Criteria C - Development

List of Techniques Used:
SQLite/JDBC (+DB Browser for SQLite)
Error/Exception Handling
Event Handling
Validation Techniques
Encapsulation
Design and Implementation of Swing GUI
External Libraries — *Javax Mail, JFileChooser*
JSON Object and Arrays
TextField Auto-Complete

*entire annotated code listing is presented in **Appendix E**

1) SQLite /JDBC(+ DB Browser for SQLite):

The reason for choosing SQLite as my database management system (instead of MySQL) can be seen below.

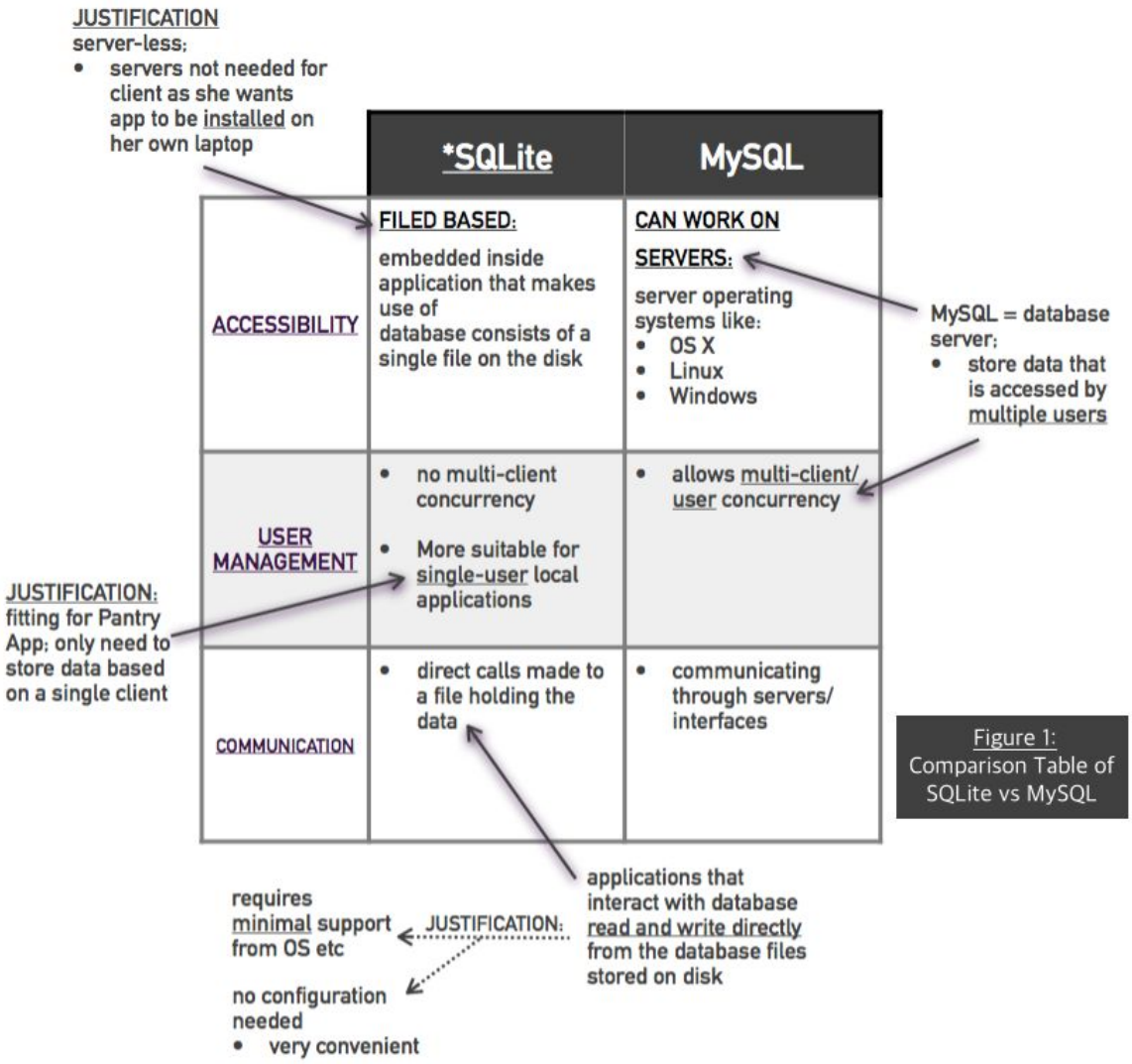


Figure 1: Comparison Table of SQLite vs MySQL

I constructed a SQLite database through using the application DB Browser. The programme required many sets of data to be stored (ex. Kitchen/Pantry items, Shopping items, types of recipes). Hence, I created multiple tables in the database (see Figure 2) to satisfy the different data storage needs.

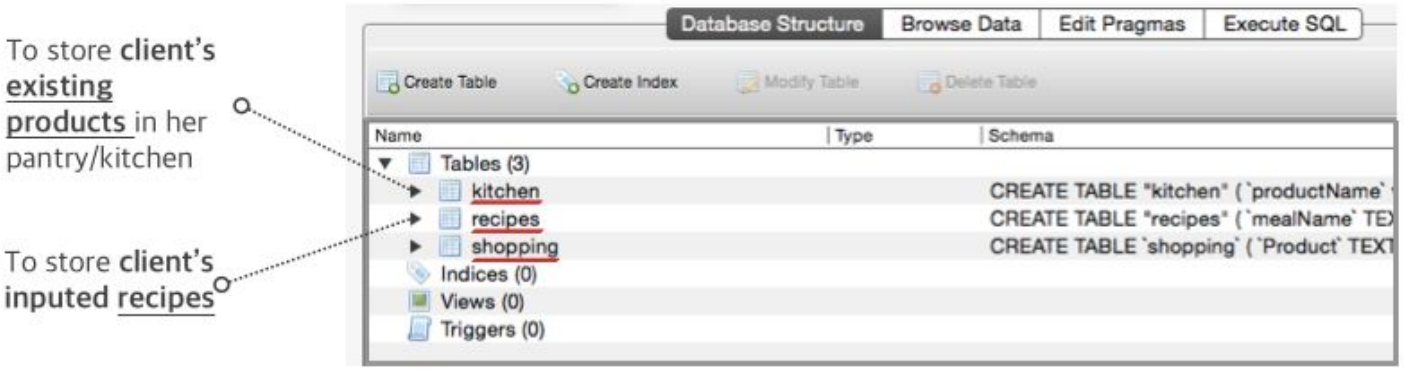


Figure 2: Overview of TABLES stored in SQLite Database using DB Browser

In order to generate connections between the SQLite database and multiple classes, JDBC Driver had to be first installed into NetBeans (see Figure 3 below),

