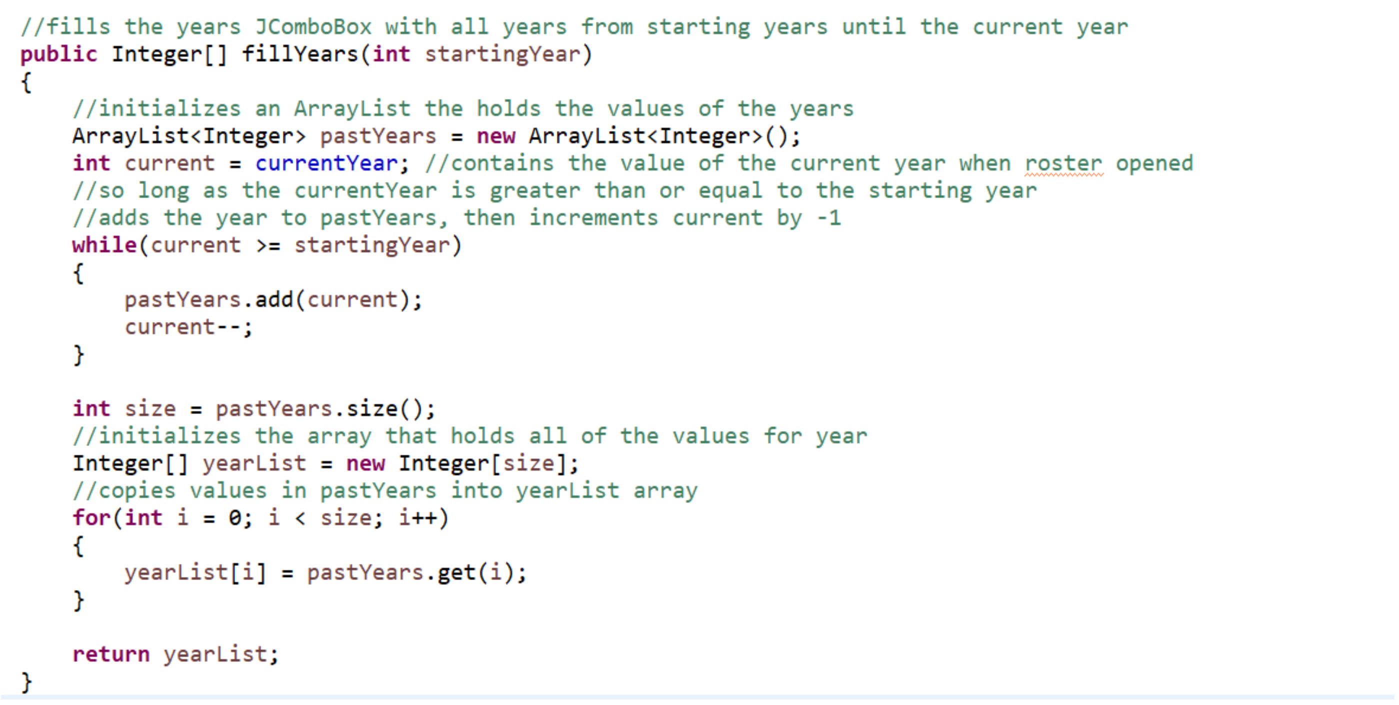
# Criterion C: Development

*(Word Count: 1003)*

## Table of Contents

## Arrays

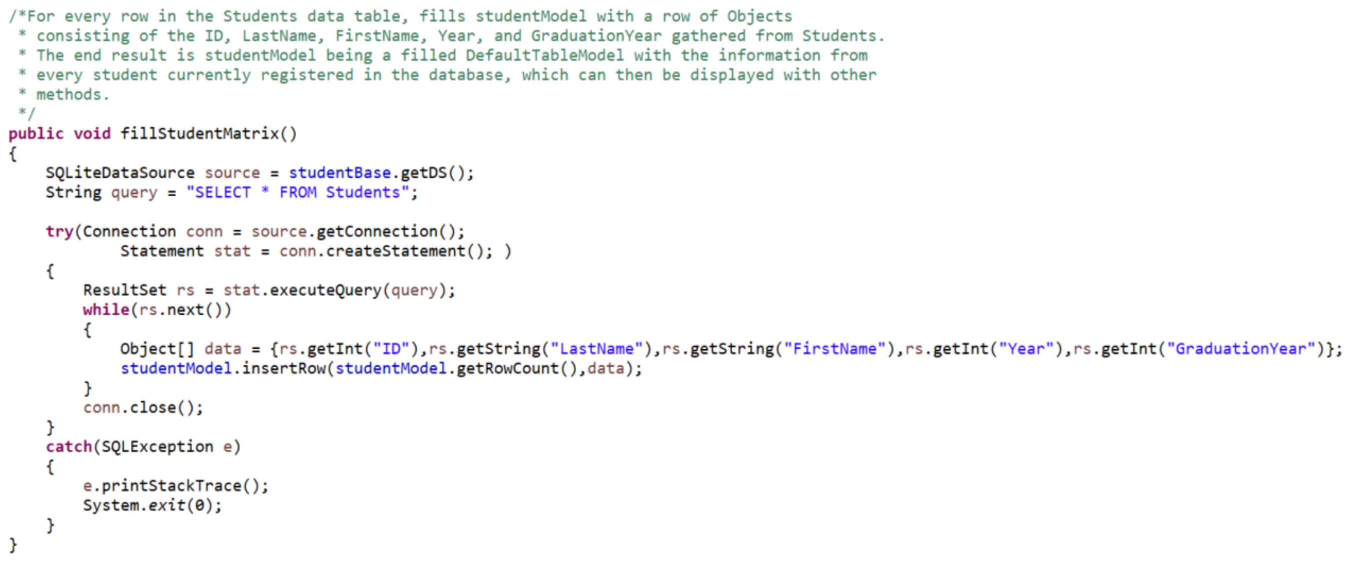


```
//fills the years JComboBox with all years from starting years until the current year
public Integer[] fillYears(int startingYear)
{
    //initializes an ArrayList the holds the values of the years
    ArrayList<Integer> pastYears = new ArrayList<Integer>();
    int current = currentYear; //contains the value of the current year when roster opened
    //so long as the currentYear is greater than or equal to the starting year
    //adds the year to pastYears, then increments current by -1
    while(current >= startingYear)
    {
        pastYears.add(current);
        current--;
    }

    int size = pastYears.size();
    //initializes the array that holds all of the values for year
    Integer[] yearList = new Integer[size];
    //copies values in pastYears into yearList array
    for(int i = 0; i < size; i++)
    {
        yearList[i] = pastYears.get(i);
    }

    return yearList;
}
```

