Criteria C: Development

Introduction

To construct the application, I chose Netbeans as my IDE since it has various types of tools that Java Wing offered to create the Graphic User Interface (GUI).

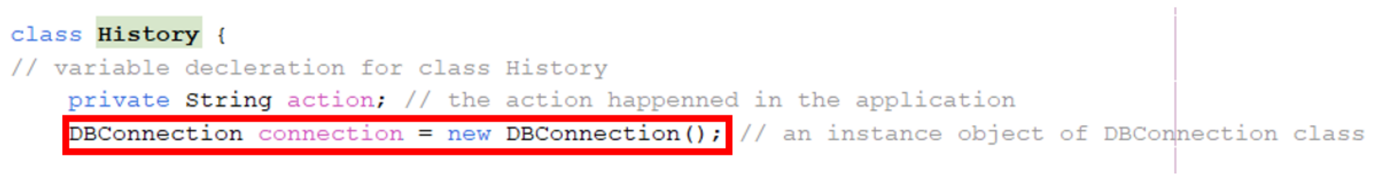
List of Techniques

Advanced techniques	Graphic User Interface (GUI)
	External Libraries
	Exception Handling
	Inclusion
	Polymorphism
	ArrayList Data Structure
	Encapsulation
	Database SQL Connection
	Hash Function
	Email Configuration
Basic techniques	If statements
	Loops
	Collection Data Structure

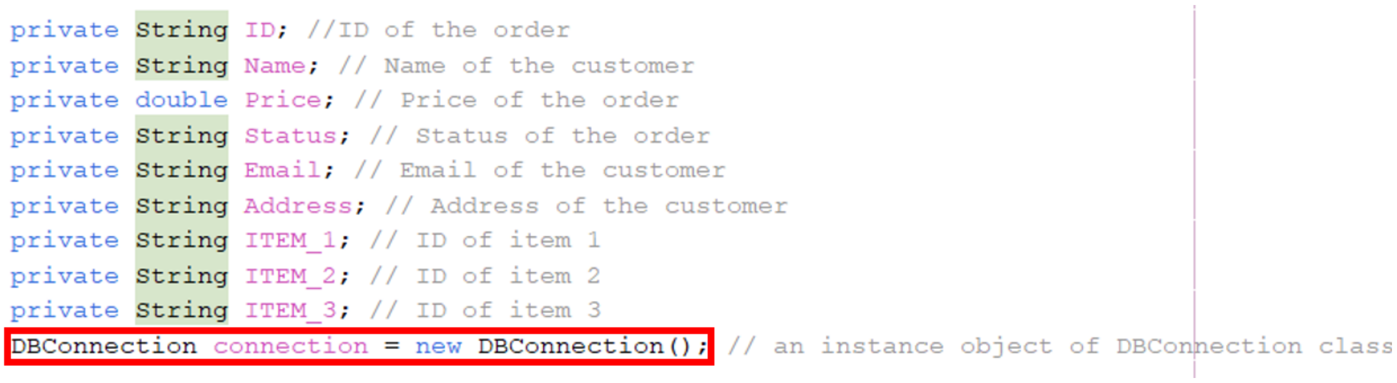
Code Analysis

Object Oriented Programming (OOP)

One of the benefits of OOP is the ability to split programs into many smaller components, hence lowering their complexity *(Gillis & Lewis, 2021)*. By using OOP, I can quickly divide my code into numerous classes, making it simpler to create and maintain the code. As a result, I can modify one class without making comparable changes to the others. For instance, the 5 controller classes—DBConnection, History, Orders, Storages, and USER—show the organization and simplification of my code.

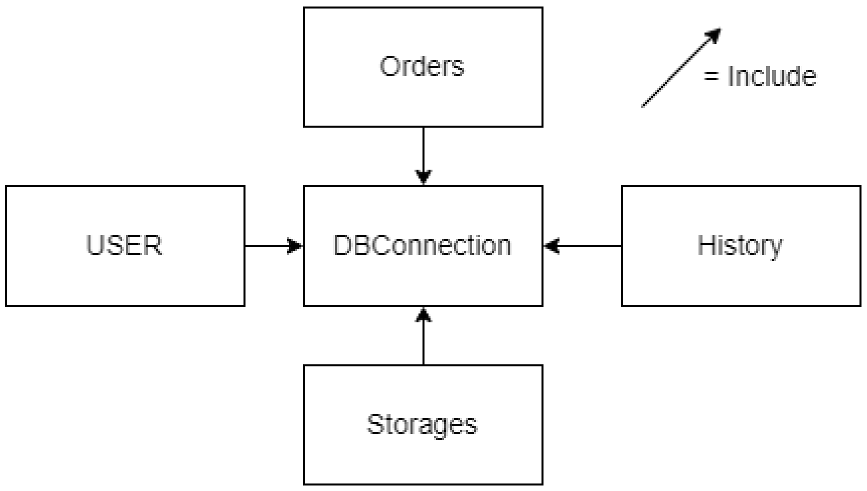


```
class History {
// variable decleration for class History
    private String action; // the action happenned in the application
    DBConnection connection = new DBConnection(); // an instance object of DBConnection class
```



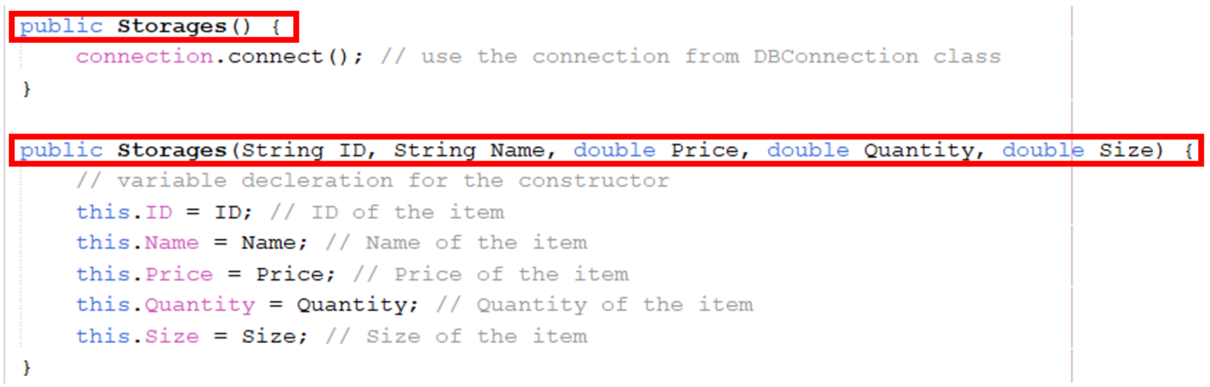
```
private String ID; //ID of the order
private String Name; // Name of the customer
private double Price; // Price of the order
private String Status; // Status of the order
private String Email; // Email of the customer
private String Address; // Address of the customer
private String ITEM_1; // ID of item 1
private String ITEM_2; // ID of item 2
private String ITEM_3; // ID of item 3
DBConnection connection = new DBConnection(); // an instance object of DBConnection class
```

In those 5 classes listed above, except for DBConnection, all have an instance object of DBConnetion class to connect to the database; in other words, those classes include the DBConnection class. This can be seen from the two examples above (red boxes). This **inclusion** creates a class hierarchy that looks like:



Every class in the aforementioned hierarchy contains the DBConnection class. In this instance, I used inclusion to use the *connect()* method found in the DBConnection class. I'll be able to connect to the database and make changes using this function from any class.

