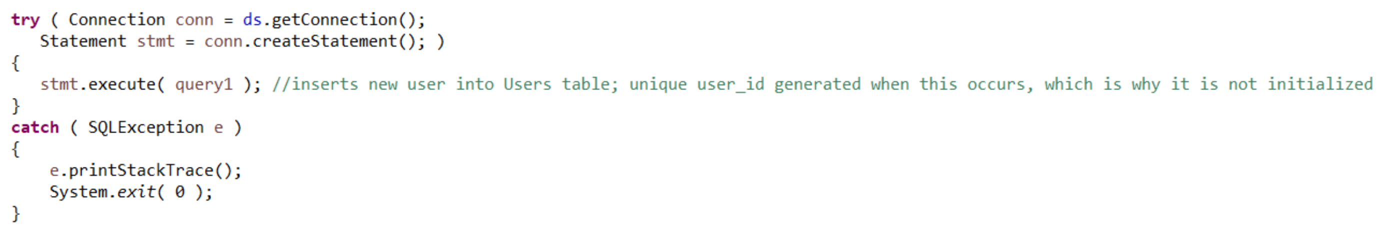
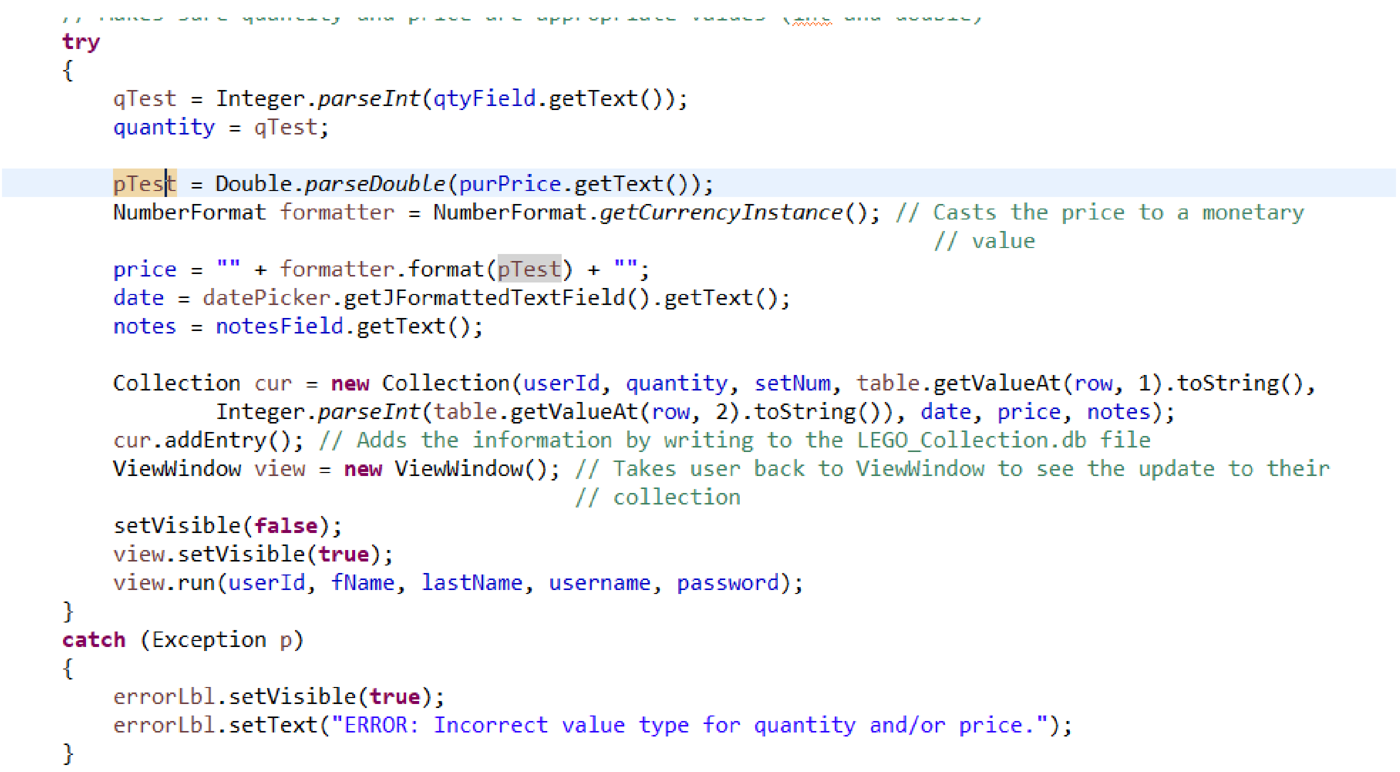
Appendix C – Development

