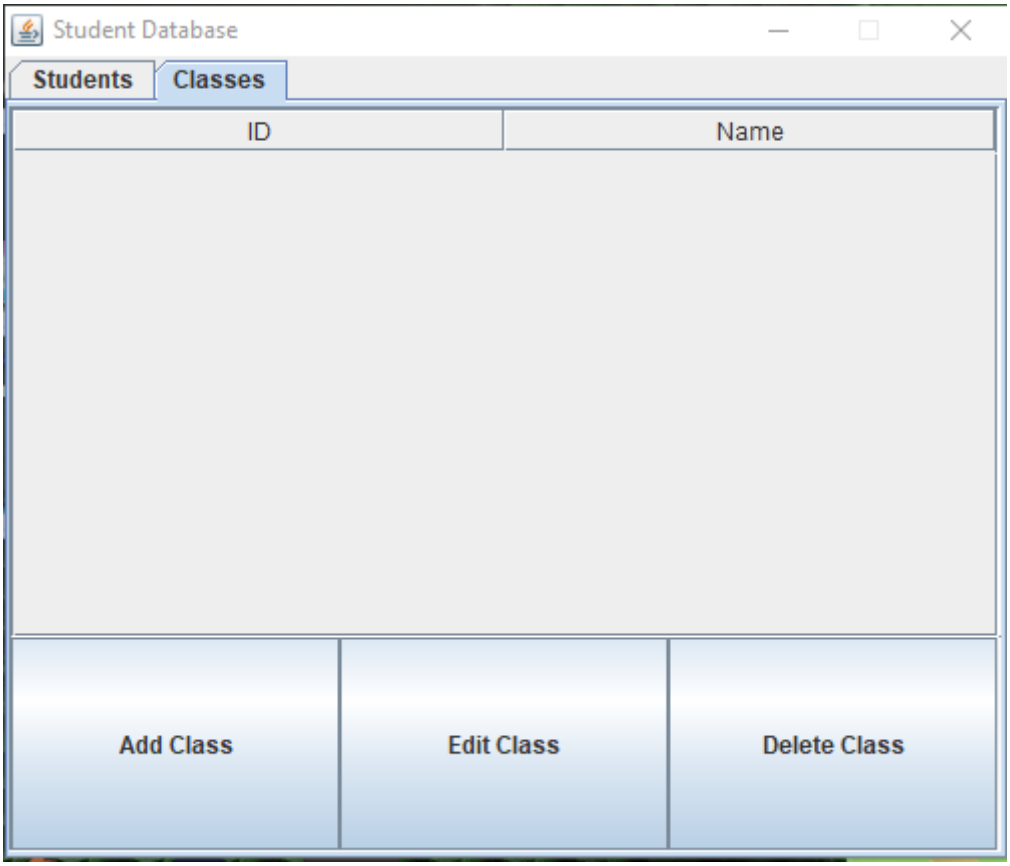


Figure 2. Panel of class table and corresponding buttons

```
tabbedPanes = new javax.swing.JTabbedPane();
studentSplitPane = new javax.swing.JSplitPane();
studentScrollPane = new javax.swing.JScrollPane();
StudentTable = new javax.swing.JTable();
studentButtonsPanel = new javax.swing.JPanel();
addingStudentButton = new javax.swing.JButton();
editStudentButton = new javax.swing.JButton();
deleteStudentButton = new javax.swing.JButton();
goToStudentPageButton = new javax.swing.JButton();
classSplitPlane = new javax.swing.JSplitPane();
classScrollPanel = new javax.swing.JScrollPane();
classTable = new javax.swing.JTable();
classButtonPanel = new javax.swing.JPanel();
addClassButton = new javax.swing.JButton();
editClassButton = new javax.swing.JButton();
deleteClassButton = new javax.swing.JButton();
```

Figure 3. Creation of GUI components for figures 1 and 2 except for the JPanel which is extended from the class

By using a JTabbedPane in figures 1 and 2, the user can switch between the following screens at any point without having to add additional buttons. By using a JSplitPlane inside each of the tabbed panes, I could set the divider line horizontally so that the top stores a JScrollPlane which stores a JTable, and the bottom stores the JButtons.