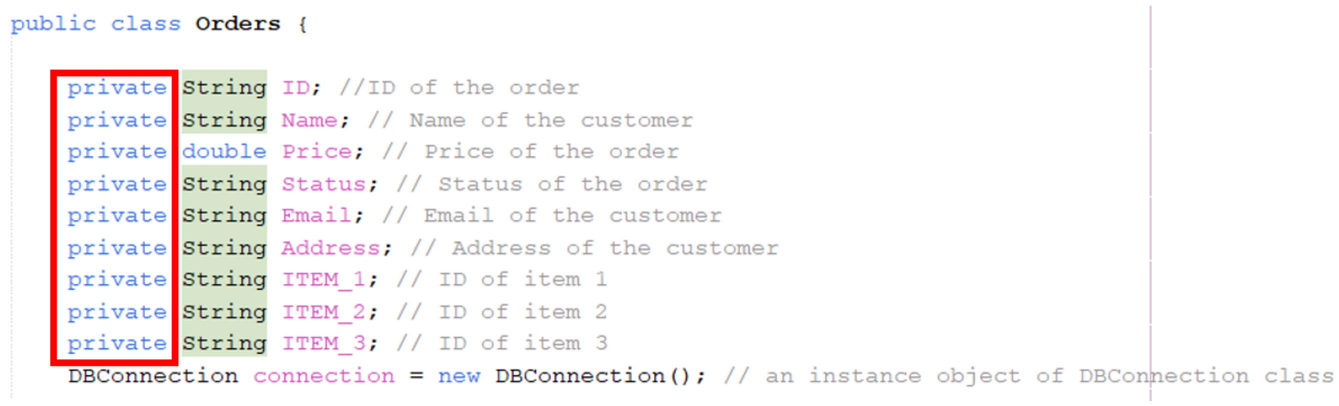
OOP also offers **polymorphism** features, for example:



```
public Storages() {
    connection.connect(); // use the connection from DBConnection class
}

public Storages(String ID, String Name, double Price, double Quantity, double Size) {
    // variable decleration for the constructor
    this.ID = ID; // ID of the item
    this.Name = Name; // Name of the item
    this.Price = Price; // Price of the item
    this.Quantity = Quantity; // Quantity of the item
    this.Size = Size; // Size of the item
}
```

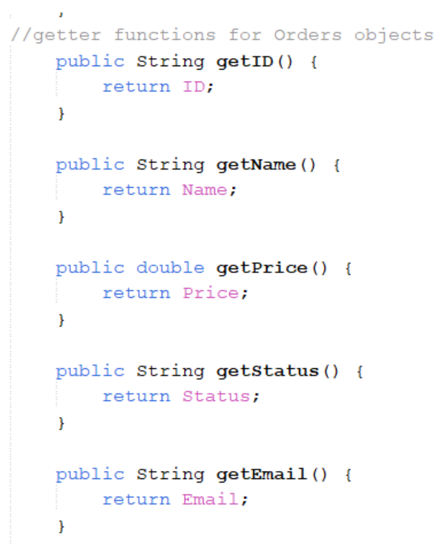
As indicated in the image above, in the Storages class, I have two constructors with the name "Storages," but they require different parameters to be called. This is known as overloading or static polymorphism *(Java Polymorphism, n.d.)*; it allows for reusability and reduces code complexity, making it easier to debug.

Another feature of OOP that I implemented in my program is **encapsulation.**



```
public class Orders {

    private String ID; //ID of the order
    private String Name; // Name of the customer
    private double Price; // Price of the order
    private String Status; // Status of the order
    private String Email; // Email of the customer
    private String Address; // Address of the customer
    private String ITEM_1; // ID of item 1
    private String ITEM_2; // ID of item 2
    private String ITEM_3; // ID of item 3
    DBConnection connection = new DBConnection(); // an instance object of DBConnection class
```

In the picture above, I declared several variables in the Orders class as private as an example. This feature improves software security by making variables unreachable from outside the class. Furthermore, encapsulation wraps the data into a single unit, making debugging easier because data may be updated inside without affecting outside elements *(Java - Encapsulation, n.d.)*.



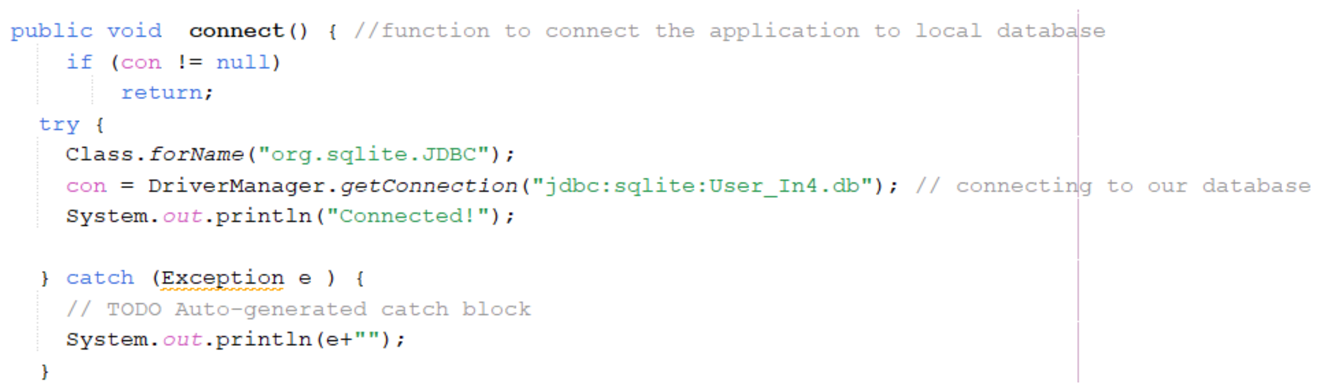
```
//getter functions for Orders objects
    public String getID() {
        return ID;
    }

    public String getName() {
        return Name;
    }

    public double getPrice() {
        return Price;
    }

    public String getStatus() {
        return Status;
    }

    public String getEmail() {
        return Email;
    }
```

Because all data updates in my software will be handled by queries in specific functions, setters methods for those encapsulated variables are unnecessary. However, in order to retrieve the required data, getter methods will be implemented (in the picture above).