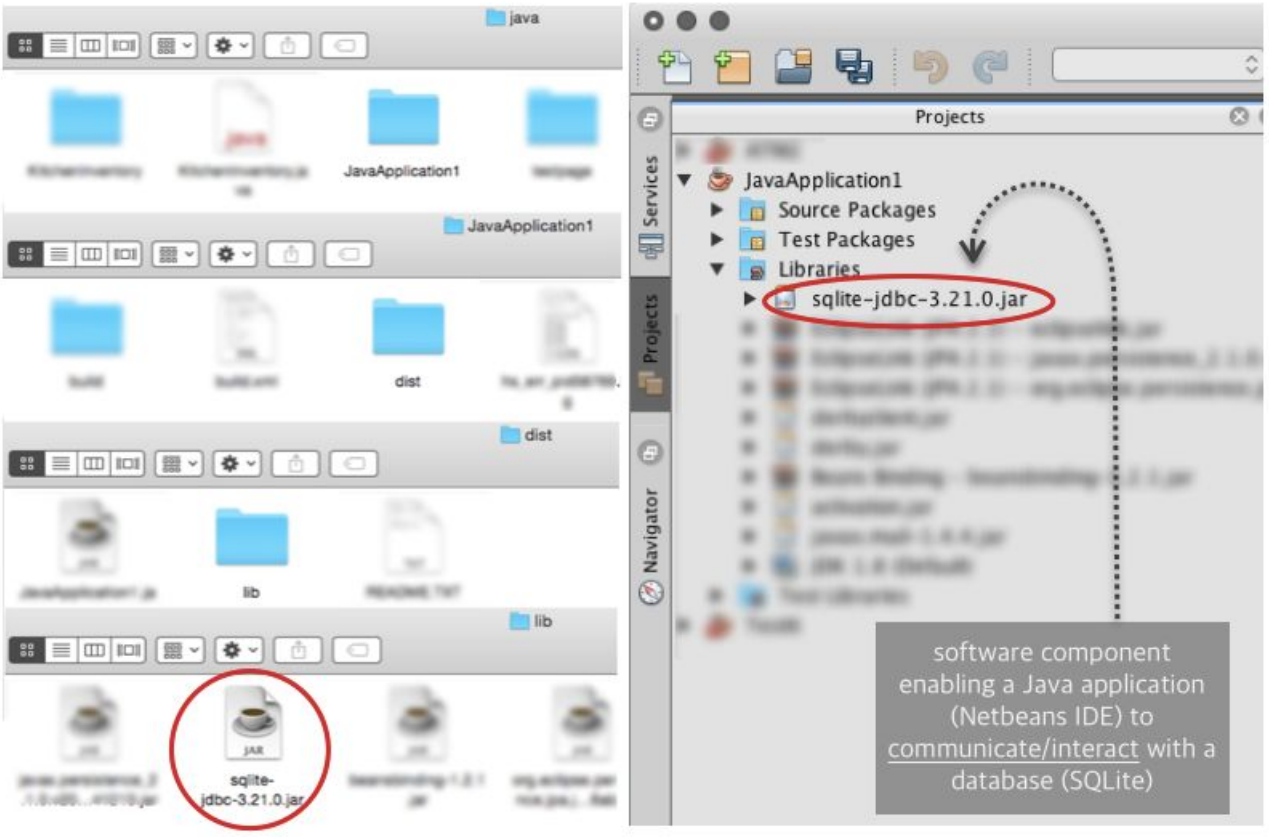


Figure 3: Installation Process of JDBC Driver into NetBeans IDE

and then a formulated code of Java syntax (see Figure 4) had to be used throughout the project to allow access and interaction with the contents of the SQLite database.

```java
//Try-Catch statement required as connection to SQLite database
//may sometimes not be successful due to specific issues
try {
    //Load the (already installed) JDBC Driver from NetBeans
    Class.forName("org.sqlite.JDBC");
    //Creating the connection with the specific database
    //in SQLite (project.db)
    c = DriverManager.getConnection("jdbc:sqlite:project.db");
    //'false' to disable AUTO COMMIT for connection
    c.setAutoCommit(false);
    //'Statement' is an interface which represents a SQL statement
    Statement s = c.createStatement();



    //code



    //saves and stops connection with SQLite database
    s.close();
    c.commit();
    c.close();
//catches and displays the exception/error if SQLite connection does
//not go through successfully
} catch (Exception e) {
    System.err.println(e.getClass().getName() + ": " + e.getMessage());
    System.exit(0);
}
```

Figure 4: Java syntax for connection to SQLite Database

To interact with specific data from different tables in the SQLite Database, SQL query statements were used, as shown below:

Class: PantryJFrame -'INSERT' in jButton_addActionPerformed() method

```
//name of product inputted by client is retrieved
String nameText = someTextField.getText();

//if nameText inputted by client consists of multiple words, the first letter
//of each word is automatically capitalized for better appearance/aesthetics.

//words are split up (by spaces)
String[] words = nameText.split(" ");

//capitalize each word
for (int i = 0; i < words.length; i++) {
    words[i] = words[i].substring(0, 1).toUpperCase() + words[i].substring(1).toLowerCase();
}

//words are rejoined back into a sentence
nameText = String.join(" ", words);


//the quantity of each product is retrieved from textfield inputted by client
String quantityText = jTextField_quantity.getText();

//retrieving inputs from 3 Combo Boxes of Expiry Day, Month, and Year from client and
//storing them together as a single object
Object expiryDayText = jComboBox_expiryDay.getSelectedItem(); //day of expiry
Object expiryMonthText = jComboBox_expiryMonth.getSelectedItem(); //month of expiry
Object expiryYearText = jComboBox_expiryYear.getSelectedItem();// year of expiry
Object eDate = "" + expiryDayText + "-" + expiryMonthText + "-" + expiryYearText + ""; //full expiry date

//converting format of expiry date inputed from user into a more universal format
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MMM-yyyy");
LocalDate eFormattedDate = LocalDate.parse((CharSequence) eDate, formatter);
```

specified name of data table in SQLite database

names of the column headers in 'kitchen' table

```
//adding of product into database by using SQLite query statement
s.executeUpdate("INSERT INTO kitchen (productName,amount,expiryDay)values"
        + "('" + nameText + "','" + quantityText + "','" + eFormattedDate + "')");
```