1. Error Handling
    a. Try/catch



```
try ( Connection conn = ds.getConnection();
    Statement stmt = conn.createStatement(); )
{
    stmt.execute( query1 ); //inserts new user into Users table; unique user_id generated when this occurs, which is why it is not initialized
}
catch ( SQLException e )
{
     e.printStackTrace();
     System.exit( 0 );
}
```

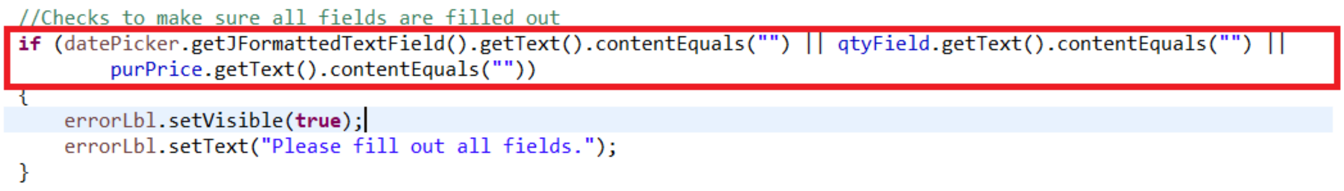
Try/catch blocks are vital to running my program. They check if the database connection has been established correctly, and throw errors otherwise.



```
// Makes sure quantity and price are appropriate values (int and double)
try
{
    qTest = Integer.parseInt(qtyField.getText());
    quantity = qTest;

    pTest = Double.parseDouble(purPrice.getText());
    NumberFormat formatter = NumberFormat.getCurrencyInstance(); // Casts the price to a monetary
                                                                 // value
    price = "" + formatter.format(pTest) + "";
    date = datePicker.getJFormattedTextField().getText();
    notes = notesField.getText();

    Collection cur = new Collection(userId, quantity, setNum, table.getValueAt(row, 1).toString(),
            Integer.parseInt(table.getValueAt(row, 2).toString()), date, price, notes);
    cur.addEntry(); // Adds the information by writing to the LEGO_Collection.db file
    ViewWindow view = new ViewWindow(); // Takes user back to ViewWindow to see the update to their
                                        // collection
    setVisible(false);
    view.setVisible(true);
    view.run(userId, fName, lastName, username, password);
}
catch (Exception p)
{
    errorLbl.setVisible(true);
    errorLbl.setText("ERROR: Incorrect value type for quantity and/or price.");
}
```

In addition, try/catch blocks also prevent the user from entering invalid values for the quantity and price whenever a user adds or edits a set. They check to make sure the values entered are appropriate by casting them to numerical data types (ints and doubles); if the program cannot, then it produces an error. Both try/catch blocks allow the program to keep running, even with errors.