An Array was used to store the data for the years in AdminViewRoster, which can be seen the “Filter Students By Year” button is selected and the dropdown appears. The dropdown was filled with the method fillYears(int startYear), and contains integers representing the year  starting from startYear until 2020. Since there would be no need to edit the data once the years were initialized, the most efficient data type was Arrays. Arrays were best suited for the task because they allowed for each value to be accessed individually, and unlike ArrayLists, take up less memory.

## 2D Arrays – Matrixes

In all instances where information from a database needed to be displayed in a table, a DefaultTableModel was filled using a matrix. Matrixes are an Array of Arrays, where each row is its own separate array. They are the most convenient for tables because they can easily represent each row and column. Take, for example, fillStudentMatrix() in AdminViewRoster.

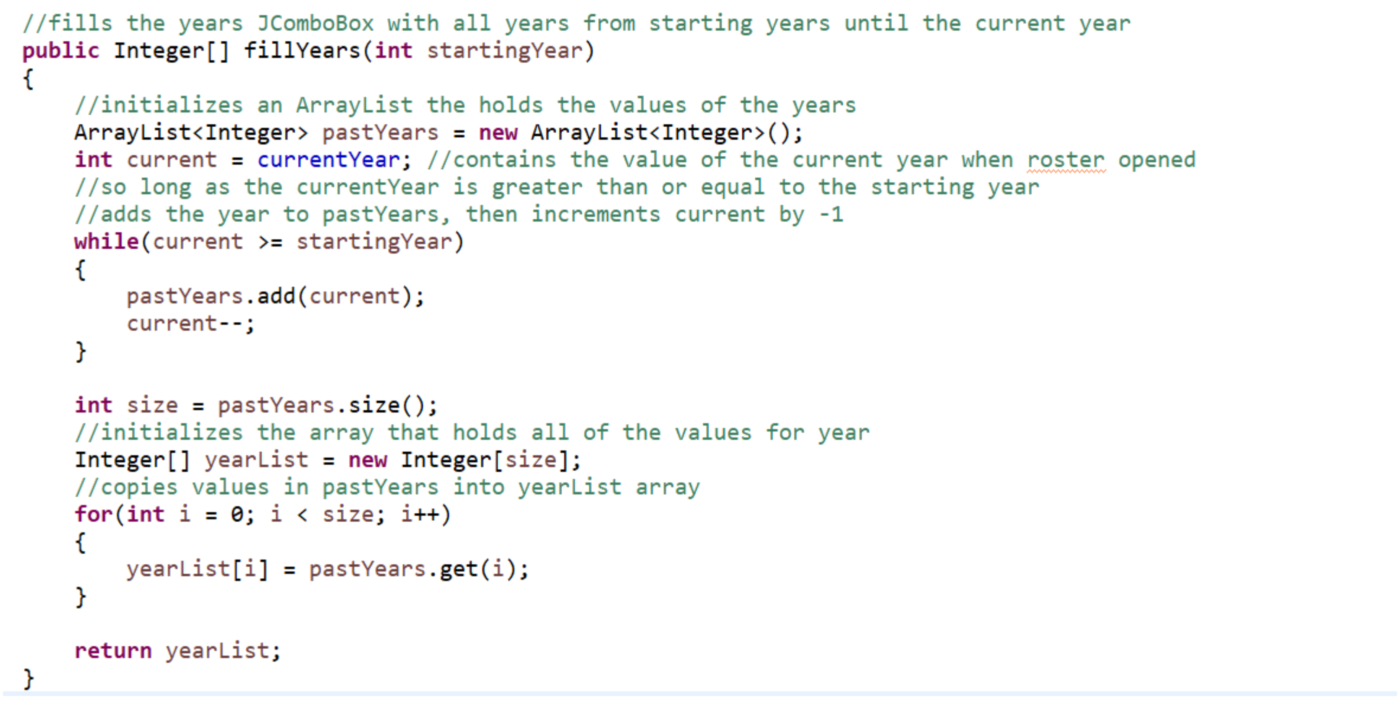


```
/*For every row in the Students data table, fills studentModel with a row of Objects
 * consisting of the ID, LastName, FirstName, Year, and GraduationYear gathered from Students.
 * The end result is studentModel being a filled DefaultTableModel with the information from
 * every student currently registered in the database, which can then be displayed with other
 * methods.
 */
public void fillStudentMatrix()
{
    SQLiteDataSource source = studentBase.getDS();
    String query = "SELECT * FROM Students";

    try(Connection conn = source.getConnection();
            Statement stat = conn.createStatement(); )
    {
        ResultSet rs = stat.executeQuery(query);
        while(rs.next())
        {
            Object[] data = {rs.getInt("ID"),rs.getString("LastName"),rs.getString("FirstName"),rs.getInt("Year"),rs.getInt("GraduationYear")};
            studentModel.insertRow(studentModel.getRowCount(),data);
        }
        conn.close();
    }
    catch(SQLException e)
    {
        e.printStackTrace();
        System.exit(0);
    }
}
```

The DefaultTableModel studentModel has five columns: ID, LastName, FirstName, Year, and GraduationYear. These columns are stored in an array, and consist of the first row in the Array which stores all of studentModel's data. To further fill studentModel, fillStudentMatrix() scans a row from the Students table in the database, then creates an Array called data storing the information obtained from that row. This Array is then added to studentModel, essentially reinitializing it as an Array containing two Arrays: the Array with the column names, and data. This process is repeated until every row from Students has been input into studentModel. With studentModel now filled, the code initializes a JTable to display the data, with all rows and columns displayed in table format.