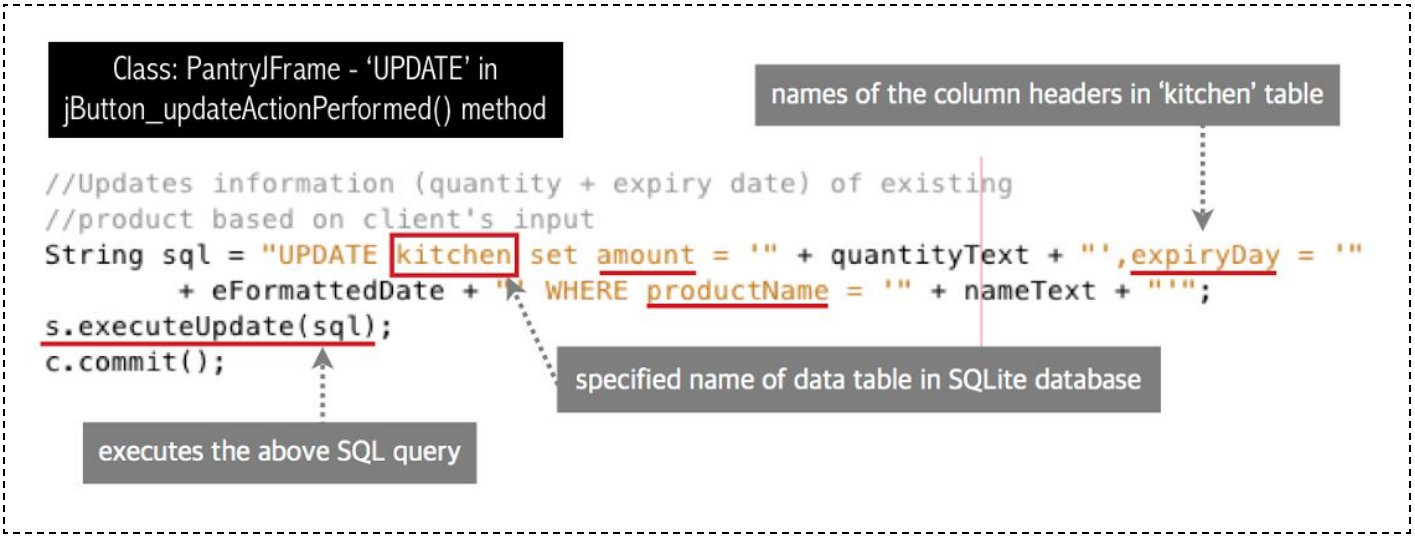
values that are being inputted into the respective columns into SQLite database

Table: kitchen

	productName	amount	expiryDay
	Filter	Filter	Filter
1	Yoghurt	3	2018-06-07
2	Oatmeal	5	2018-03-04
3	Butter	1	2020-02-09
4	White Bread	4	2019-03-06
5	Red Pepper	2	2023-04-02

DB Browser displaying data stored in 'kitchen' table in SQLite database


no specific columns/values are mentioned because
Class: ShopListJFrame - 'DELETE' in jButton_clearActionPerformed() method
//SQL query statement for removing all items from 'shopping'table
//in SQLite database
String query = "DELETE from shopping';
s.executeUpdate(query);
c.commit();
executes the above SQL query
specified name of data table in SQLite database


Class: PantryJFrame - 'UPDATE' in jButton_updateActionPerformed() method
names of the column headers in 'kitchen' table
//Updates information (quantity + expiry date) of existing
//product based on client's input
String sql = "UPDATE kitchen set amount = '" + quantityText + "',expiryDay = '"
+ eFormattedDate + "' WHERE productName = '" + nameText + "'";
s.executeUpdate(sql);
c.commit();
specified name of data table in SQLite database
executes the above SQL query

Class: PantryJFrame - 'SELECT' in formComponentShown() method

```
//Retrieve the current date
LocalDateTime currentTime = LocalDateTime.now();

//Instantiates a universal date format
DateTimeFormatter formatter2 = DateTimeFormatter.ofPattern("yyyy-MM-dd");

//Sets the format to the current date
String formatDateTime = currentTime.format(formatter2);

//Checking if any existing products found in database is expired on current date
//Selects all the existing products that has their respective expiry date(s) equal the current date
ResultSet rs = s.executeQuery("SELECT EXISTS(SELECT 1 FROM kitchen WHERE expiryDay "
        + "LIKE '" + formatDateTime + "')AS productName");

while (rs.next()) {
    int exp = rs.getInt("productName");
    String expired = "EXPIRED";

    //if product's expiration date = current date, means it has expired
    if (exp == 1) {

        //the 'expiryDay' column of all expired products
        //is updated to indicate to client it has expired
        String query = "UPDATE kitchen set expiryDay = '"
                + expired + "' WHERE expiryDay LIKE '"
                + formatDateTime + "'";
        s.executeUpdate(query);
        c.commit();

        DefaultTableModel model = (DefaultTableModel) jTable_Display_User.getModel();
        model.setRowCount(0);
        show_user();
```

1 will be outputted if formatDateTime exists/is present within the dates of existing products