Database SQLite

The program's connection to the **local SQLite database** was established using the **external SQLite JDBC library**. The SQLite JDBC library is where the code gets all of its queries from.

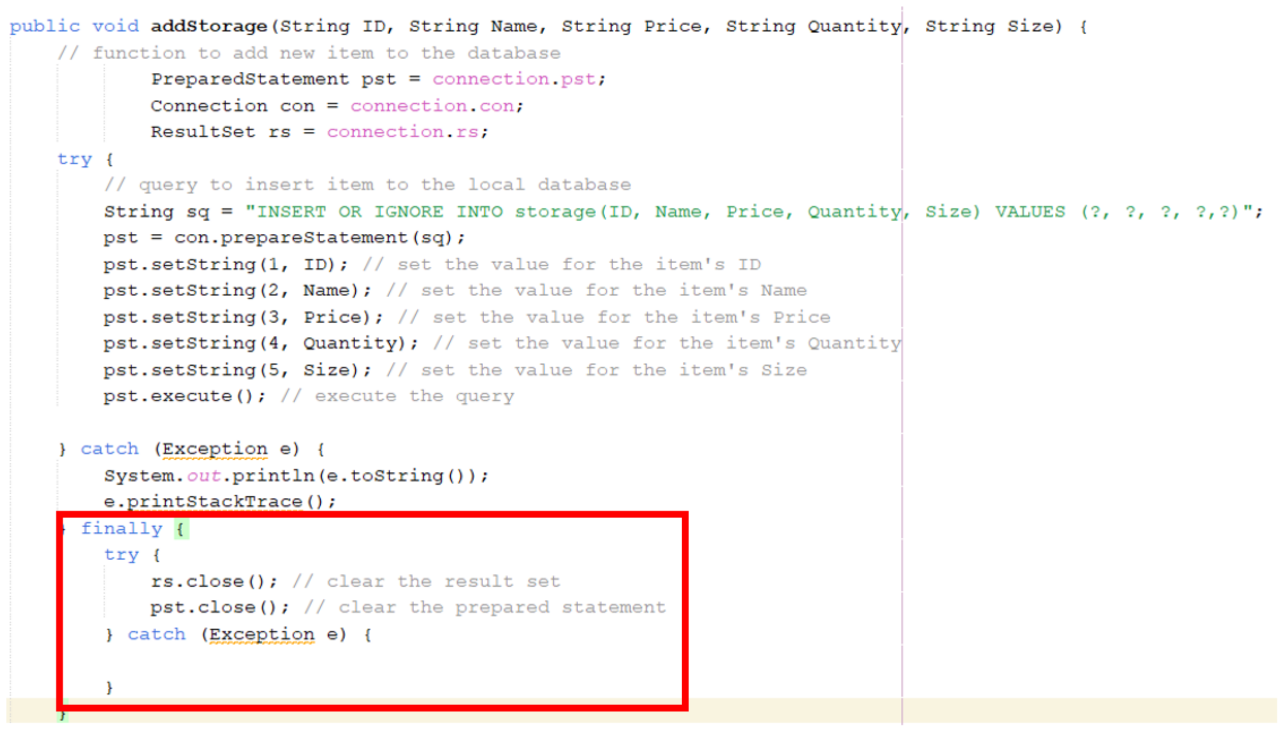
The *connect()* function, which establishes a connection to the database and is utilized throughout the application, was built into the DBConnection class.



```java
public void  connect() { //function to connect the application to local database
    if (con != null)
        return;
  try {
    Class.forName("org.sqlite.JDBC");
    con = DriverManager.getConnection("jdbc:sqlite:User_In4.db"); // connecting to our database
    System.out.println("Connected!");

  } catch (Exception e ) {
    // TODO Auto-generated catch block
    System.out.println(e+"");
  }
```

The *connect()* function, which connects the program to a local database that is present in the application's file, is seen in the image above.

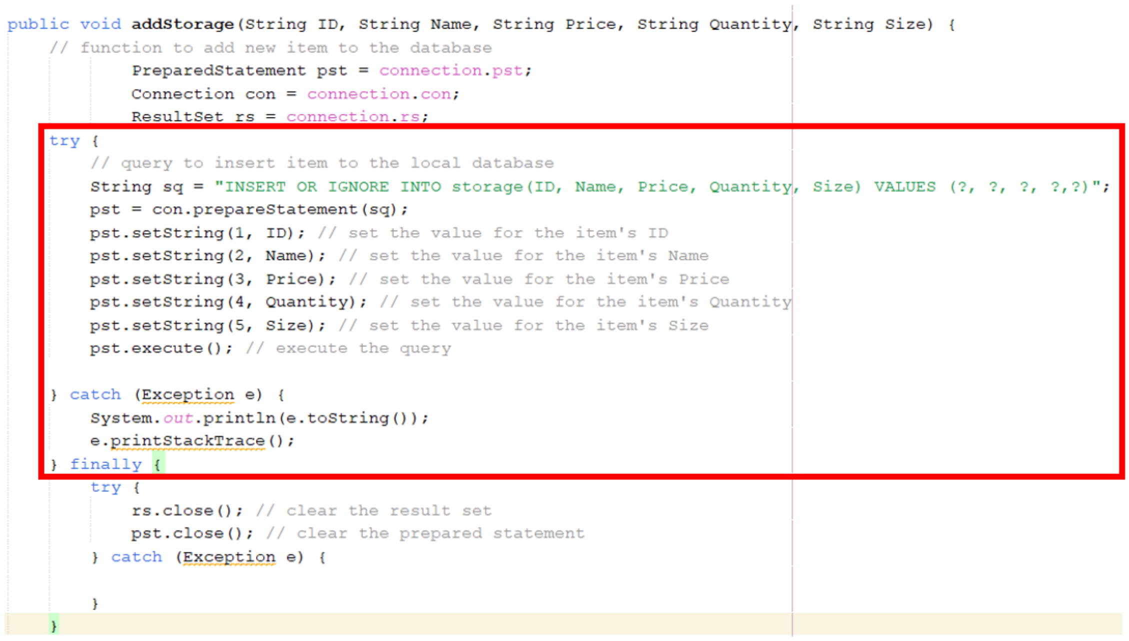
I have implemented the following technique to prevent crashes when several database changes are being made simultaneously (SQLite Exception: SQLite Busy) *(Understanding SQLITE_BUSY • ActiveSphere - Software Consulting, 2018)*:



```java
public void addStorage(String ID, String Name, String Price, String Quantity, String Size) {
    // function to add new item to the database
            PreparedStatement pst = connection.pst;
            Connection con = connection.con;
            ResultSet rs = connection.rs;
    try {
        // query to insert item to the local database
        String sq = "INSERT OR IGNORE INTO storage(ID, Name, Price, Quantity, Size) VALUES (?, ?, ?, ?,?)";
        pst = con.prepareStatement(sq);
        pst.setString(1, ID); // set the value for the item's ID
        pst.setString(2, Name); // set the value for the item's Name
        pst.setString(3, Price); // set the value for the item's Price
        pst.setString(4, Quantity); // set the value for the item's Quantity
        pst.setString(5, Size); // set the value for the item's Size
        pst.execute(); // execute the query

    } catch (Exception e) {
        System.out.println(e.toString());
        e.printStackTrace();
    } finally {
        try {
            rs.close(); // clear the result set
            pst.close(); // clear the prepared statement
        } catch (Exception e) {

        }
    }
```

As shown in the image above, I added an additional try-catch statement as an **exception handling** that would close the resultset (the **collection** output of the query) and the prepared statement (a feature used to pre-compile SQL code) in order to reduce database congestion if more access is performed after that.