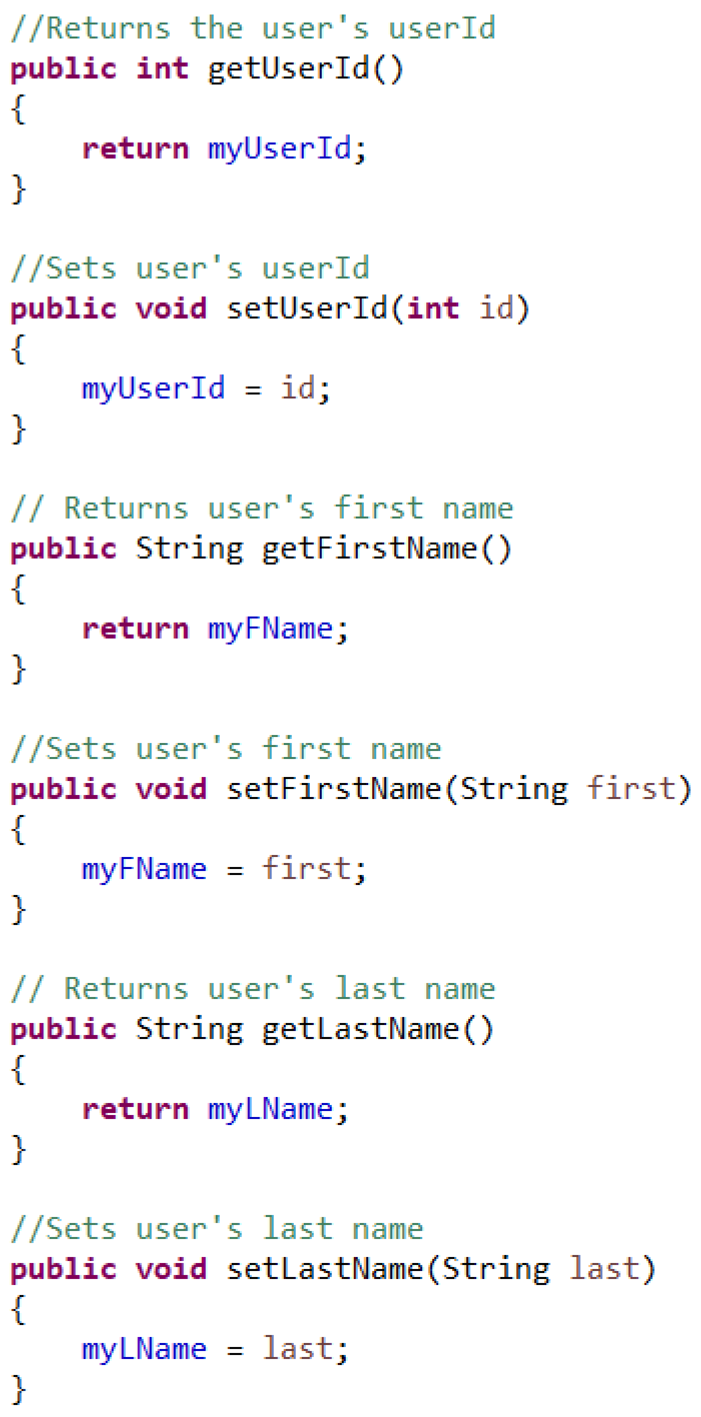
b. Handling Null Values



```
//Checks to make sure all fields are filled out
if (datePicker.getJFormattedTextField().getText().contentEquals("") || qtyField.getText().contentEquals("") ||
        purPrice.getText().contentEquals(""))
{
    errorLbl.setVisible(true);
    errorLbl.setText("Please fill out all fields.");
}
```

When adding or editing set entries, the program checks if any JTextFields or JFormattedTextFields (for the date) are null, and displays an error if so.

2. Encapsulation



```
//Returns the user's userId
public int getUserId()
{
    return myUserId;
}

//Sets user's userId
public void setUserId(int id)
{
    myUserId = id;
}

// Returns user's first name
public String getFirstName()
{
    return myFName;
}

//Sets user's first name
public void setFirstName(String first)
{
    myFName = first;
}

// Returns user's last name
public String getLastName()
{
    return myLName;
}

//Sets user's last name
public void setLastName(String last)
{
    myLName = last;
}
```

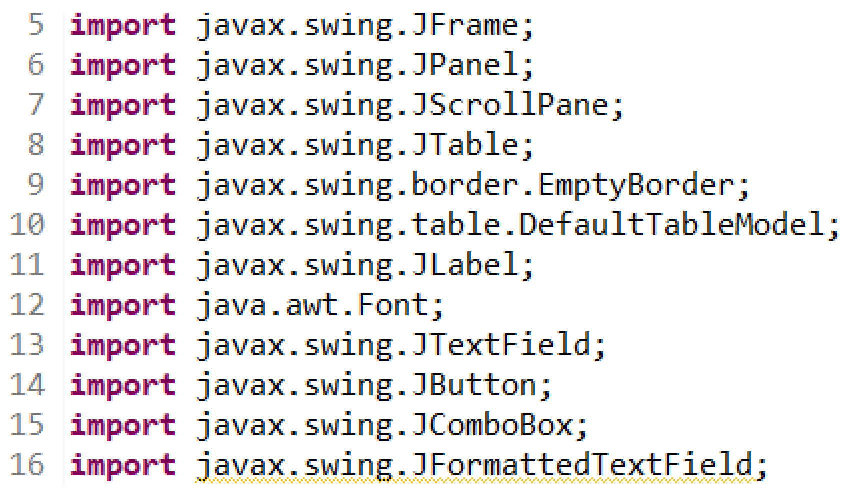
The Users, Collection, and Database classes utilize encapsulation.