## ArrayLists

While used in only the fillYears(int startingYear) method from AdminViewRoster, ArrayLists were nonetheless efficient for temporarily storing data.

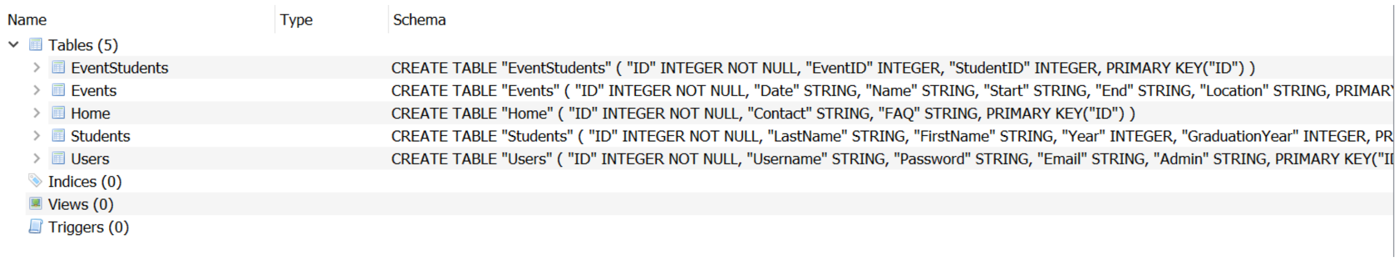


```
//fills the years JComboBox with all years from starting years until the current year
public Integer[] fillYears(int startingYear)
{
    //initializes an ArrayList the holds the values of the years
    ArrayList<Integer> pastYears = new ArrayList<Integer>();
    int current = currentYear; //contains the value of the current year when roster opened
    //so long as the currentYear is greater than or equal to the starting year
    //adds the year to pastYears, then increments current by -1
    while(current >= startingYear)
    {
        pastYears.add(current);
        current--;
    }

    int size = pastYears.size();
    //initializes the array that holds all of the values for year
    Integer[] yearList = new Integer[size];
    //copies values in pastYears into yearList array
    for(int i = 0; i < size; i++)
    {
        yearList[i] = pastYears.get(i);
    }

    return yearList;
}
```

While ArrayLists take up more memory than Arrays, it is more convenient to add information to an ArrayList. For Arrays, the number of data slots is pre-set, and changing the number of slots requires re-initializing the Array. Meanwhile, ArrayLists can freely add or remove information without requiring re-ini-tializaiton. For this reason, the ArrayList pastYears was used to store the Integers representing the years the club was active, with each new year being added to the end of the ArrayList. Once all values were added to pastYears, it was simple to copy them over to the Array yearList, so that an Integer Array could by returned by the method.

## Database

As the name suggests, databases are a collection of data. The database JapaneseClub is the backbone of this project, as every class reads, inserts, or deletes from it. Databases were used instead of reading and writing to text files because it requires less files to work. If text files were used, at least four would be needed. But with databases, only one database with several tables is needed. JapaneseClub was constructed using SQLite in the DB Browser for SQLite, and is shown below.



| Name | Type | Schema |
|---|---|---|
| Tables (5) | | |
| EventStudents | | CREATE TABLE "EventStudents" ( "ID" INTEGER NOT NULL, "EventID" INTEGER, "StudentID" INTEGER, PRIMARY KEY("ID") ) |
| Events | | CREATE TABLE "Events" ( "ID" INTEGER NOT NULL, "Date" STRING, "Name" STRING, "Start" STRING, "End" STRING, "Location" STRING, PRIMAR |
| Home | | CREATE TABLE "Home" ( "ID" INTEGER NOT NULL, "Contact" STRING, "FAQ" STRING, PRIMARY KEY("ID") ) |
| Students | | CREATE TABLE "Students" ( "ID" INTEGER NOT NULL, "LastName" STRING, "FirstName" STRING, "Year" INTEGER, "GraduationYear" INTEGER, PR |
| Users | | CREATE TABLE "Users" ( "ID" INTEGER NOT NULL, "Username" STRING, "Password" STRING, "Email" STRING, "Admin" STRING, PRIMARY KEY("I |
| Indices (0) | | |
| Views (0) | | |
| Triggers (0) | | |

JapaneseClub consists of five tables: EventStudents, Events, Home, Students, and Users. Each table consists of a row of information and a column specifying the data type of said information.

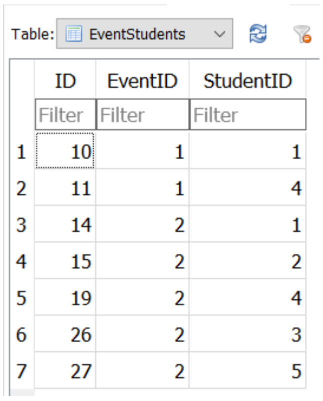


Table: EventStudents

| | ID | EventID | StudentID |
|---|---|---|---|
| 1 | 10 | 1 | 1 |
| 2 | 11 | 1 | 4 |
| 3 | 14 | 2 | 1 |
| 4 | 15 | 2 | 2 |
| 5 | 19 | 2 | 4 |
| 6 | 26 | 2 | 3 |
| 7 | 27 | 2 | 5 |