By utilizing the connection to the database, I can conduct changes to the database by using different queries.

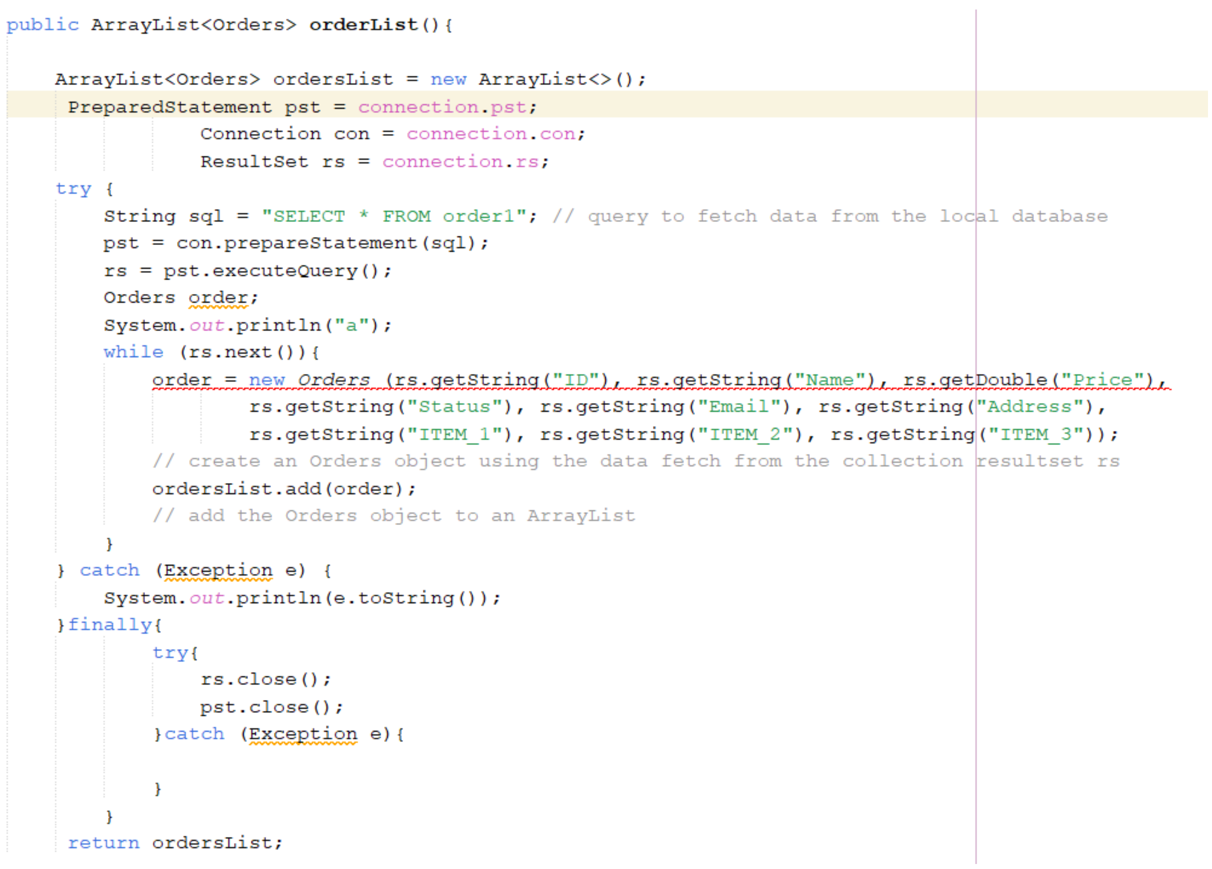

```
public void addStorage(String ID, String Name, String Price, String Quantity, String Size) {
    // function to add new item to the database
            PreparedStatement pst = connection.pst;
            Connection con = connection.con;
            ResultSet rs = connection.rs;
    try {
        // query to insert item to the local database
        String sq = "INSERT OR IGNORE INTO storage(ID, Name, Price, Quantity, Size) VALUES (?, ?, ?, ?,?)";
        pst = con.prepareStatement(sq);
        pst.setString(1, ID); // set the value for the item's ID
        pst.setString(2, Name); // set the value for the item's Name
        pst.setString(3, Price); // set the value for the item's Price
        pst.setString(4, Quantity); // set the value for the item's Quantity
        pst.setString(5, Size); // set the value for the item's Size
        pst.execute(); // execute the query

    } catch (Exception e) {
        System.out.println(e.toString());
        e.printStackTrace();
    } finally {
        try {
            rs.close(); // clear the result set
            pst.close(); // clear the prepared statement
        } catch (Exception e) {

        }
    }
```

The code in the red boxes, for example, uses a typical try-catch statement to modify the database. In this scenario, a query, which is a request for data or information from a database table or combination tables, is saved as a String (String sq in the code fraction above), and then executed by the prepared statement (pst in the code fraction above) *(Processing SQL Statements With JDBC (the Java^{TM} Tutorials > JDBC Database Access > JDBC Basics), n.d.)*. Many different types of queries were utilized in the software (as listed in Criterion B, Proposed Queries), but their execution was generally identical.

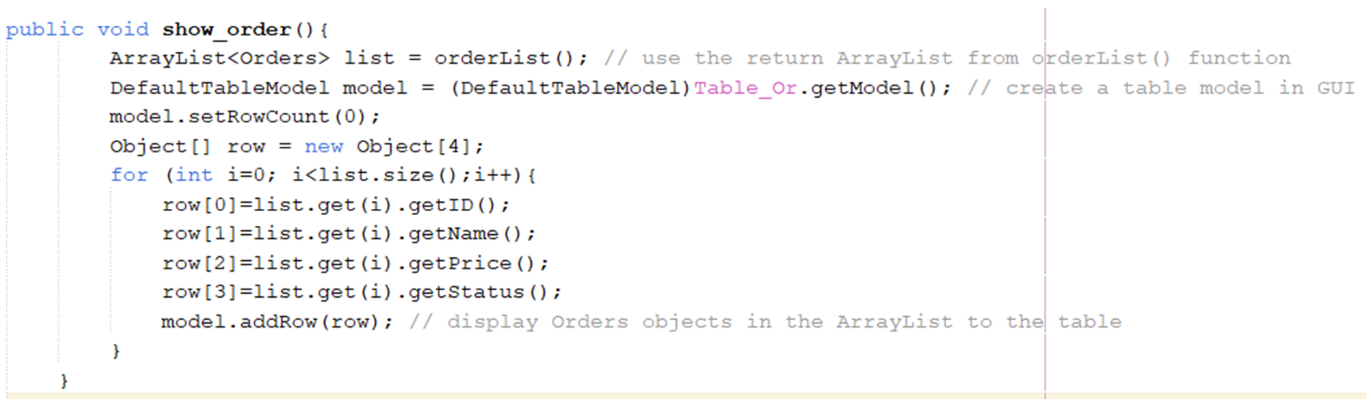
GUI

Because of the application's property, tables in GUI play an important role. I use **Java GUI Swing's Table Model** functionality to construct numerous tables that display database orders and storage data.