For the Users class, the only way the private variables can only be accessed via accessor methods. In this case, these accessor methods are get() and set() methods for each variable within the class. In addition, new Users can only be added to the Users table in the database via a call to addUser().

For the Collection class, the only way to modify any user's collection is through calls to the accessor methods addEntry(), editEntry(), or deleteEntry(). These methods create, update, or delete set information from the user’s collection, effectively serving as set() methods.

For the Database class, the database private variable can only be initialized externally via a call to the set() method within the Database class.

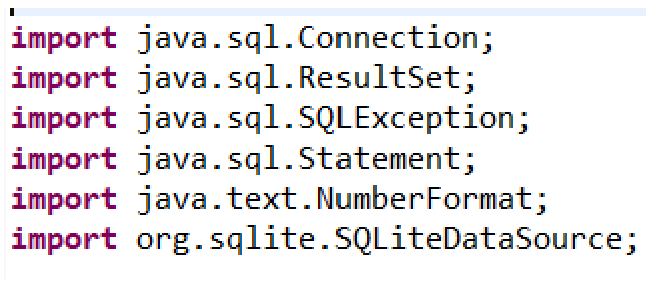
3. External Libraries
    a. JSwing



```java
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.border.EmptyBorder;
import javax.swing.table.DefaultTableModel;
import javax.swing.JLabel;
import java.awt.Font;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFormattedTextField;
```

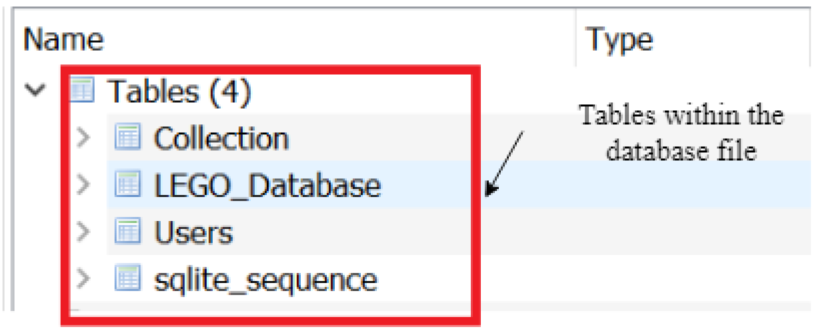
The code implements external libraries to utilize premade methods and algorithms. In the image above, a sample of the imported libraries are displayed. These particular libraries are used to create an intuitive GUI that enhances the program's functionality. The java.swing classes allow for the implementation of various graphical interface features, such as JTables and JButtons. These graphical interface features allow the user to easily understand the program's flow and functionality, and follow the logical progression of events.

b. JDBC Driver for SQLite



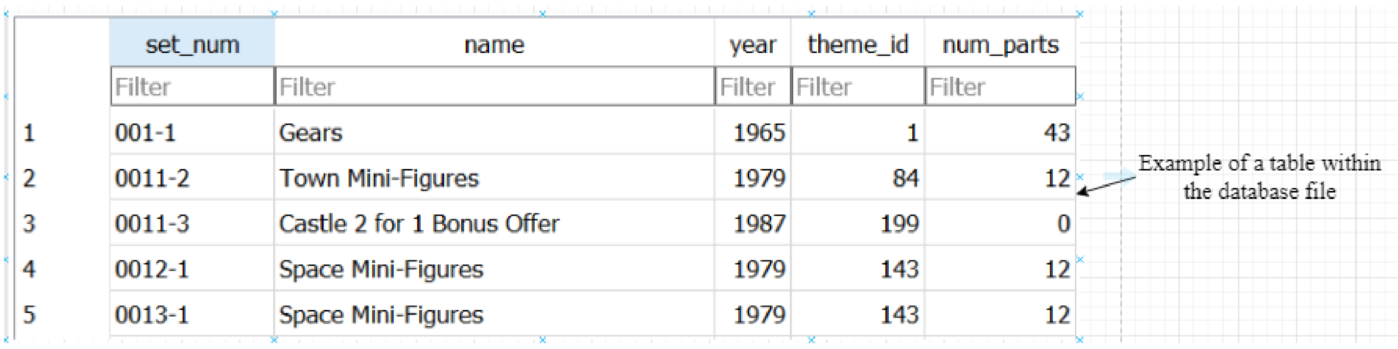
```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.text.NumberFormat;
import org.sqlite.SQLiteDataSource;
```