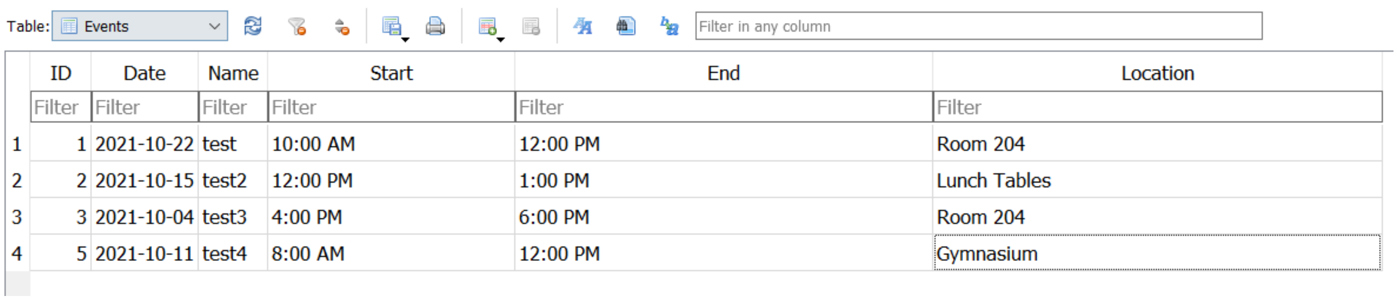


Table: Events

Filter in any column

| | ID | Date | Name | Start | End | Location |
|---|---|---|---|---|---|---|
| 1 | 1 | 2021-10-22 | test | 10:00 AM | 12:00 PM | Room 204 |
| 2 | 2 | 2021-10-15 | test2 | 12:00 PM | 1:00 PM | Lunch Tables |
| 3 | 3 | 2021-10-04 | test3 | 4:00 PM | 6:00 PM | Room 204 |
| 4 | 5 | 2021-10-11 | test4 | 8:00 AM | 12:00 PM | Gymnasium |

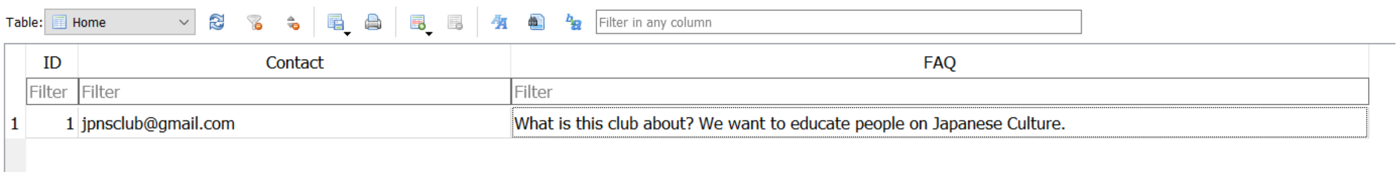


Table: Home

| | ID | Contact | FAQ |
|---|---|---|---|
| 1 | 1 | jpnsclub@gmail.com | What is this club about? We want to educate people on Japanese Culture. |

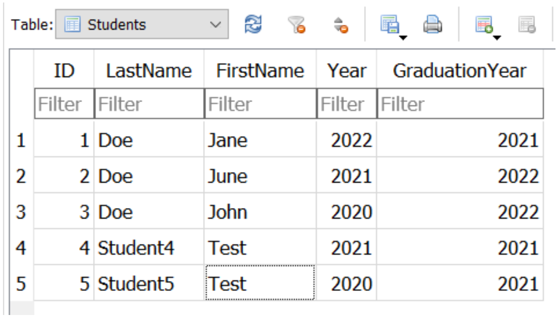


Table: Students

| | ID | LastName | FirstName | Year | GraduationYear |
|---|---|---|---|---|---|
| 1 | 1 | Doe | Jane | 2022 | 2021 |
| 2 | 2 | Doe | June | 2021 | 2022 |
| 3 | 3 | Doe | John | 2020 | 2022 |
| 4 | 4 | Student4 | Test | 2021 | 2021 |
| 5 | 5 | Student5 | Test | 2020 | 2021 |

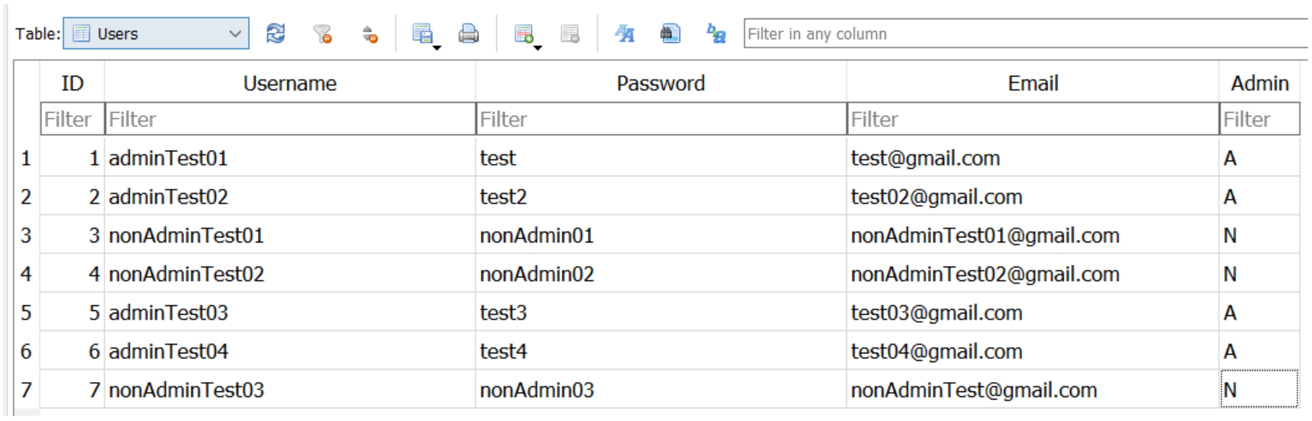
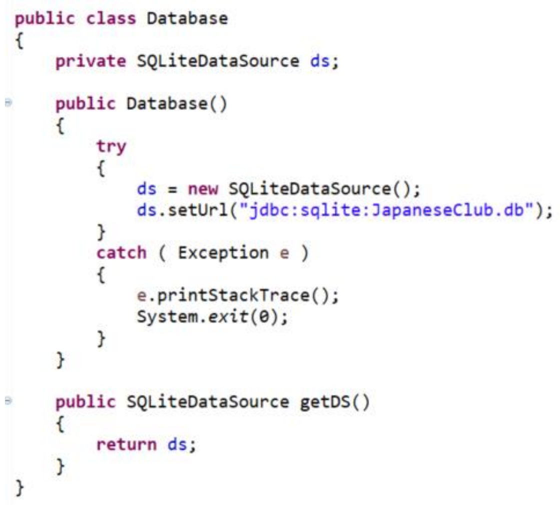