**Student Page Frame**

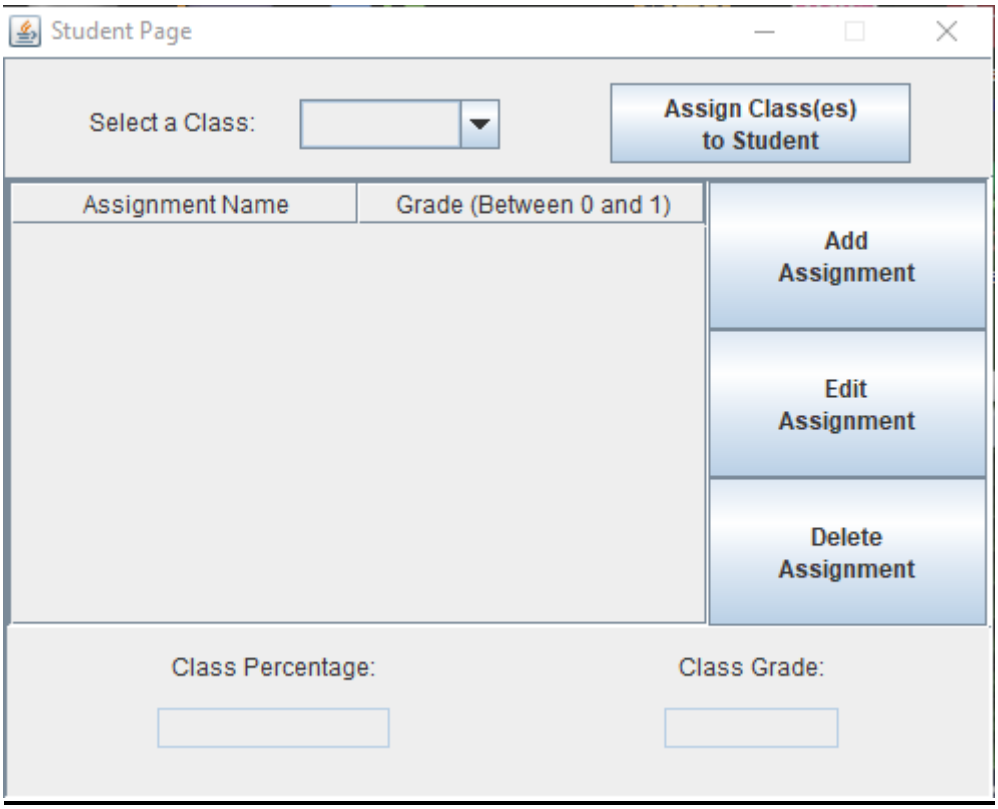


Figure 4. Student page frame accessed by “Go to Student’s Page” button once a student is made and selected

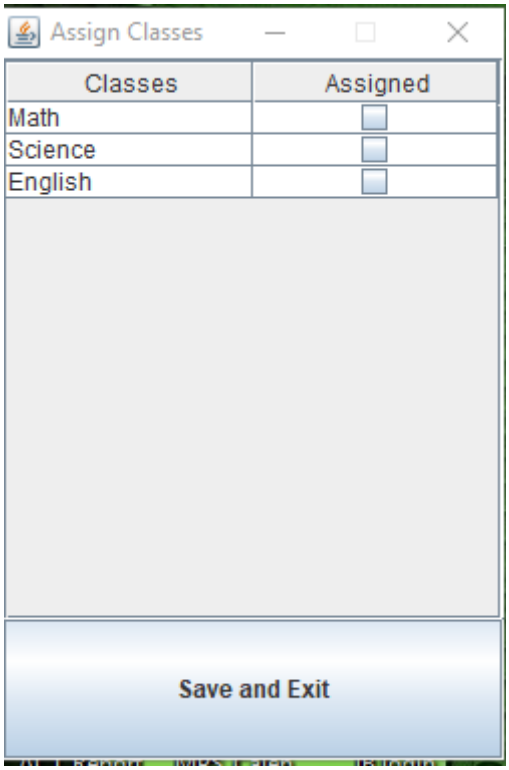


Figure 5. Assign classes frame accessed by clicking “Assign Class(es) to Student” button in figure 4

In figure 4, just like the previous figures has a JTable and JButtons for assignments, but also included a JComboBox to select the classes that were assigned to the student from the JFrame in figure 5. By adding assignments to the JTable in figure 4, the JLabels under “Class Percentage” and “Class Grade” will show the average percentage and letter grade based on all of them.