The JDBC Driver for SQLite allows for the connection to the DB Browser for SQLite from Eclipse. SQLite is a relational database software that stores multiple tables' worth of information within a single database file. In this database file, each group of information that would necessitate its own file (i.e. login information, the LEGO set catalogue, etc.) would be stored in separate tables within that database. It makes the program intuitive because it allows me to store information that would normally necessitate the existence of multiple files in one location. In addition, SQLite can read, write, delete, and filter through information within a database file. The software can also specify which user a set belongs to using the primary key function, which automatically assigns a value to each user in the database file. This makes the program more code efficient because I do not have to create my own methods for all of these processes; rather, I can just take advantage of SQLite's innate ability to handle them.



Name | Type

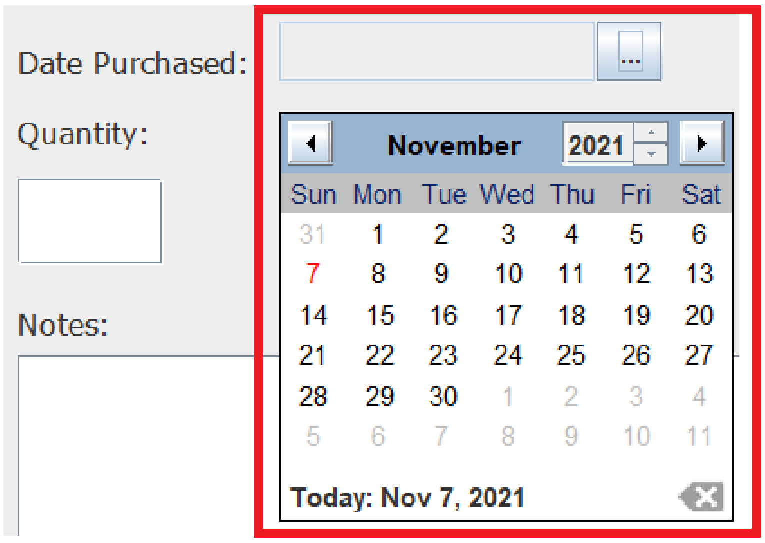
- Tables (4)
  - Collection
  - LEGO_Database
  - Users
  - sqlite_sequence



	set_num	name	year	theme_id	num_parts
	Filter	Filter	Filter	Filter	Filter
1	001-1	Gears	1965	1	43
2	0011-2	Town Mini-Figures	1979	84	12
3	0011-3	Castle 2 for 1 Bonus Offer	1987	199	0
4	0012-1	Space Mini-Figures	1979	143	12
5	0013-1	Space Mini-Figures	1979	143	12

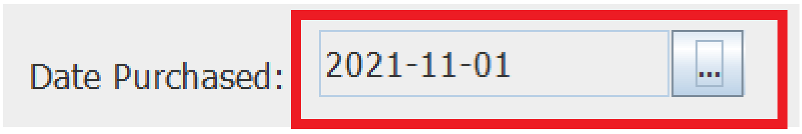
c. datePicker

```
//Initializes a datePicker
//Which lets the user choose the date they purchased a set from an
//Interactive calendar GUI
UtilDateModel dModel = new UtilDateModel();
Properties p = new Properties();
p.put("text.today", "Today");
p.put("text.month", "Month");
p.put("text.year", "Year");
JDatePanelImpl datePanel = new JDatePanelImpl(dModel, p);
JDatePickerImpl datePicker = new JDatePickerImpl(datePanel, new DateLabelFormatter());
datePicker.getJFormattedTextField().setFont(new Font("Tahoma", Font.PLAIN, 14));
datePicker.setBounds(214, 376, 170, 40);
contentPane.add(datePicker);
```