In GUI-displayed classes ex. RecipesJFrame, the data stored in their respective tables in SQLite must be displayed to the client. This was achieved by having jTable(s) be populated by data from the database.

```
public ArrayList<Recipe> recipeList() {
    Connection c = null;
    //Inserting data from 'recipes' table from SQLite database into a new Arraylist
    //before being populated into jTable

    //'Recipe' is from the Recipe class where there are accessors and mutators
    //based on column headings for 'recipe' table
    //ArrayList used to populate jTable
    ArrayList<Recipe> recipesList = new ArrayList<>();
    try {
        //Establish SQLite connection
        Class.forName("org.sqlite.JDBC");
        c = DriverManager.getConnection("jdbc:sqlite:project.db");
        c.setAutoCommit(false);
        Statement s = c.createStatement();

        //Programmatically setting up properties/dimensions/appearance of jTable in GUI
        TableColumnModel columnModelR = recipesjTable.getColumnModel();
        columnModelR.getColumn(0).setPreferredWidth(10000);
        columnModelR.getColumn(1).setPreferredWidth(0);
        columnModelR.getColumn(2).setPreferredWidth(0);
        columnModelR.getColumn(3).setPreferredWidth(0);
        recipesjTable.setRowHeight(70);

        //Retrieving ALL data from 'recipes' table in SQLite database
        //so can be populated into jTable
        String querySelectR = "SELECT * FROM recipes";
        ResultSet rsR = s.executeQuery(querySelectR);
        Recipe recipe;

         //Retrieving ALL data from 'recipes' table in SQLite database
         //so can be populated into jTable
         String querySelectR = "SELECT * FROM recipes";
         ResultSet rsR = s.executeQuery(querySelectR);
         Recipe recipe;

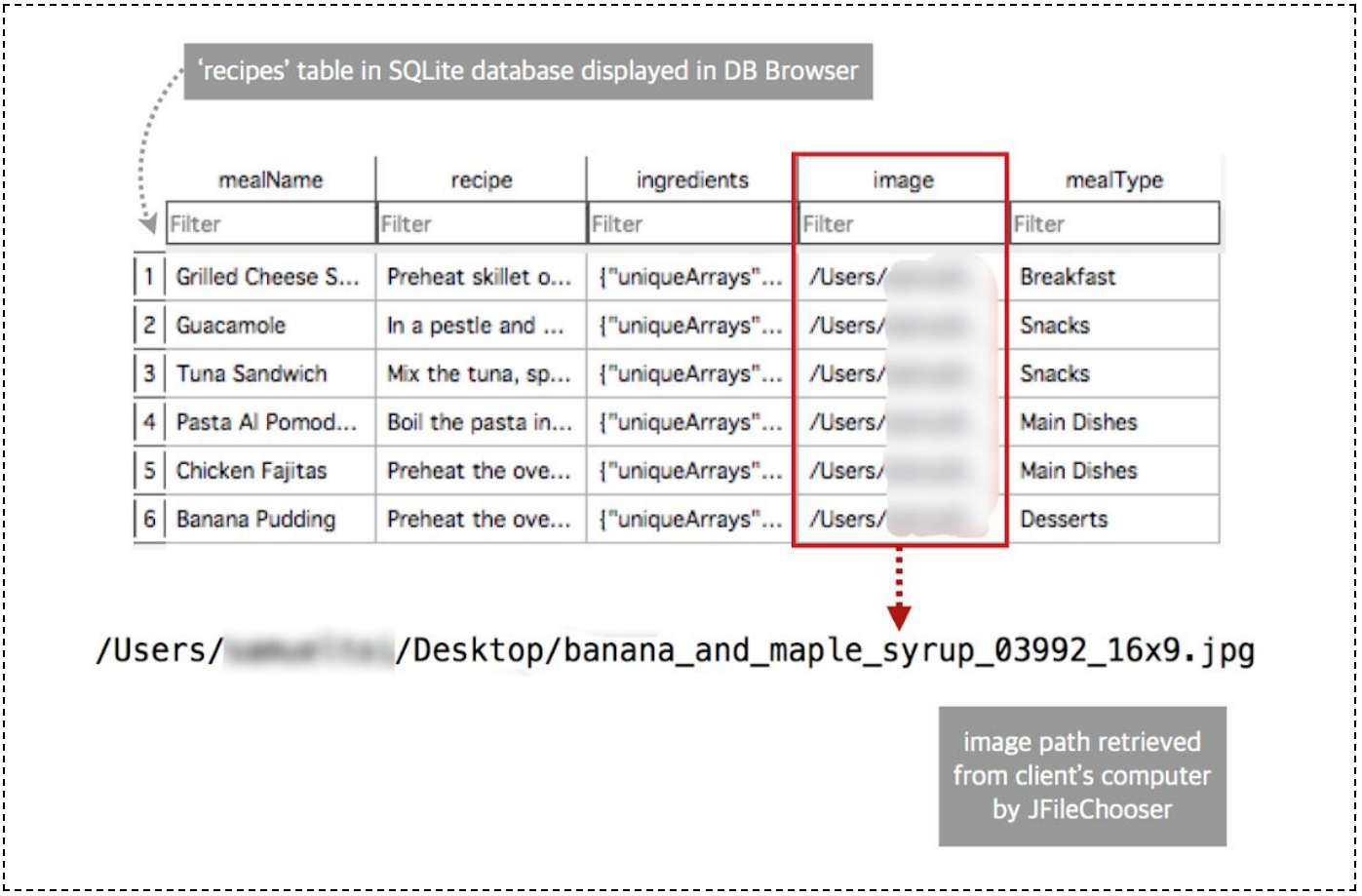
         //retrieving all the column headers from 'recipes' table
         //in SQLite database
         while (rsR.next()) {
             recipe = new Recipe(rsR.getString("mealName"), rsR.getString("recipe"),
                     rsR.getString("ingredients"), rsR.getString("image"),
                     rsR.getString("mealType"));
             recipesList.add(recipe);
         }
         //saves and stops connection with SQLite database
         s.close();
         c.commit();
         c.close();

         //catches and displays the exception/error if SQLite connection does
         //not go through successfully
     } catch (Exception e) {
         System.err.println(e.getClass().getName() + ": " + e.getMessage());
         System.exit(0);
     }
     return recipesList;
 }
```

Justification: There various ways to populate a jTable with data from a database, but I chose to do it through Arraylists. Other approaches included implementing the whole database as a jTable using the functionality provided by NetBeans; a less programmatic way, and a more structural one. Having the data in Arraylists not only allows dynamic changes, but also increased flexibility in terms of interacting with the data from SQLite. Having the data be implemented programmatically gave me opportunities to improve the functionality as well.