I used a mix of the two functions to retrieve data from one of the database tables and display it. To display a table of all orders from the database, for example, we have:



```
public ArrayList<Orders> orderList(){

    ArrayList<Orders> ordersList = new ArrayList<>();
     PreparedStatement pst = connection.pst;
                Connection con = connection.con;
                ResultSet rs = connection.rs;
    try {
        String sql = "SELECT * FROM order1"; // query to fetch data from the local database
        pst = con.prepareStatement(sql);
        rs = pst.executeQuery();
        Orders order;
        System.out.println("a");
        while (rs.next()){
            order = new Orders (rs.getString("ID"), rs.getString("Name"), rs.getDouble("Price"),
                    rs.getString("Status"), rs.getString("Email"), rs.getString("Address"),
                    rs.getString("ITEM_1"), rs.getString("ITEM_2"), rs.getString("ITEM_3"));
            // create an Orders object using the data fetch from the collection resultset rs
            ordersList.add(order);
            // add the Orders object to an ArrayList
        }
    } catch (Exception e) {
        System.out.println(e.toString());
    }finally{
            try{
                rs.close();
                pst.close();
            }catch (Exception e){

            }
        }
     return ordersList;
```

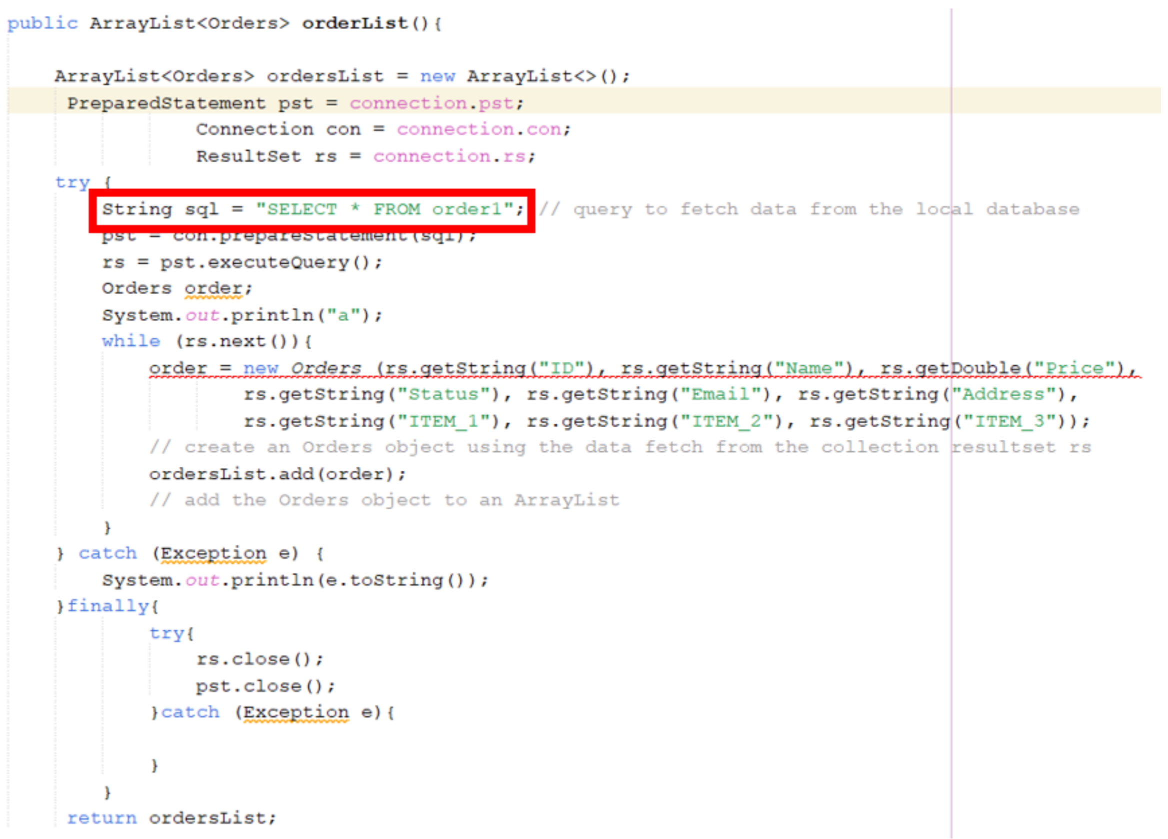


```
public void show_order(){
        ArrayList<Orders> list = orderList(); // use the return ArrayList from orderList() function
        DefaultTableModel model = (DefaultTableModel)Table_Or.getModel(); // create a table model in GUI
        model.setRowCount(0);
        Object[] row = new Object[4];
        for (int i=0; i<list.size();i++){
            row[0]=list.get(i).getID();
            row[1]=list.get(i).getName();
            row[2]=list.get(i).getPrice();
            row[3]=list.get(i).getStatus();
            model.addRow(row); // display Orders objects in the ArrayList to the table
        }
    }
```