Table: Users

| | ID | Username | Password | Email | Admin |
|---|---|---|---|---|---|
| 1 | 1 | adminTest01 | test | test@gmail.com | A |
| 2 | 2 | adminTest02 | test2 | test02@gmail.com | A |
| 3 | 3 | nonAdminTest01 | nonAdmin01 | nonAdminTest01@gmail.com | N |
| 4 | 4 | nonAdminTest02 | nonAdmin02 | nonAdminTest02@gmail.com | N |
| 5 | 5 | adminTest03 | test3 | test03@gmail.com | A |
| 6 | 6 | adminTest04 | test4 | test04@gmail.com | A |
| 7 | 7 | nonAdminTest03 | nonAdmin03 | nonAdminTest@gmail.com | N |

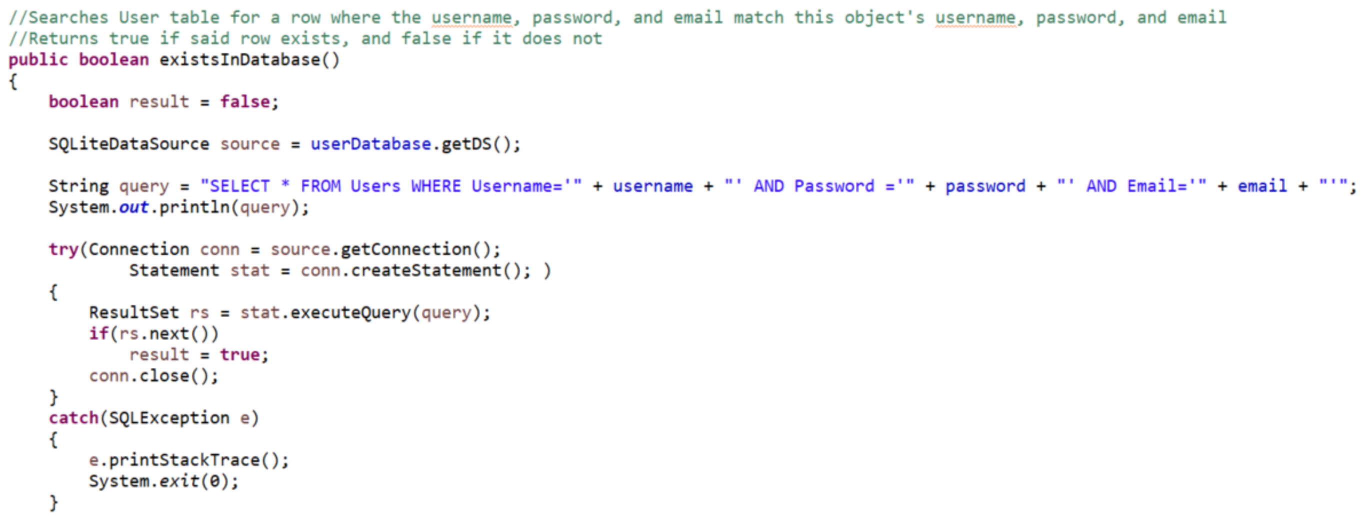
To access a database, a connection must be built. The initial pathway to the database JapaneseClub is built in the class Database, which creates the SQLiteDataSource ds by setting the URL to JapaneseClub (McDonald). By sending an SQLite command, also known as a query, to the database through the ds, various effects can be achieved.



```java
public class Database
{
    private SQLiteDataSource ds;

    public Database()
    {
        try
        {
            ds = new SQLiteDataSource();
            ds.setUrl("jdbc:sqlite:JapaneseClub.db");
        }
        catch ( Exception e )
        {
            e.printStackTrace();
            System.exit(0);
        }
    }

    public SQLiteDataSource getDS()
    {
        return ds;
    }
}
```