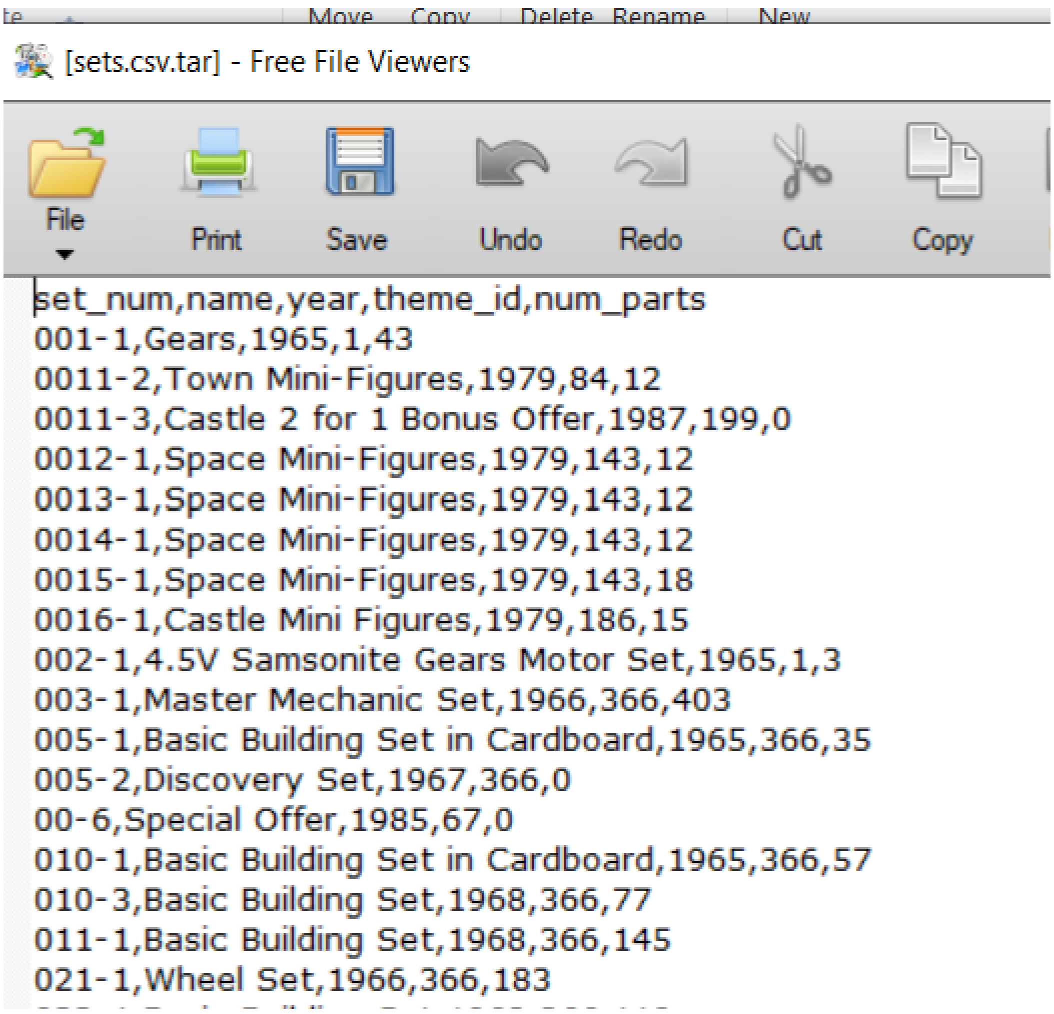
The datePicker library is a third-party GUI which allows the user to pick any date where they purchased a set from a premade calendar GUI, and then displays that date as a String in the “yyyy-mm-dd” format.

d. JFormattedTextField


Date Purchased: 2021-11-01

The program utilizes a JFormattedTextField for the date. The JFormattedTextField restricts the information that can be entered into a text field. In this particular instance, the JFormattedTextField class will only store the date in the "yyyy-mm-dd" format. The date is determined by using the datePicker widget.