**Action Listeners**

By clicking on the “Add Student” button in figure 1, it will open another JFrame for which the user can input the student’s information. But an ActionListener method will be required to create that frame once it is clicked.

```
addingStudentButton.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        addingStudentButtonActionPerformed(evt);
    }
});
```

```java
private void addingStudentButtonActionPerformed(java.awt.event.ActionEvent evt) {
    AddingStudentFrame addFrame = new AddingStudentFrame();
    addFrame.setVisible(true);
}
```

Figure 6 & 7. adding ActionListener to JButton and creating method to open another frame

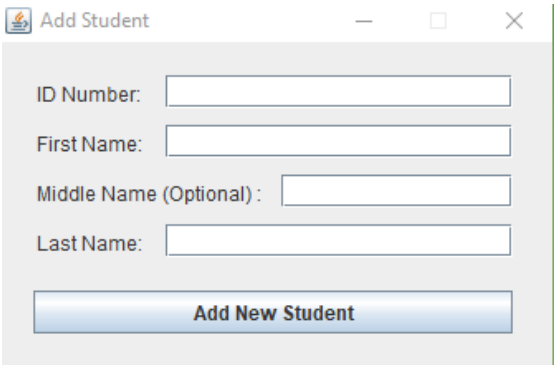


Figure 8. JFrame to add a student (Opened by “Add Student Button”)

The user can now add the following information into the JTextFields on figure 8 to add a new student to the program. But to successfully add the students, the program will need to use a try-catch block in order to prevent any incorrect inputs.

## b. Try-Catch blocks

```java
try {

 int savedID = Integer.parseInt(idTextField.getText());

 if (savedID <= 0)
 {
     JOptionPane.showMessageDialog(null, "Please enter an ID number between 1 and 999999999");
     return;
 }

 String savedFirst = firstTextField.getText();
 String savedMiddle = middleTextField.getText();
 String savedLast = lastTextField.getText();

 SQLite.addStudentToDatabse(savedID, savedFirst, savedMiddle, savedLast);

 Object[] row = {savedID, savedFirst, savedMiddle, savedLast};

 DataPanel.addStudentRowToJTable(row);
 dispose();
 }

 catch(NumberFormatException e) {
     JOptionPane.showMessageDialog(null, "Please Enter Digits Only For The ID");
 } catch (ClassNotFoundException | SQLException ex) {
     Logger.getLogger(AddingStudentFrame.class.getName()).log(Level.SEVERE, null, ex);
 }
```

Figure 9. Try catch block from the ActionListener method on the “Add New Student” button in figure

Starting from Line 84 in figure 9, in order to get the ID Number from the corresponding JTextField, idTextField.getText() is called which returns the value in the field as a string. But to convert it into an integer, I would have to call the parseInt() method from the Integer wrapper class. If the textField cannot be converted into the integer due to letters or other symbols being present, it will call a NumberFormatException error which I can use in my catch to create a JOptionPane telling the user to input “Digits Only.” Although I can still save the textID as a string and avoid the use of a try-catch, I still need it as an int to later pass it to my DataPanel method in line 100 to add the ID to the student JTable.