Storing Images inside SQLite Database:

In RecipeJFrame class, images for each recipe(s) must be saved into SQLite database and thus, the 'recipes' SQLite table included a column called 'image', where the image path was stored.



*student's name is blurred for anonymity

Justification: Other approaches include storing the actual image file inside SQLite. However, storing images as it's respective paths allows more flexibility. It allows storage of many images without worrying about image sizes and more importantly, about the space and performance of the database. Storing the actual image files would potentially slow down the performance, as it takes more effort to load them to a high resolution.

2) Error/Exception Handling

One of my main techniques to combat any errors/exceptions that may appear throughout the project was the used of Try and Catch statements.

```
try {

    JLabel label = new JLabel("Your shopping list is currently empty. Please try again");
    final ImageIcon icon2 = new ImageIcon
    (new URL("file:///Users/samueltoi/Desktop/Screen%20Shot%202018-02-18%20at%209.59.18%20am.png"));
    JOptionPane.showMessageDialog(null, label, "Alert", JOptionPane.INFORMATION_MESSAGE, icon2);
   // jList4.setVisible(true);


} catch (MalformedURLException ex) {
    Logger.getLogger(DatesJFrame.class.getName()).log(Level.SEVERE, null, ex);
}
```

Figure 9: Example of Try-Catch Block used in retrieving an Image Icon for JOptionPane Error

```
try {
    //Establishes SQLite connection to database
    Connection c = null;
    Class.forName("org.sqlite.JDBC");
    c = DriverManager.getConnection("jdbc:sqlite:project.db");
    c.setAutoCommit(false);
    Statement s = c.createStatement();

    //SQL query statement for removing all items from shopping database
    String query = "DELETE from shopping";

    s.executeUpdate(query);
    c.commit();

    //JTable is refreshed to show new changes
    DefaultTableModel model = (DefaultTableModel) jTable_shopping.getModel();
    model.setRowCount(0);

    //SQLite connection saved and closed
    s.close();
    c.commit();
    c.close();
} catch (Exception e) {
    JOptionPane.showMessageDialog(null, "Error");
}
```

Figure 10: Example of Try-Catch Block for SQLite Connection in ShopListJFrame

When there is an exception, whether it is IOException, or SQLException, instead of terminating the whole program, the 'catch' takes care of it, and the rest of the program can be executed smoothly. The Try-Catch statement also makes it more convenient for programmers to easily identify the issue; like a debugging technique. The type of exception + it's issue will be displayed in the console (or in the form of a JOptionPane).

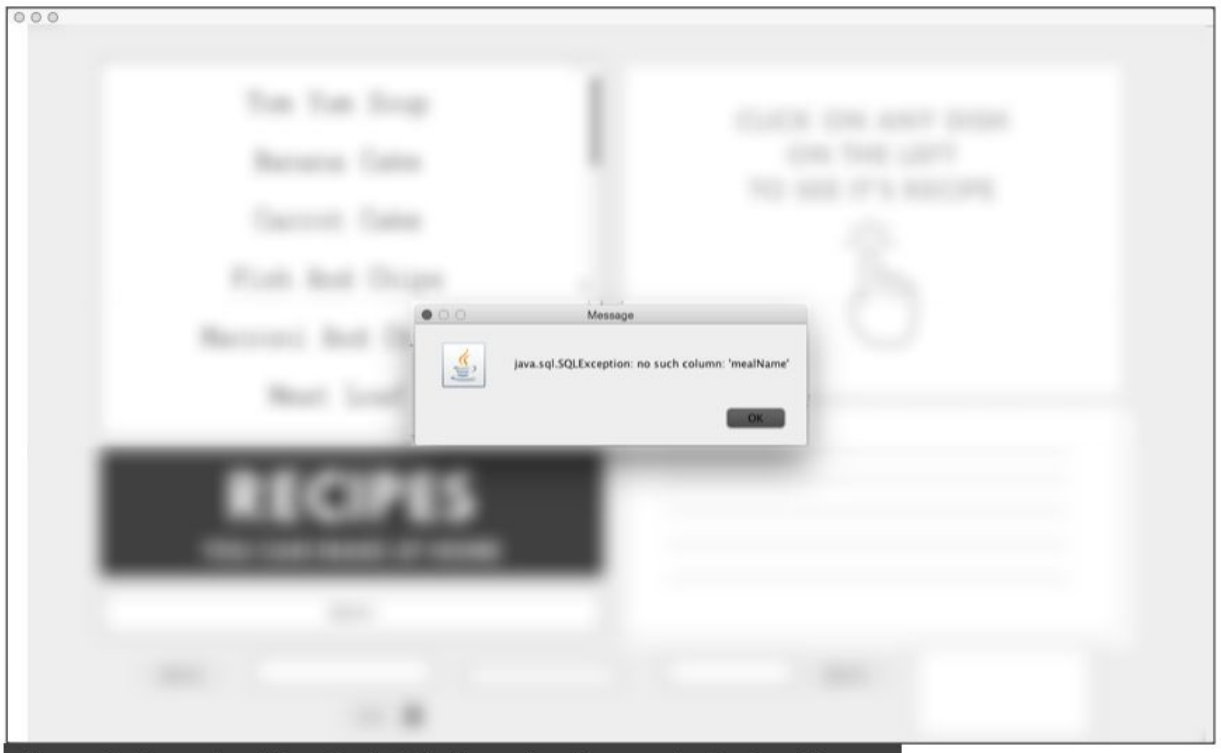


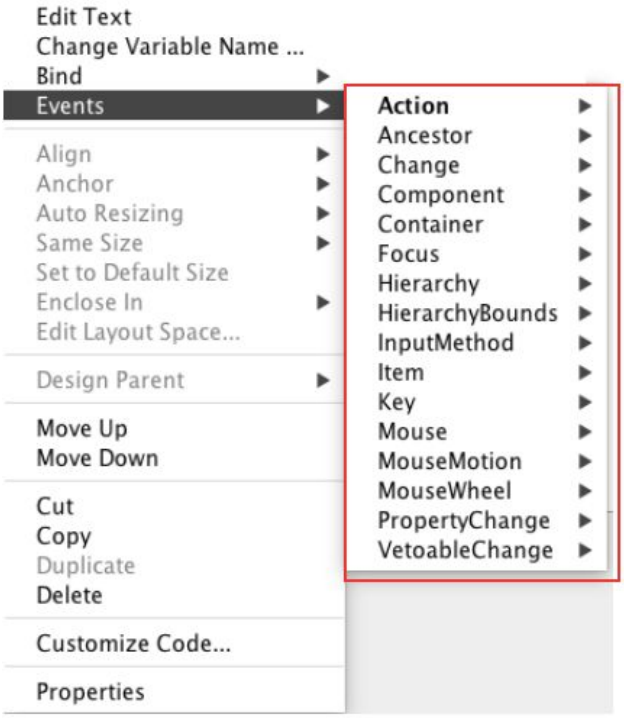
Figure 11: Example of Try-Catch SQL Exception Message in RecipesJFrame