## Searching From a Database

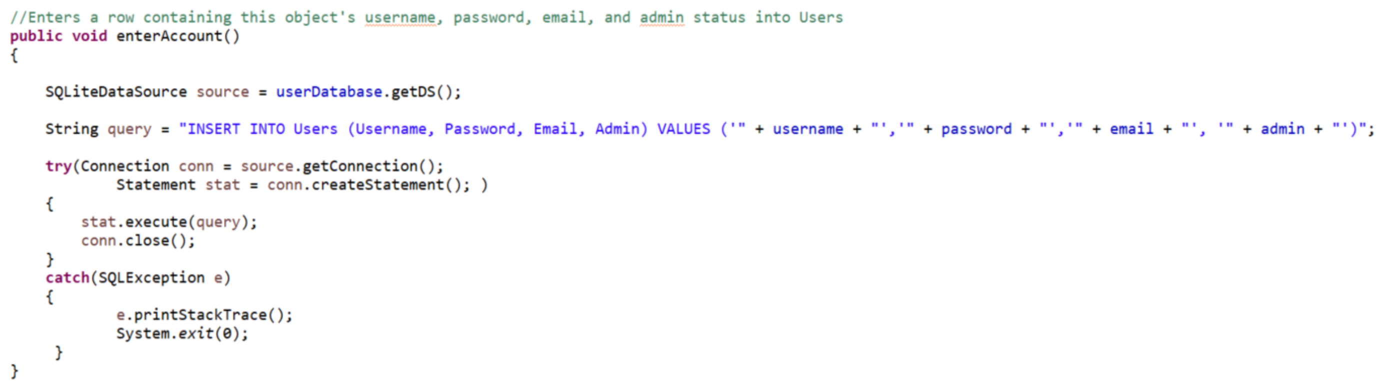
Information in databases is organized into tables. It is possible to search these tables for rows where certain criteria are fulfilled. In SQLite, searching is done via the SELECT command. The syntax is as follows: "SELECT * FROM table WHERE condition" (SQLite). This query returns the data from "table" from every row where "condition" is true in the form of a ResultSet. If the ResultSet has no data, there are no values where "condition" is true. This is best shown in the existsInDatabase() method of the Account class.



```
//Searches User table for a row where the username, password, and email match this object's username, password, and email
//Returns true if said row exists, and false if it does not
public boolean existsInDatabase()
{
    boolean result = false;

    SQLiteDataSource source = userDatabase.getDS();

    String query = "SELECT * FROM Users WHERE Username='" + username + "' AND Password ='" + password + "' AND Email='" + email + "'";
    System.out.println(query);

    try(Connection conn = source.getConnection();
            Statement stat = conn.createStatement(); )
    {
        ResultSet rs = stat.executeQuery(query);
        if(rs.next())
            result = true;
        conn.close();
    }
    catch(SQLException e)
    {
        e.printStackTrace();
        System.exit(0);
    }
```

## Inserting From a Database

Databases can have information added into them by the user.  The SQLite command for inserting is called INSERT, and its syntax is "INSERT into table (column names) VALUES (values to be inserted)" (SQLite). This query inserts a row in "table" with the values of "values to be inserted" being put into each column in "column names", as shown with the Account class's enterAccount() method.



```
//Enters a row containing this object's username, password, email, and admin status into Users
public void enterAccount()
{

    SQLiteDataSource source = userDatabase.getDS();

    String query = "INSERT INTO Users (Username, Password, Email, Admin) VALUES ('" + username + "','" + password + "','" + email + "', '" + admin + "')";

    try(Connection conn = source.getConnection();
            Statement stat = conn.createStatement(); )
    {
        stat.execute(query);
        conn.close();
    }
    catch(SQLException e)
    {
            e.printStackTrace();
            System.exit(0);
     }
}
```

## **Updating a Database**

Data already existing in a table can be updated. In SQLite, the command is UPDATE, and the syntax is “UPDATE table SET column = value WHERE condition” (SQLite). This query finds a row in “table” where “condition” is true, then sets the value of the “column” in that row to whatever “value” is. This is shown in the Student class’s updateMyYear(int year, int studentID) method.

```
//updates Students table so that row where ID = studentID has its Year value set to the new value of myYear
public void updateMyYear(int year, int studentID) {
    myYear = year;
    SQLiteDataSource source = myStudentBase.getDS();
    String query = "UPDATE Students SET Year = '" + myYear + "' WHERE ID=" + studentID;
    try(Connection conn = source.getConnection();
        Statement stat = conn.createStatement(); )
    {
        stat.execute(query);
        conn.close();
    }
    catch(SQLException e1)
    {
        e1.printStackTrace();
        System.exit(0);
    }
}
```

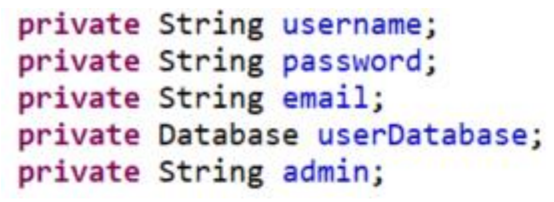
## **Deleting from a Database**

Data existing in a table can be deleted. In SQLite, the command is DELETE, and its syntax is “DELETE FROM table WHERE condition”. This query finds a row in “table” where “condition” is true, then deletes that row. This is shown in the Student class’s deleteStudent(int id) method.

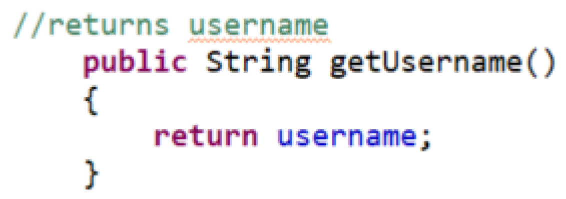
```
//deletes row from Student where ID=id
public void deleteStudent(int id)
{
    SQLiteDataSource source = myStudentBase.getDS();

    String query = "DELETE FROM Students WHERE ID=" + id;
    System.out.println(query);

    try(Connection conn = source.getConnection();
            Statement stat = conn.createStatement(); )
    {
        stat.execute(query);
        conn.close();
    }
    catch(SQLException e)
    {
        e.printStackTrace();
        System.exit(0);
    }
    System.out.print("deleted student");

}
```

## Encapsulation

The Account, Event, and Student classes are encapsulated, meaning that they have private variables that can only be accessed through get() and set() methods. For example, in Account, the username is private and can only be accessed by the getUsername() method.