### c. SQL / throws

```
54  public static void addStudentToDatabse (int id, String first, String middle, String last) throws ClassNotFoundException, SQLException
55  {
56      Class.forName("org.sqlite.JDBC");
        Connection conn = DriverManager.getConnection("jdbc:sqlite:Database.db");
58
59      String sqlStatement = "INSERT INTO Students (ID,First,Middle,Last) " +
60      "VALUES (" + id + "," + "'" + first + "'" + "," +"'" + middle + "'" + "," + "'" + last + "'" + ");";
61
62      Statement state = conn.createStatement();
63      state.executeUpdate(sqlStatement);
64
65      state.close();
66      conn.close();
67  }
```

Figure 10. SQL method that add students into database

```
69      public static void deleteStudentFromDatabase (String id) throws ClassNotFoundException, SQLException
70      {
71          Class.forName("org.sqlite.JDBC");
            Connection conn = DriverManager.getConnection("jdbc:sqlite:Database.db");
73
74          String sqlStatement = "DELETE FROM Students WHERE ID = " + id + ";";
75
76          Statement state = conn.createStatement();
77          state.executeUpdate(sqlStatement);
78
79          state.close();
80          conn.close();
81
82      }
83
84      public static void editStudentFromDatabase (int originalID, int newID, String first, String middle, String last) throws ClassNotFoundException,
85      {
86          Class.forName("org.sqlite.JDBC");
            Connection conn = DriverManager.getConnection("jdbc:sqlite:Database.db");
88
89          String sqlStatement = "UPDATE Students "
90                                + "SET ID = " + newID
91                                + ", First = " + "'" + first + "'"
92                                + ", Middle = " + "'" + middle + "'"
93                                + ", Last = " + "'" + last + "'"
94                                + " WHERE ID = " + originalID + ";";
95
96          Statement state = conn.createStatement();
97          state.executeUpdate(sqlStatement);
98
99          state.close();
100         conn.close();
101     }
102
```

Figure 11. SQL methods to edit and delete students from database

In looking at figure 9, on line 96 a SQLite.addStudentToDatabase() method is called which took all the variables I saved from the JTextFields, and puts them into the database under a “Students” table which has the same columns as the JTable seen in figure 1. In figure 10, adding the students to the database will require concatenation inside the sqlStatement String which will then be executed by the Statement object. Figures 10 and 11 should give a general idea behind the order of how most SQL methods are made in this program. Even looking at the throws of each method, they all have the same throws exceptions when trying to connect to the Java JDBC API to use SQLite (ClassNotFoundException) and creating the connection to the database file (SQLException).

## Restarting Program

```java
    public static void DatabaseToJTables (JTable table1, JTable table2) throws ClassNotFoundException, SQLException
    {

        Class.forName("org.sqlite.JDBC");
        Connection conn = DriverManager.getConnection("jdbc:sqlite:Database.db");

        String sqlStatement1 = "SELECT * FROM Students";
        String sqlStatement2 = "SELECT * FROM Classes";

        Statement state1 = conn.createStatement();
        Statement state2 = conn.createStatement();

        ResultSet rs1 = state1.executeQuery(sqlStatement1);
        ResultSet rs2 = state2.executeQuery(sqlStatement2);

        DefaultTableModel modelOne = (DefaultTableModel)table1.getModel();
        DefaultTableModel modelTwo = (DefaultTableModel)table2.getModel();

        while(rs1.next())
            {
                int id = rs1.getInt("ID");
                String first = rs1.getString("First");
                String middle = rs1.getString("Middle");
                String last = rs1.getString("Last");

                Object[] databaseRow1 = {id, first, middle, last};
                modelOne.addRow(databaseRow1);
            }

        while(rs2.next())
            {
                int id = rs2.getInt("ID");
                String name = rs2.getString("Name");

                Object[] databaseRow2 = {id, name};
                modelTwo.addRow(databaseRow2);
            }
    }
```

Figure 12. SQL method that gathers all the data from the database, and updates the JTables with it. Normally used when the project first starts

```java
public class DataPanel extends JPanel {

    public DataPanel() throws ClassNotFoundException, SQLException{
        initComponents();
        centerTables();
        SQLite.DatabaseToJTables(StudentTable, classTable);
    }
```

Figure 13. Method from figure 12 being called into the constructor of the JPanel in figures 1 and 2.

A big advantage about using a database to store all the information, is that once the program is closed, the information stored when using the program is saved to the database. So when the

program runs again, I can call a method in the constructor of the JPanel to send all the information to the student and class JTables.