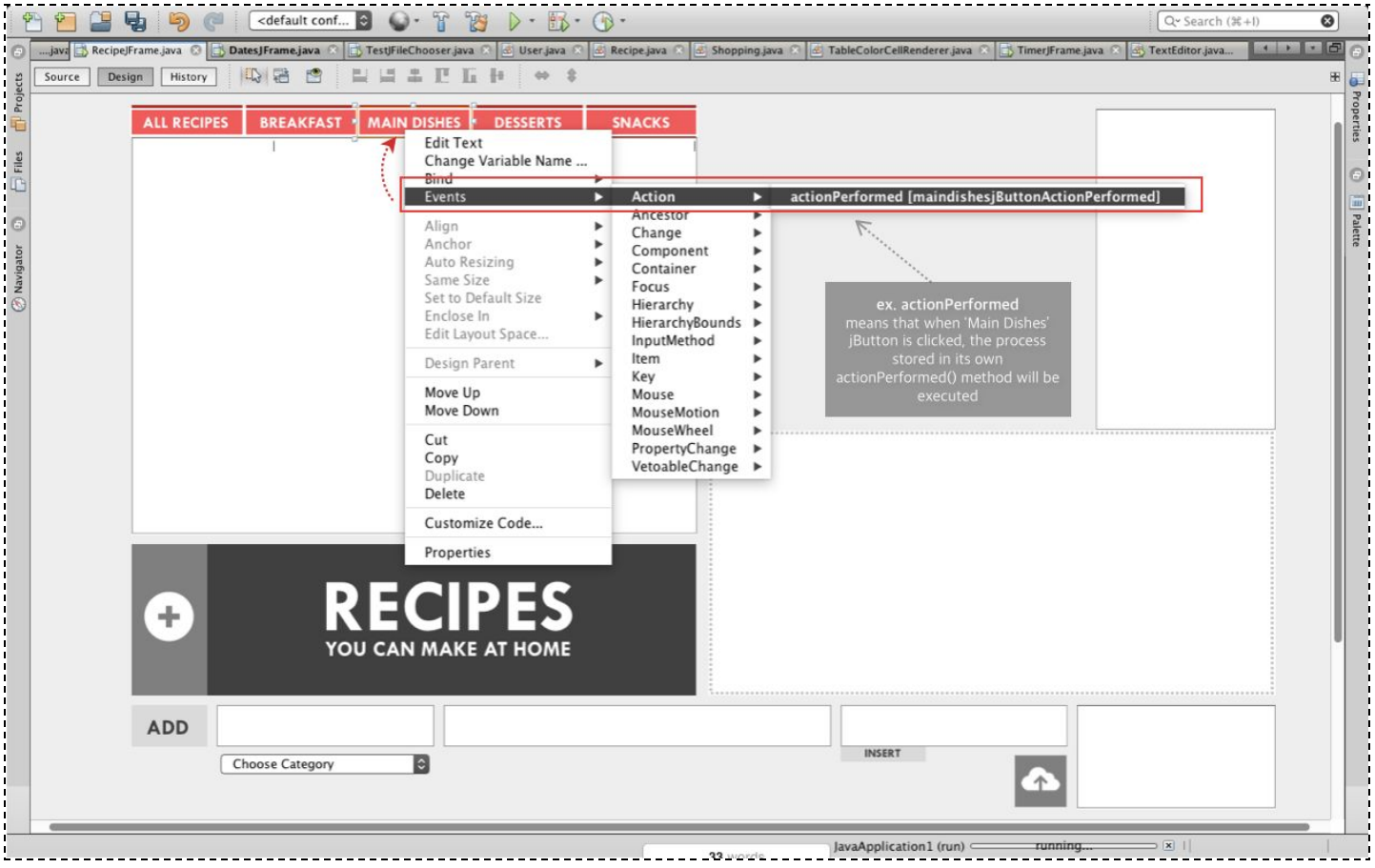
Justification: Having the specific exception identified also indicates the type of issue that has to be examined further. For example, if the exception is 'SQLException', the programmer knows that the problem is not within the actual Java syntax itself, but maybe the SQLite statements or database. Hence, the use of the Try-Catch statement allows the separation of the identification between program code and error-handling code. It makes it easier for the programmer to fully understand and maintain the logical flow of the program as a whole.

SQLite databases is an external system, so there is a high risk of unpredictability and unknown issues, including network connection errors, unreachable databases; not being able to retrieve data etc. Therefore, Implementing Try-Catch statements is extremely vital when a database management system (ex. SQLite) is constantly referred to throughout the project.

3) Event Handling

Most importantly, the project involves many user-interacting aspects throughout the GUI(s), including clicking buttons etc. Each of these swing components has its own 'event', being able to perform and execute various of tasks and actions when it is interacted with.