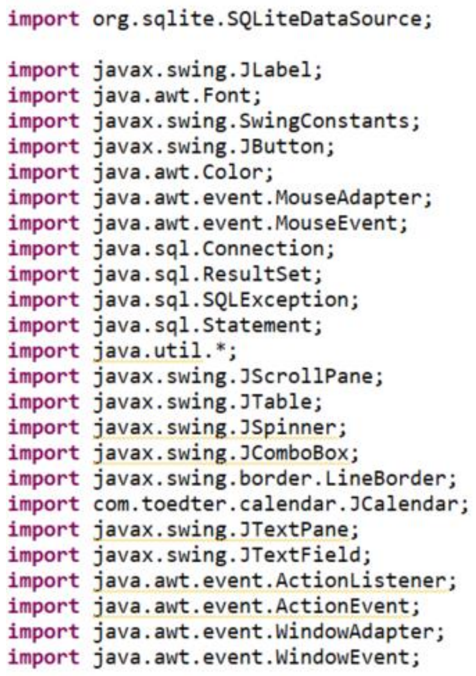
```
private String username;
private String password;
private String email;
private Database userDatabase;
private String admin;
```



```
//returns username
    public String getUsername()
    {
        return username;
    }
```

## External Libraries

There are various external libraries imported in order to make this program more efficient.

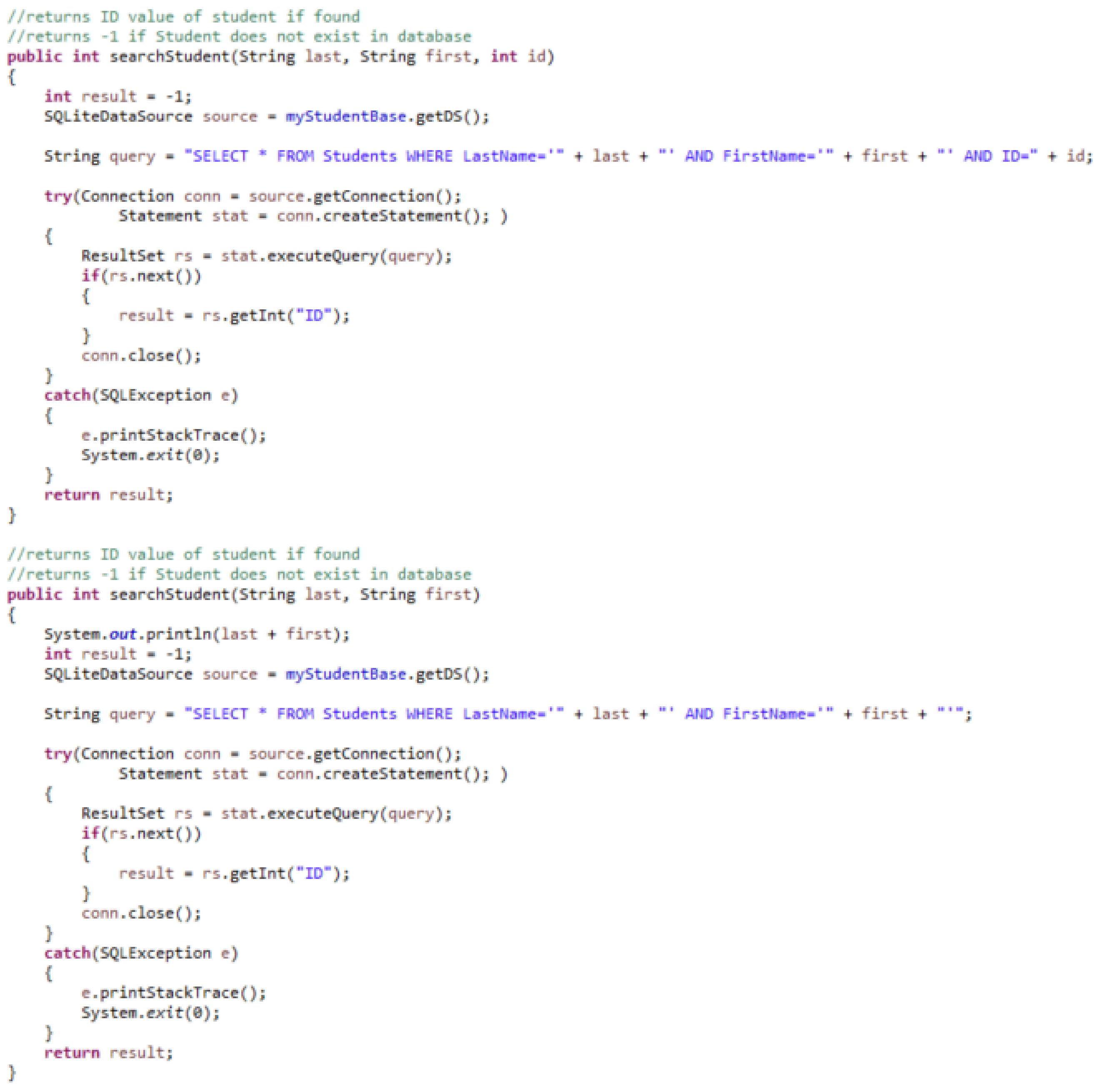


```
import org.sqlite.SQLiteDataSource;

import javax.swing.JLabel;
import java.awt.Font;
import javax.swing.SwingConstants;
import javax.swing.JButton;
import java.awt.Color;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.*;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JSpinner;
import javax.swing.JComboBox;
import javax.swing.border.LineBorder;
import com.toedter.calendar.JCalendar;
import javax.swing.JTextPane;
import javax.swing.JTextField;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
```

Take these imports from the AdminViewCalendar. The java.swing libraries allow the user to interact with the graphical elements of the program, like the JButtons and JScrollPane. The java.util library gives access to ArrayLists. The java.sql gives the program access to SQLite, and is also used to connect to the Jap-aneseClub database. Finally, in both AdminViewCalendar and RegularUserCalendar, com.toedter.calen-dar.JCalendar gives access to the JCalendar, which creates a Graphical User Interface where users can select dates (Toedter).