### d. Encapsulation

```
private javax.swing.JTextField firstTextField;
private java.awt.Label id;
private javax.swing.JTextField idTextField;
private java.awt.Label last;
private javax.swing.JTextField lastTextField;
```

Figure 14. Private JTextField variables from EditingSutdentFrame class

```
private void editStudentButtonActionPerformed(java.awt.event.ActionEvent evt) {
    if (StudentTable.getSelectionModel().isSelectionEmpty())
    {
        JOptionPane.showMessageDialog(null, "Please Select a student from the table");
        return;
    }

    int rowIndex = StudentTable.getSelectedRow();

    Object id = StudentTable.getModel().getValueAt(rowIndex, 0);

    DataPanel.setOriginalID((Integer)id);

    Object first = StudentTable.getModel().getValueAt(rowIndex, 1);
    Object middle = StudentTable.getModel().getValueAt(rowIndex, 2);
    Object last = StudentTable.getModel().getValueAt(rowIndex, 3);

    EditingStudentFrame editFrame = new EditingStudentFrame();

    JTextField idField = editFrame.getIdTextField();
    JTextField firstField = editFrame.getFirstTextField();
    JTextField middleField = editFrame.getMiddleTextField();
    JTextField lastField = editFrame.getLastTextField();

    idField.setText(Integer.toString((int)id));
    firstField.setText((String)first);
    middleField.setText((String)middle);
    lastField.setText((String)last);

    editFrame.setIdTextField(idField);
    editFrame.setFirstTextField(firstField);
    editFrame.setMiddleTextField(middleField);
    editFrame.setLastTextField(lastField);

    editFrame.setVisible(true);
}
```

Figure 15. Use of encapsulation on the JTextField variables in figure 14 for the ActionListener method on the “Edit Student” button

Once the editing student JFrame is made when the “edit button” is clicked, a convenient feature I added to the frame is to have it set up where the information on the selected row in the student JTable shows up again in the JTextFields. But in order to do that, getters and setters would

have to be made for each of the JTextField objects so that in the ActionListener method can get each of the fields (figure 15. lines 216-219), set the text inside each field equal to the info on the selected row (221-224), and set it back to the text fields in the frame (226-229).

### e. Static Variables and Methods

Throughout my project, a struggle I faced was updating my main visual objects in my program like my JTables and JTextFields which often had to be done from different classes and JFrames. In doing some research I found that a solution to that was to make the objects static so I don't have to reference an instance of those objects every time when trying to change it (1BestCsharp blog, 2019).

```
private static javax.swing.JTable classTable;
```

```
private static javax.swing.JTable StudentTable;
```

```
private static javax.swing.JTable assignmentJTable;
private javax.swing.JScrollPane assignmentJTableScrollPane;
private javax.swing.JPanel bottomPanel;
private java.awt.Label classGradeLabel;
private static javax.swing.JTextField classGradeTextField;
private java.awt.Label classPercentageLabel;
private static javax.swing.JTextField classPercentageTextField;
private static javax.swing.JComboBox<String> classSelectionComboBox;
```

Figure 16-18. list of GUI static variables objects used in product

```
public static void addStudentRowToJTable(Object[] dataRow)
{
    DefaultTableModel model = (DefaultTableModel) StudentTable.getModel();
    model.addRow(dataRow);
}
```

Figure 19. static method that uses the student JTable to add rows to it (also referenced in line 100 in figure 7)

```
public static void editStudentRowOnJTable(int id, String first, String middle, String last)
{
    int rowIndex = StudentTable.getSelectedRow();
    DefaultTableModel model = (DefaultTableModel) StudentTable.getModel();
    model.setValueAt(id, rowIndex, 0);
    model.setValueAt(first, rowIndex, 1);
    model.setValueAt(middle, rowIndex, 2);
    model.setValueAt(last, rowIndex, 3);
}
```