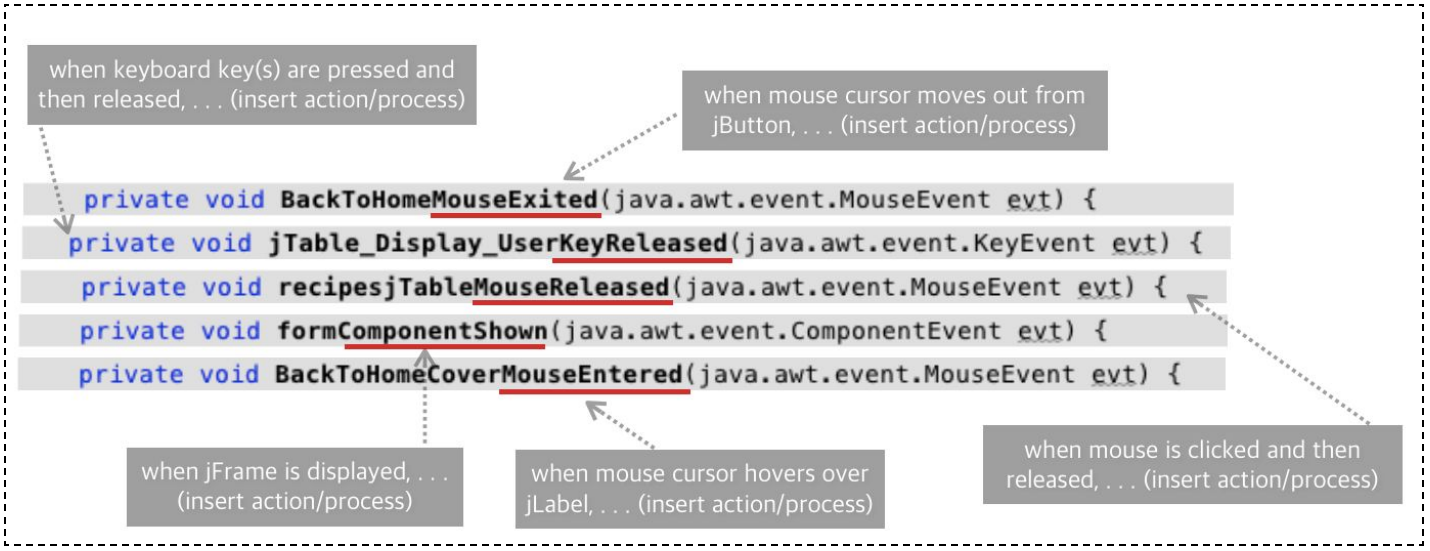




```
    private void cboxSecCActionPerformed(java.awt.event.ActionEvent evt) {
        lblSecC.setText("" + cboxSecC.getSelectedItem());
        secC = Integer.parseInt(lblSecC.getText());
    }
```

```
private void ClickToRecipesJButtonActionPerformed(java.awt.event.ActionEvent evt) {
    //Directs client to RecipeJFrame
    RecipeJFrame RJF = new RecipeJFrame(this);
    RJF.setMyData(12345);
    RJF.setVisible(true);
    this.setVisible(false);
}
```

Other examples are displayed below:

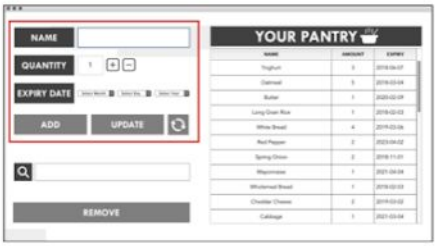


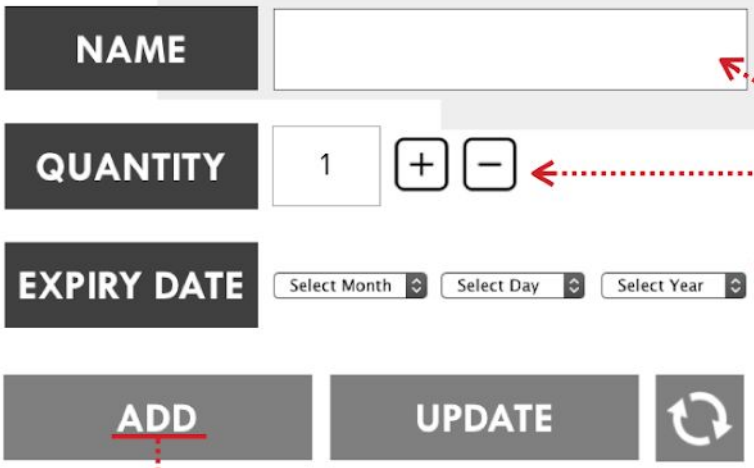
The use of event handling not only improves the overall functionality of the program, but also strengthens the convenience provided by the swing components, hence improving the client experience/interaction with the GUI. With each swing component consisting of their individual code, it also makes it easy and fast for programmers to locate the source of error(s) and debug/fix them.

4) Validation Techniques

a) <u>Presence Checker</u>:

Class:
PantryJFrame





'Name' ,
'Quantity',
'Expiry Date'
are all <u>mandatory fields</u> client needs to fill out before proceeding to 'ADD'

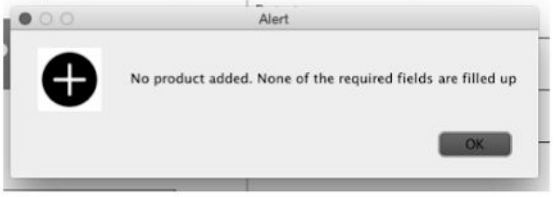
When '<u>ADD</u>' button is clicked . . .

```
//validates if any information is present in any required fields before being added to database
if (jTF_productName.getText() != null && jTF_productName.getText().equals("") && jTextField_quantity.getText() != null
    && jTextField_quantity.getText().equals("1") && jComboBox_expiryDay.getSelectedItem() != null
    && jComboBox_expiryDay.getSelectedItem().equals("Select Day") && jComboBox_expiryMonth.getSelectedItem() != null
    && jComboBox_expiryMonth.getSelectedItem().equals("Select Month") && jComboBox_expiryYear.getSelectedItem() != null
    && jComboBox_expiryYear.getSelectedItem().equals("Select Year")) {

    //if NO information is present in required fields), a warning message is given to client
    try {
        JLabel label = new JLabel("No product added. None of the required fields are filled up");
        final ImageIcon icon3 = new ImageIcon(new URL
        ("file:///Users/samueltoi/Desktop/Screen%20Shot%202018-02-18%20at%2010.57.27%20am.png"));
        JOptionPane.showMessageDialog(null, label, "Alert", JOptionPane.INFORMATION_MESSAGE, icon3);

    } catch (MalformedURLException ex) {
        Logger.getLogger(PantryJFrame.class.getName()).log(Level.SEVERE, null, ex);
    }
    //after warning message, mouse is re-focused back onto first jTextField, allowing client to make changes
    jTF_productName.requestFocusInWindow();
```