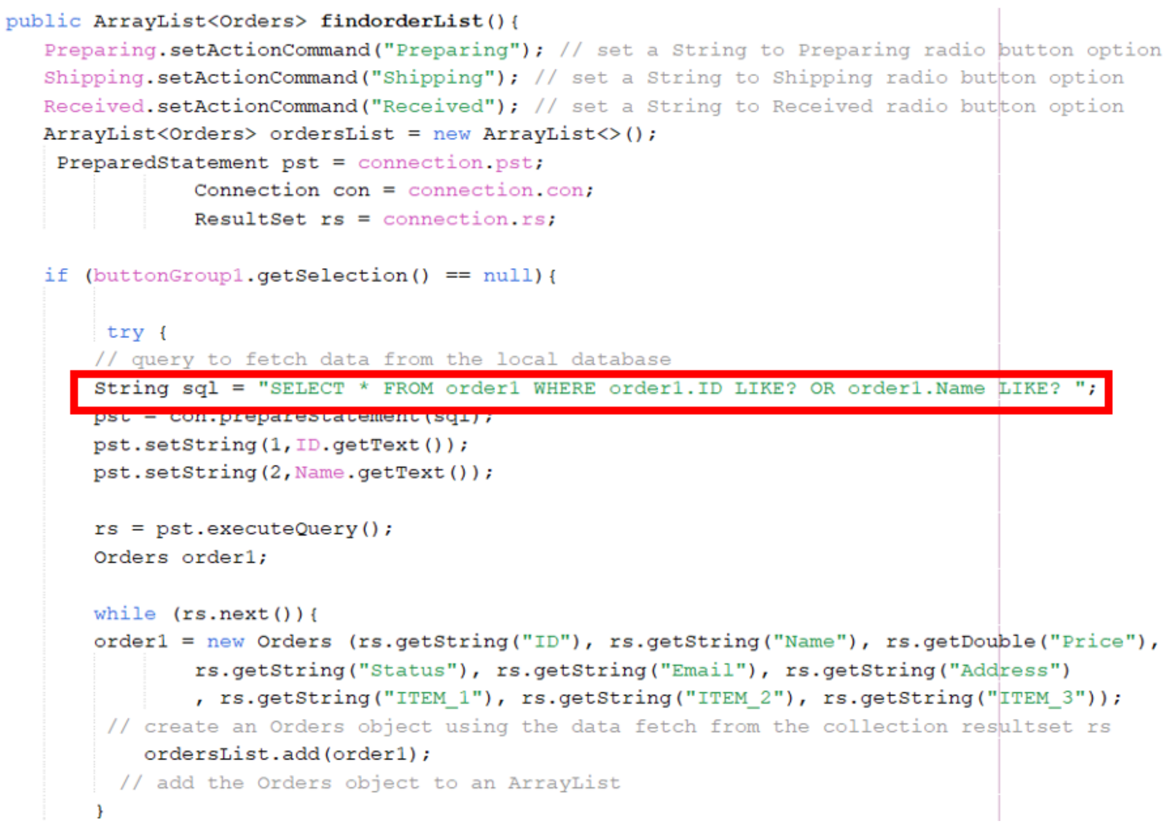
The first function, *orderList()*, utilizes a query to retrieve data from the specified database table. It then creates instances of the objects with the associated data from the result set. The elements will subsequently be placed in an **ArrayList**. The ArrayList is a re-sizable or dynamic array, which means that its size is not pre-determined, thus it is appropriate for our circumstance because we don't know how many items will be generated *(Java ArrayList, n.d.)*. Furthermore, it can be accessed from any location, whereas linked lists can only be accessed sequentially.

The data from the ArrayList is displayed to the table model in GUI design using the display function show *order()*. By looping through the ArrayList, obtaining the object, and using getter functions to obtain the required data, we can show the data in the table.

The program's application also enables users to display in the table specific sets of criteria. The technique will remain the same; the only difference will be the use of different queries to collect data from the database to the ArrayList (as shown below):



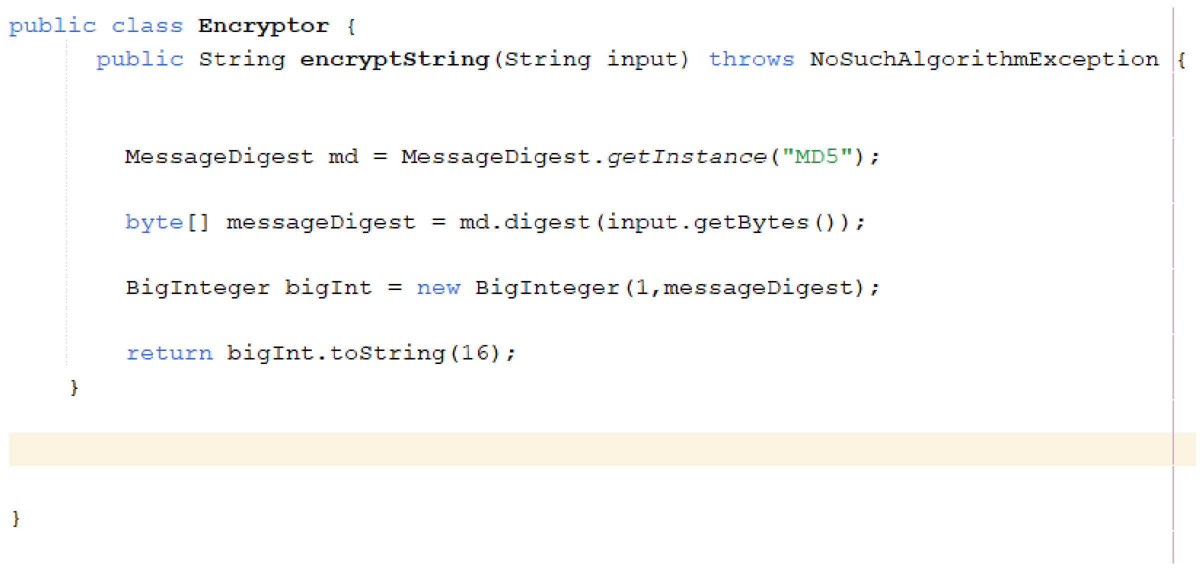
```
public ArrayList<Orders> orderList(){

    ArrayList<Orders> ordersList = new ArrayList<>();
     PreparedStatement pst = connection.pst;
                Connection con = connection.con;
                ResultSet rs = connection.rs;
    try {
        String sql = "SELECT * FROM order1"; // query to fetch data from the local database
        pst = con.prepareStatement(sql);
        rs = pst.executeQuery();
        Orders order;
        System.out.println("a");
        while (rs.next()){
            order = new Orders (rs.getString("ID"), rs.getString("Name"), rs.getDouble("Price"),
                    rs.getString("Status"), rs.getString("Email"), rs.getString("Address"),
                    rs.getString("ITEM_1"), rs.getString("ITEM_2"), rs.getString("ITEM_3"));
            // create an Orders object using the data fetch from the collection resultset rs
            ordersList.add(order);
            // add the Orders object to an ArrayList
        }
    } catch (Exception e) {
        System.out.println(e.toString());
    }finally{
            try{
                rs.close();
                pst.close();
            }catch (Exception e){

            }
        }
     return ordersList;
```



```
public ArrayList<Orders> findorderList(){
   Preparing.setActionCommand("Preparing"); // set a String to Preparing radio button option
   Shipping.setActionCommand("Shipping"); // set a String to Shipping radio button option
   Received.setActionCommand("Received"); // set a String to Received radio button option
   ArrayList<Orders> ordersList = new ArrayList<>();
    PreparedStatement pst = connection.pst;
               Connection con = connection.con;
               ResultSet rs = connection.rs;

   if (buttonGroup1.getSelection() == null){

        try {
       // query to fetch data from the local database
       String sql = "SELECT * FROM order1 WHERE order1.ID LIKE? OR order1.Name LIKE? ";
       pst = con.prepareStatement(sql);
       pst.setString(1,ID.getText());
       pst.setString(2,Name.getText());

       rs = pst.executeQuery();
       Orders order1;

       while (rs.next()){
       order1 = new Orders (rs.getString("ID"), rs.getString("Name"), rs.getDouble("Price"),
               rs.getString("Status"), rs.getString("Email"), rs.getString("Address")
               , rs.getString("ITEM_1"), rs.getString("ITEM_2"), rs.getString("ITEM_3"));
        // create an Orders object using the data fetch from the collection resultset rs
           ordersList.add(order1);
         // add the Orders object to an ArrayList
       }
```

Hash function

To have better security for the application, I implemented a **hash function** to encrypt the newly registered passwords before entering them into the database.