Figure 20. static methods that edits a student row on the Jtable

```java
public static void addClassRowToJTable (Object[] dataRow)
{
    DefaultTableModel model = (DefaultTableModel) classTable.getModel();
    model.addRow(dataRow);
}
```

```java
public static void editClassRowOnJTable(int id, String name)
{
    int rowIndex = classTable.getSelectedRow();
    DefaultTableModel model = (DefaultTableModel) classTable.getModel();
    model.setValueAt(id, rowIndex, 0);
    model.setValueAt(name, rowIndex, 1);
}
```

Figure 21-22. Add and edit methods for classTable that complete the same task in figures 19 & 20

```java
13  public class SQLite {
14
15      public static void DatabaseToJTables (JTable table1, JTable table2) throws ClassNotFoundException, SQLException
16      {...37 lines }
53
54      public static void addStudentToDatabse (int id, String first, String middle, String last) throws ClassNotFoundException, SQLException
55      {...10 lines }
65
66      public static void deleteStudentFromDatabase (String id) throws ClassNotFoundException, SQLException
67      {...11 lines }
78
79      public static void editStudentFromDatabase (int originalID, int newID, String first, String middle, String last) throws ClassNotFoundException, SQLException
80      {...14 lines }
94
95      public static void addClassToDatabase (int id, String name) throws ClassNotFoundException, SQLException
96      {...10 lines }
106
107     public static void deleteClassFromDatabse(String id) throws ClassNotFoundException, SQLException
108     {...9 lines }
117
118     public static void editClassFromDatabase(int originalID, int newID, String name) throws ClassNotFoundException, SQLException
119     {...12 lines }
131
132     public static void getAndAddClassNamesFromDatabase(JTable table) throws SQLException, ClassNotFoundException
133     {...17 lines }
150
151     public static ArrayList<String> saveClassSelectionToDatabase(JTable table) throws SQLException, ClassNotFoundException
152     {...47 lines }
199
200     public static ArrayList<String> getAssignedClasses (int id) throws ClassNotFoundException, SQLException
201     {...19 lines }
220
221     public static void addAssignmentToDatabase (String classSelected, String name, double grade) throws ClassNotFoundException, SQLException
222     {...14 lines }
236     public static void deleteAssignmentFromDatbase (String classSelected, String name, double grade) throws ClassNotFoundException, SQLException
237     {...10 lines }
247     public static void editAssignmentFromDatabase(String classSelected, String name, double grade, String assignmentName) throws ClassNotFoundException, SQLException
248     {...12 lines }
260
261     public static void assignmentDatabasetoJTable(JTable table, String classSelected) throws ClassNotFoundException, SQLException
262     {...19 lines }
281 }
282
```

Figure 23. List of all SQLite methods in SQLite class that are all static methods for quick access in other classes

### f. Arrays and ArrayLists

In looking at line 98 in figure 9, the majority of the arrays I used in my program was to save the information collected by the JTextFields, put it in an object array, and send that to my addStudent method in figure 19 to update the student JTable. However, I did use an ArrayList to save the classes selected from figure 5, so they can be updated to the JComboBox in figure 4.

```
158
159     public static ArrayList<String> saveClassSelectionToDatabase(JTable table) throws SQLException, ClassNotFoundException
160     {
161         Class.forName("org.sqlite.JDBC");
162         Connection conn = DriverManager.getConnection("jdbc:sqlite:Database.db");
163         Statement state = conn.createStatement();
164         ArrayList<String> classList = new ArrayList<>();
165
166         for (int i = 0; i < table.getRowCount(); i++)
167         {
168             Object assigned = table.getValueAt(i, 1);
169
170             if (assigned == null) {
171
172             }
173
174             else if ((boolean)assigned == true)
175             {
176                 String classSelected = (String)table.getValueAt(i, 0);
177                 classList.add(classSelected);
178                 String sqlStatement = "SELECT ID FROM Classes WHERE Name = " + "'" + classSelected + "'";
179                 ResultSet rs = state.executeQuery(sqlStatement);
180                 int classID = rs.getInt(1);
181                 sqlStatement = "INSERT INTO Assigned_Classes (Student_ID, Class_ID)" +
182                                " VALUES (" + DataPanel.getOriginalID() + ", " + classID + ")";
183                 state.executeUpdate(sqlStatement);
184             }
185
186             else if ((boolean)assigned == false)
187             {
188                 String classSelected = (String)table.getValueAt(i, 0);
189                 String sqlStatement = "DELETE FROM Assigned_Classes WHERE Student_ID = " + DataPanel.getOriginalID() +
190                                " AND Class_ID = (SELECT ID FROM Classes WHERE Name = '" + classSelected + "')";
191                 state.executeUpdate(sqlStatement);
192             }
193         }
194
195         String sqlStatement = "DELETE   FROM Assigned_Classes " +
196                               "WHERE    ROWID NOT IN" +
197                               "         (" +
198                               "         SELECT  min(ROWID)" +
199                               "         FROM    Assigned_Classes" +
200                               "         GROUP BY" +
201                               "                 Student_ID" +
202                               "         ,       Class_ID" +
203                               "         )";
204         state.executeUpdate(sqlStatement);
205         return classList;
206     }
```