e. sets.csv



```
set_num,name,year,theme_id,num_parts
001-1,Gears,1965,1,43
0011-2,Town Mini-Figures,1979,84,12
0011-3,Castle 2 for 1 Bonus Offer,1987,199,0
0012-1,Space Mini-Figures,1979,143,12
0013-1,Space Mini-Figures,1979,143,12
0014-1,Space Mini-Figures,1979,143,12
0015-1,Space Mini-Figures,1979,143,18
0016-1,Castle Mini Figures,1979,186,15
002-1,4.5V Samsonite Gears Motor Set,1965,1,3
003-1,Master Mechanic Set,1966,366,403
005-1,Basic Building Set in Cardboard,1965,366,35
005-2,Discovery Set,1967,366,0
00-6,Special Offer,1985,67,0
010-1,Basic Building Set in Cardboard,1965,366,57
010-3,Basic Building Set,1968,366,77
011-1,Basic Building Set,1968,366,145
021-1,Wheel Set,1966,366,183
```

I imported the LEGO set catalogue from Rebrickable.com, where it was saved as a .csv file. I then imported this file into SQLite, where it could be accessed whenever the user needed to add a set to their collection. This made the program more intuitive because it eliminated the need for me to create my own LEGO set catalogue file, as I could just take use a pre-existing one. In addition, I could just have SQLite handle any processes where the user needed to read information from the LEGO set catalogue, instead of making my own methods for it.

4. File Access

a. Reading from a Database File

```
String query = "SELECT quantity, set_num, name, year, pur_date, price, notes FROM Collection WHERE user_id = "
        + userId + " ORDER BY " + selectedSort + " ASC";
SQLiteDataSource source = collection.getDs();

try (Connection conn = source.getConnection(); Statement stmt = conn.createStatement();)
{

    ResultSet rs = stmt.executeQuery(query);
    while (rs.next())
    {
        model.insertRow(model.getRowCount(),
                new String[] { rs.getString("quantity"), rs.getString("set_num"), rs.getString("name"),
                        rs.getString("year"), rs.getString("pur_date"), rs.getString("price"),
                        rs.getString("notes") });
    }
}
catch (SQLException e)
{
    e.printStackTrace();
    System.exit(0);
}
```

My program utilized a query statement, to read from a database file. The query statement for reading from a database file is a String which consists of a SELECT statement, which specifies which columns in the table will be read, a FROM statement, which when used in conjunction with the table name specifies the table being read from, and a WHERE statement, which specifies the criteria for which information is being read. In the example above, only the quantity, set_num, name, year, pur_date, price, and notes columns in the Collection table are only being read from rows where the user_id is the specified userId String. The ORDER BY statement only specifies what information the information will be read in. This makes the program more intuitive because SQLite can handle all of the file reading using its own innate methods, instead of me having to write my own methods for it.