## Polymorphism – Overloading

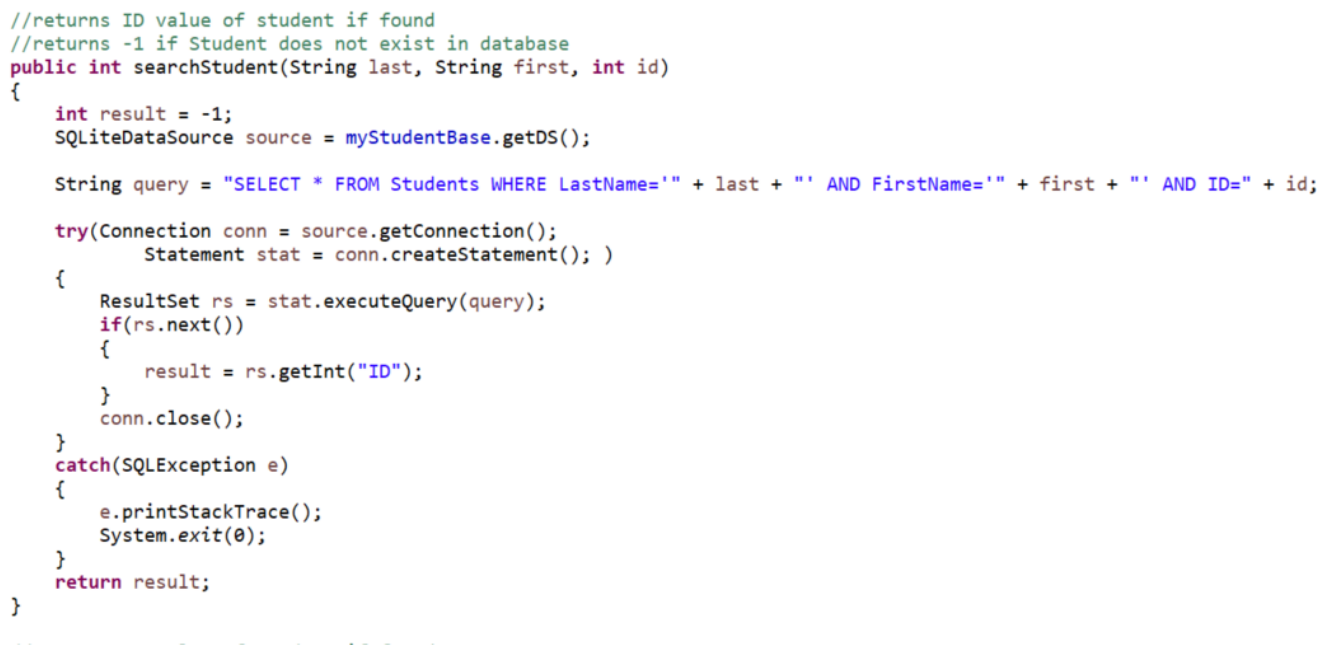
Polymorphism exists in my code through overloading. In the Student class, there are two searchStudentMethods: searchStudent(String last, String first, int id) and serachStudent(String last, String first).


```
//returns ID value of student if found
//returns -1 if Student does not exist in database
public int searchStudent(String last, String first, int id)
{
    int result = -1;
    SQLiteDataSource source = myStudentBase.getDS();

    String query = "SELECT * FROM Students WHERE LastName='" + last + "' AND FirstName='" + first + "' AND ID=" + id;

    try(Connection conn = source.getConnection();
            Statement stat = conn.createStatement(); )
    {
        ResultSet rs = stat.executeQuery(query);
        if(rs.next())
        {
            result = rs.getInt("ID");
        }
        conn.close();
    }
    catch(SQLException e)
    {
        e.printStackTrace();
        System.exit(0);
    }
    return result;
}

//returns ID value of student if found
//returns -1 if Student does not exist in database
public int searchStudent(String last, String first)
{
    System.out.println(last + first);
    int result = -1;
    SQLiteDataSource source = myStudentBase.getDS();

    String query = "SELECT * FROM Students WHERE LastName='" + last + "' AND FirstName='" + first + "'";

    try(Connection conn = source.getConnection();
            Statement stat = conn.createStatement(); )
    {
        ResultSet rs = stat.executeQuery(query);
        if(rs.next())
        {
            result = rs.getInt("ID");
        }
        conn.close();
    }
    catch(SQLException e)
    {
        e.printStackTrace();
        System.exit(0);
    }
    return result;
}
```

These two methods have different signatures, so despite both both searching the Students table, they do so in different ways.

## Try/Catch Block

Whenever an SQLite query attempts to be executed, a Try/Catch block is used to ensure that only valid queries are sent to the database. If the database attempted to execute a non-valid query, the program would crash.



```
//returns ID value of student if found
//returns -1 if Student does not exist in database
public int searchStudent(String last, String first, int id)
{
    int result = -1;
    SQLiteDataSource source = myStudentBase.getDS();

    String query = "SELECT * FROM Students WHERE LastName='" + last + "' AND FirstName='" + first + "' AND ID=" + id;

    try(Connection conn = source.getConnection();
            Statement stat = conn.createStatement(); )
    {
        ResultSet rs = stat.executeQuery(query);
        if(rs.next())
        {
            result = rs.getInt("ID");
        }
        conn.close();
    }
    catch(SQLException e)
    {
        e.printStackTrace();
        System.exit(0);
    }
    return result;
}
```

## Referenced Code

Hipp, R. D. 2020. *SQLite*. https://www.sqlite.org/index.html

McDonald, Shane. “How to set up SQLite with JDBC in Eclipse on Windows”. *Shane’s Computer Solution Repository*. 24 January 2020. https://shanemcd.org/2020/01/24/how-to-set-up-sqlite-with-jdbc-in-eclipse-on-windows/

Toedter, Kai. 11 July 2011. *JCalendar.* https://toedter.com/jcalendar/