Figure 24. SQL return method that saves the class selection from figure 4 into the ClassList ArrayList and gets returned at the end

Although figure 24 shows the use of an ArrayList, it also shows some of the more complex SQL methods that are used in my program. In my method, the parameter I have is JTable as this method is normally called to take in the JTable from figure 4. Then I check the boolean column (the second column) to see whether any of the classes were selected or not. A tricky thing about booleans in JTables, is that if a boolean cell is not clicked at all in the table, then the cell returns a null instead of a default boolean value of false. The only way a false is returned is if a boolean cell is clicked to turn true, then clicked again to make it false. So I have to first save the value at the cell as an object and check if it's null before I cast it into a boolean. The way the boolean column is set up can still be useful, as I can use the false as a way to check if an assigned class has been removed from the student and so I can then delete it from the database.

When it comes to the classList ArrayList, I can use it to add the classes under the condition that the boolean is true, save those classes to the database, and return that ArrayList to where it was called to set up the JComboBox from figure 4.

```
private void goToStudentPageButtonActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        if (StudentTable.getSelectionModel().isSelectionEmpty())
        {
            JOptionPane.showMessageDialog(null, "Please Select a student from the table");
            return;
        }

        int rowIndex = StudentTable.getSelectedRow();
        Object id = StudentTable.getModel().getValueAt(rowIndex, 0);
        setOriginalID((Integer)id);

        ArrayList<String> classList = SQLite.getAssignedClasses(originalID);
        StudentPageFrame pageFrame = new StudentPageFrame();
        StudentPageFrame.initializeComboBox(classList);

        pageFrame.setVisible(true);
    } catch (ClassNotFoundException | SQLException ex) {
        Logger.getLogger(DataPanel.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Figure 25. "Go To Student's Page" button ActionListener method that calls a SQLite class to retrieve the classes selected for the student to initialize the JComboBox

```java
public static ArrayList<String> getAssignedClasses (int id) throws ClassNotFoundException, SQLException
{
    ArrayList<String> classList = new ArrayList<>();

    Class.forName("org.sqlite.JDBC");
    Connection conn = DriverManager.getConnection("jdbc:sqlite:Database.db");
    Statement state = conn.createStatement();

    String sqlStatement = "SELECT Name FROM Classes WHERE ID IN" +
    "(SELECT Class_ID FROM Assigned_Classes WHERE Student_ID = (" + id + "))";
    ResultSet rs = state.executeQuery(sqlStatement);

    while(rs.next())
    {
        String classResult = rs.getString(1);
        classList.add(classResult);
    }

    return classList;
}
```

Figure 26. SQLite method called in figure 22 that gets the assigned classes from the student at the database

Here are a few more instances where an ArrayList is used, all with the same purpose of gathering a list of selected classes from the student and sending it to the ActionListener to initialize the JComboBox. Although the JComboBox gets updated every time the selection of classes is saved, it needs to get updated also when the frame is first made. That is where figure 25 plays a role, as it calls the SQLite class in figure 26, and sends an ArrayList of the selected classes to an initializeComboBox method that adds all the class names from the list into the JComboBox.

**Word Count excluding captions: 1126**