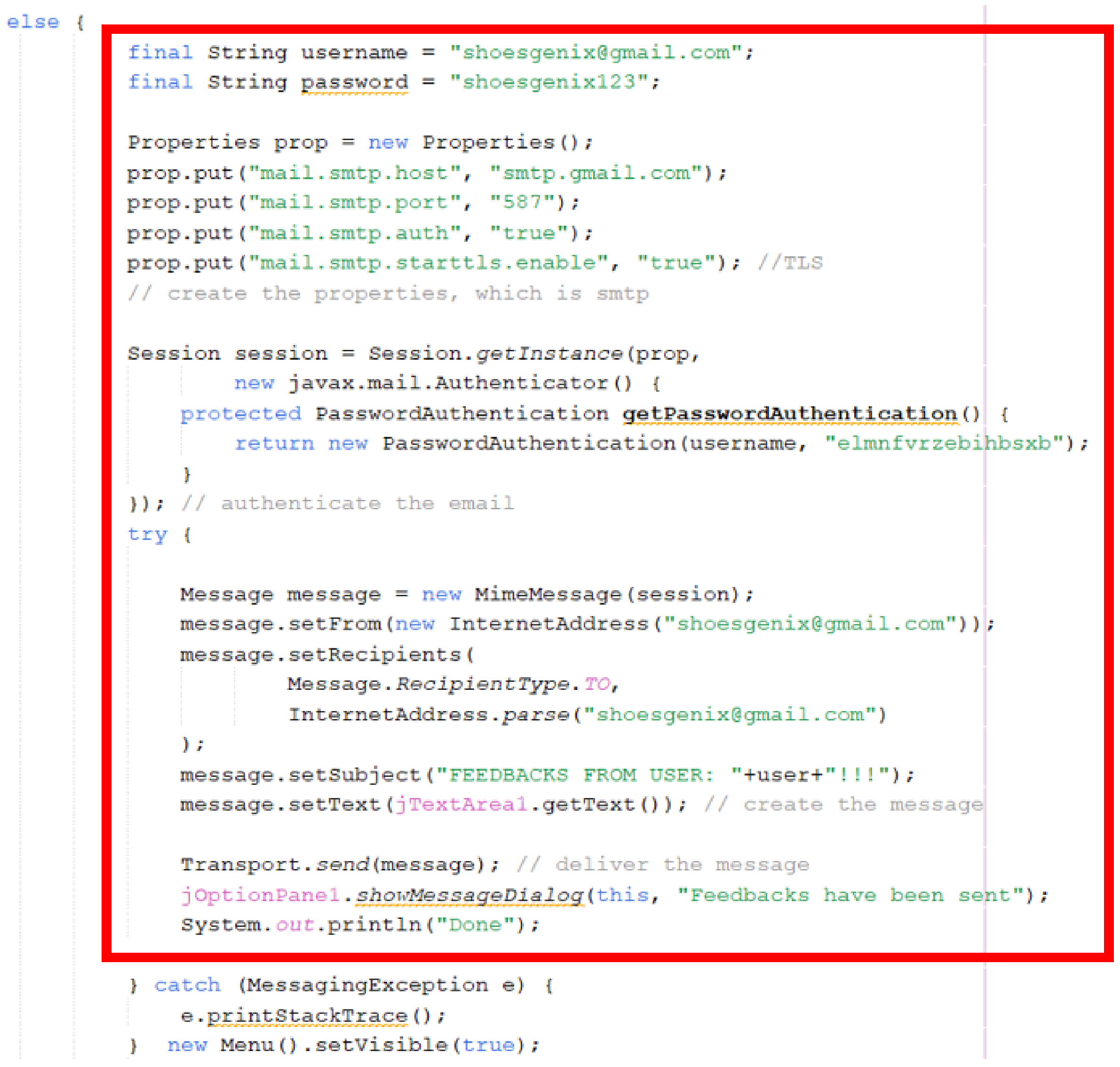


```
public class Encryptor {
    public String encryptString(String input) throws NoSuchAlgorithmException {


        MessageDigest md = MessageDigest.getInstance("MD5");

        byte[] messageDigest = md.digest(input.getBytes());

        BigInteger bigInt = new BigInteger(1,messageDigest);

        return bigInt.toString(16);
    }



}
```

I utilized the **MD5 algorithm**, which is a cryptographic system used for message authentication, content verification, and digital signatures. MD5 will generate a string of 128-bit hash values, which are commonly represented as 32-digit hexadecimal digits *(Shacklett & Loshin, 2021)*.

Email-Sending System

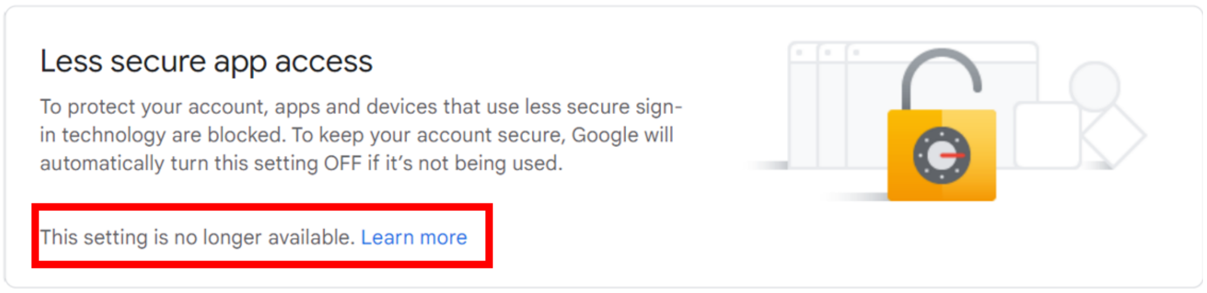
I created an email-sending system to meet the success requirement of allowing users to provide feedback. **JavaMail**, an external Java Library implemented with SMTP, IMAP, and POP3 protocol providers, enabled me to build a process to send an email from the shop's email to other emails from the application itself, without interacting with the dedicated email account.



```java
else {
    final String username = "shoesgenix@gmail.com";
    final String password = "shoesgenix123";

    Properties prop = new Properties();
    prop.put("mail.smtp.host", "smtp.gmail.com");
    prop.put("mail.smtp.port", "587");
    prop.put("mail.smtp.auth", "true");
    prop.put("mail.smtp.starttls.enable", "true"); //TLS
    // create the properties, which is smtp

    Session session = Session.getInstance(prop,
            new javax.mail.Authenticator() {
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(username, "elmnfvrzebihbsxb");
        }
    }); // authenticate the email
    try {

        Message message = new MimeMessage(session);
        message.setFrom(new InternetAddress("shoesgenix@gmail.com"));
        message.setRecipients(
                Message.RecipientType.TO,
                InternetAddress.parse("shoesgenix@gmail.com")
        );
        message.setSubject("FEEDBACKS FROM USER: "+user+"!!!");
        message.setText(jTextArea1.getText()); // create the message

        Transport.send(message); // deliver the message
        jOptionPane1.showMessageDialog(this, "Feedbacks have been sent");
        System.out.println("Done");

    } catch (MessagingException e) {
        e.printStackTrace();
    }   new Menu().setVisible(true);
```