b. Writing to a Database File

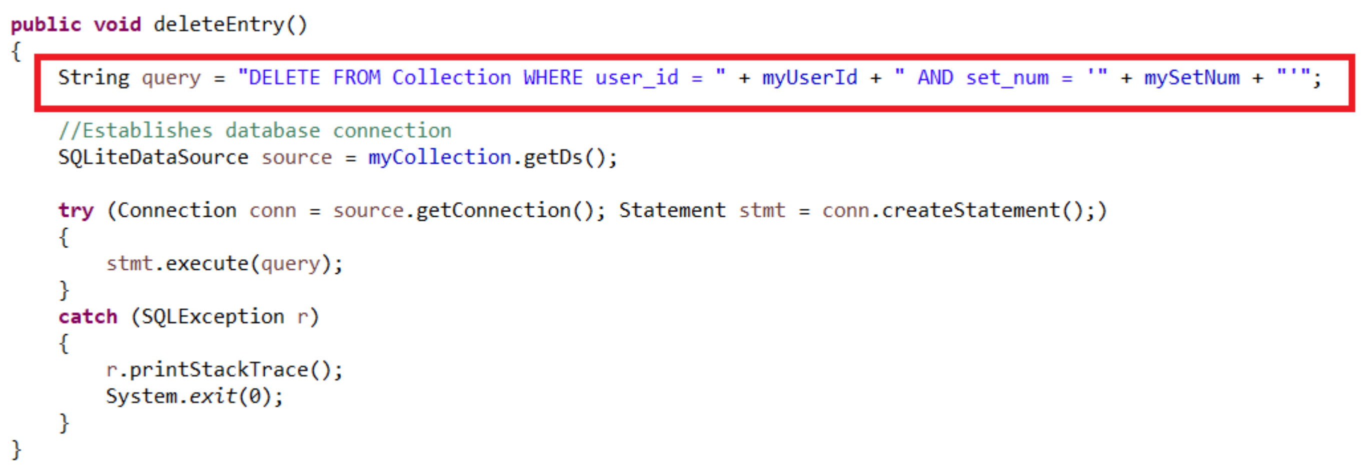
```
public void addEntry()
{
    String query = "INSERT INTO Collection(user_id, quantity, set_num, name, year, pur_date, price, notes) VALUES "
            + "(" + myUserId + ", " + myQuantity + ", '" + mySetNum + "', '" + myName + "', " + myYear + ", '"
            + myPurDate + "', '" + myPrice + "', '" + myNotes + "')";
    SQLiteDataSource source = myCollection.getDs();

    try (Connection conn = source.getConnection(); Statement stmt = conn.createStatement();) {
        stmt.execute(query);
        System.out.println(query);

    }
    catch (SQLException e)
    {
        e.printStackTrace();
        System.exit(0);
    }
}
```

My program utilized a query statement to write to a database file. For addEntry(), the query statement is a String which consists of a INSERT INTO statement, which specifies which table in the database file the information is being written to, and parameters that represent the columns which will be filled in when the information is inserted into the database. The parameters are filled in with the values in the parentheses after the VALUES statement. This information is then added to the database file in that particular table, with any columns with unspecified parameters being null.

editEntry() has a similar process to addEntry(), but instead of starting with the INSERT INTO statement, it starts with an UPDATE statement, followed by the table being updated, and a SET statement, which sets the columns to the new values.

This made the program more intuitive because SQLite handles all of the writing to the file, and automatically saves it.

c. Deleting from a Database File



```
public void deleteEntry()
{
    String query = "DELETE FROM Collection WHERE user_id = " + myUserId + " AND set_num = '" + mySetNum + "'";

    //Establishes database connection
    SQLiteDataSource source = myCollection.getDs();

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

My program utilized a query statement to delete information from the database. The query statement for deleteEntry() starts with a DELETE FROM statement, followed by the table in the database being deleted from, followed by a WHERE statement which establishes the conditions where the row will be deleted. In this case, rows in the Collection table with the specified user_id and set_num will be deleted. This makes the program more intuitive because SQLite handles the process for deleting from the database file, and automatically saves the edits made.

Word Count: 1,038

Works Cited

GeeksforGeeks. "SQL - ORDER BY." *GeeksforGeeks*, 4 Oct. 2021, www.geeksforgeeks.org/sql-order-by.

"How to Use Jdatepicker to Display Calendar Component." *CodeJava*, 5 July 2019, https://www.codejava.net/java-se/swing/how-to-use-jdatepicker-to-display-calendar-component.

"JDatePicker: Java Swing Date Picker." *SourceForge*, SourceForge, 19 Nov. 2019, https://source forge.net/projects/jdatepicker/.

"Learn SQLite UPDATE Statement with Examples." *SQLite Tutorial*, 11 Apr. 2020, www.sqlitetuto- rial.net/sqlite-update.

"Lego Database Downloads." *Rebrickable*, Rebrickable, https://rebrickable.com/downloads/.

Matsuoka, Cary and George Peck. Introduction to Computer Science with Java. Sunnyvale, California: Institute of Computer Technology (ICT).

"NumberFormat (Java Platform SE 7)." *Java^{TM} Platform, Standard Edition 7 API Specification*, Oracle, 24 June 2020, docs.oracle.com/javase/7/docs/api/java/text/NumberFormat.html#getCurren- cyInstance().

Shane. "How to Set up SQLite with JDBC in Eclipse on Windows." *Shane's Computer Solution Repository*, 25 Jan. 2020, shanemcd.org/2020/01/24/how-to-set-up-sqlite-with-jdbc-in-eclipse-on-windows.