The code in the red box above allows the user to send an email to a single recipient. I created an email-sending process using SMTP that includes everything from preparing the properties to authenticating the sending email, creating the email, and sending it.

However, in order to allow the code to interact with the dedicated program, I did an **external configuration** to run the code.

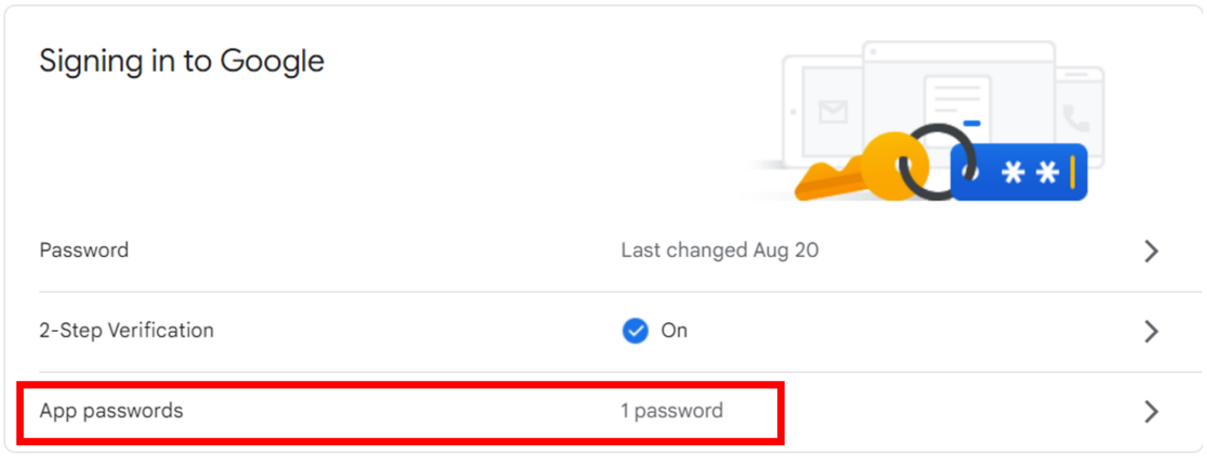


Less secure app access

To protect your account, apps and devices that use less secure sign-in technology are blocked. To keep your account secure, Google will automatically turn this setting OFF if it's not being used.

This setting is no longer available. Learn more

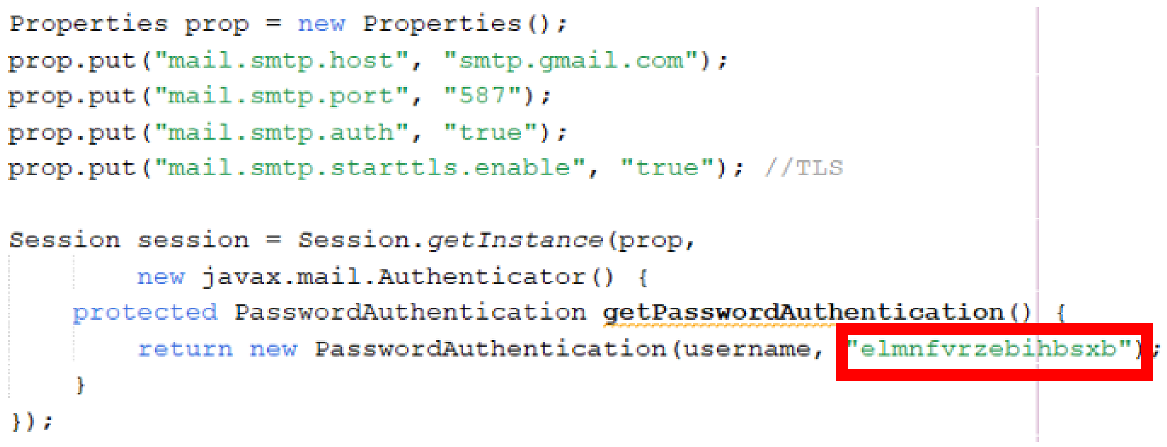
However, after May 30th, due to some privacy policies of Google, this feature is removed. I then had to create a 2-step verification and create an app password for my program to get access to the email.



Signing in to Google

Password	Last changed Aug 20
2-Step Verification	✓ On
App passwords	1 password

This app password will generate a random password for the program. And during the authentication process from the code, the random passcode was used instead of the email's password. For instance, in the class SendFeedbacks:



```
Properties prop = new Properties();
prop.put("mail.smtp.host", "smtp.gmail.com");
prop.put("mail.smtp.port", "587");
prop.put("mail.smtp.auth", "true");
prop.put("mail.smtp.starttls.enable", "true"); //TLS

Session session = Session.getInstance(prop,
        new javax.mail.Authenticator() {
    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication(username, "elmnfvrzebihbsxb");
    }
});
```

Word count: 1100

REFERENCES:

Gillis, A. S., & Lewis, S. (2021, July 13). *object-oriented programming (OOP)*. App Architecture. https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

Java - Encapsulation. (n.d.). https://www.tutorialspoint.com/java/java_encapsulation.htm

Java Polymorphism. (n.d.). https://www.w3schools.com/java/java_polymorphism.asp

Processing SQL Statements with JDBC (The Java™ Tutorials > JDBC Database Access > JDBC Basics). (n.d.). https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html

Shacklett, M. E., & Loshin, P. (2021, August 23). *MD5*. Security. https://www.techtarget.com/searchsecurity/definition/MD5

Understanding SQLITE_BUSY • ActiveSphere - Software consulting. (2018, December 24). https://activesphere.com/blog/2018/12/24/understanding-sqlite-busy

Java ArrayList. (n.d.). https://www.w3schools.com/java/java_arraylist.asp