Validates if all fields are empty

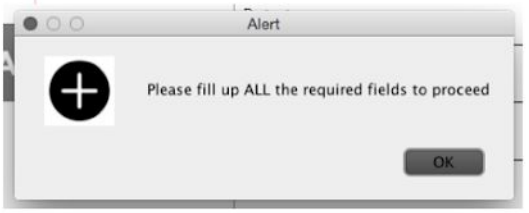



```
} else {
    //if information is present in fields, validates if ALL required fields are filled before being added to database
    if (jTF_productName.getText().trim().length() == 0 || jTextField_quantity.getText().trim().length() == 0
        || jComboBox_expiryDay.getSelectedItem() == "Select Day" || jComboBox_expiryMonth.getSelectedItem() == "Select Month"
        || jComboBox_expiryYear.getSelectedItem() == "Select Year") {

        //if required fields are not all filled up, a warning message is given to client
        try {
            JLabel label = new JLabel("Please fill up ALL the required fields to proceed");
            final ImageIcon icon3 = new ImageIcon(new URL
            ("file:///Users/samueltoi/Desktop/Screen%20Shot%202018-02-18%20at%2010.57.27%20am.png"));
            JOptionPane.showMessageDialog(null, label, "Alert", JOptionPane.INFORMATION_MESSAGE, icon3);

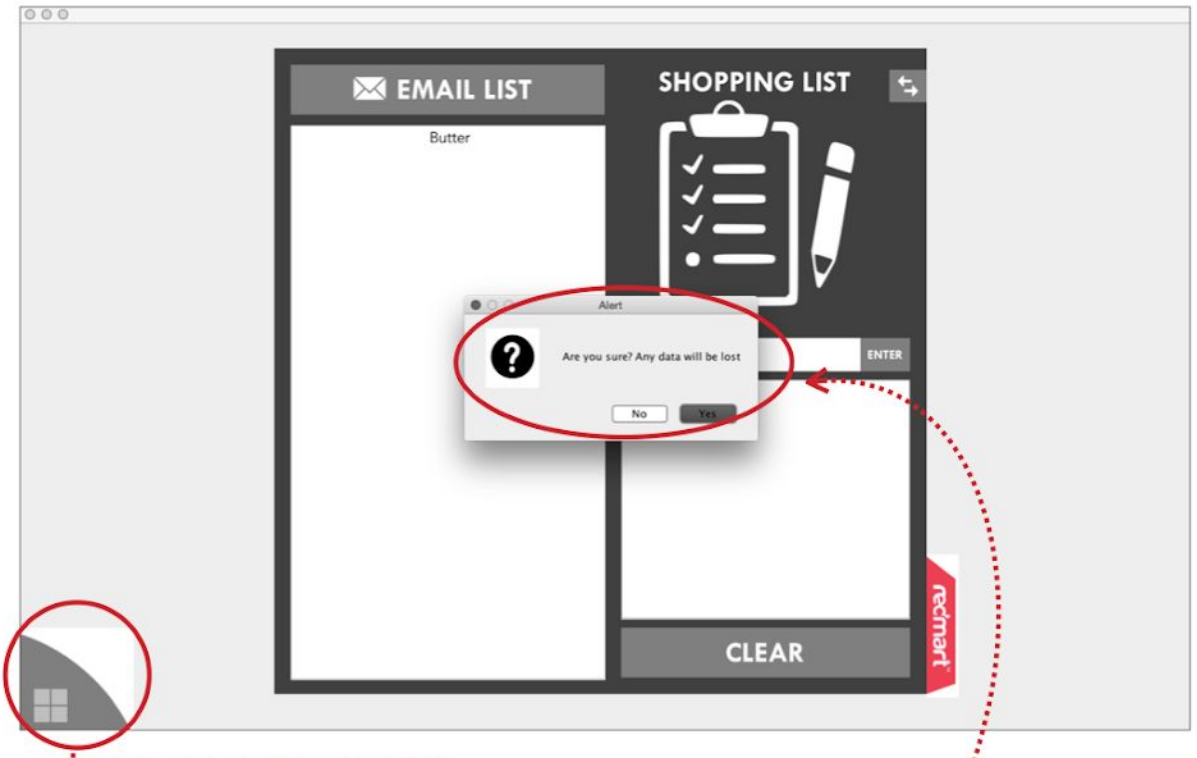
        } catch (MalformedURLException ex) {
            Logger.getLogger(PantryJFrame.class.getName()).log(Level.SEVERE, null, ex);
        }
        //after warning message, mouse is re-focused back onto first jTextField, allowing client to make changes
        jTF_productName.requestFocusInWindow();
```

then Validates if ALL fields are filled up



<u>if-else statement</u> used as a technique of validation

- can have multiple 'else' statement, thus allowing multiple conditions to be validated



When 'Back To Home' button is clicked . . .

```
//Checks if Shopping List (jList4) is empty before taking client back to HOME page
        if (jList4.getModel().getSize() == 0) {
            this.setVisible(false);
            parentDJF.setVisible(true);
        } else {
            //if shopping list contains items, warns and asks client first before going back to HOME page
            //because all contents in list will be lost
            try {
                final ImageIcon icon2 = new ImageIcon(new URL
                ("file:///Users/samueltoi/Desktop/Screen%20Shot%202018-02-18%20at%209.59.18%20am.png"));
                int dialogButton = JOptionPane.YES_NO_OPTION;
                int dialogResult = JOptionPane.showConfirmDialog(this, "Are you sure? Any data will be lost", "Alert",
                        dialogButton, JOptionPane.YES_NO_OPTION, icon2);
                if (dialogResult == 0) {
                    //if client responds by clicking 'Yes', go back to HOME page
                    System.out.println("Yes option");
                    this.setVisible(false);
                    parentDJF.setVisible(true);
                } else {
                    //if client responds by clicking 'No', stay at Shopping List
                    System.out.println("No Option");
                }

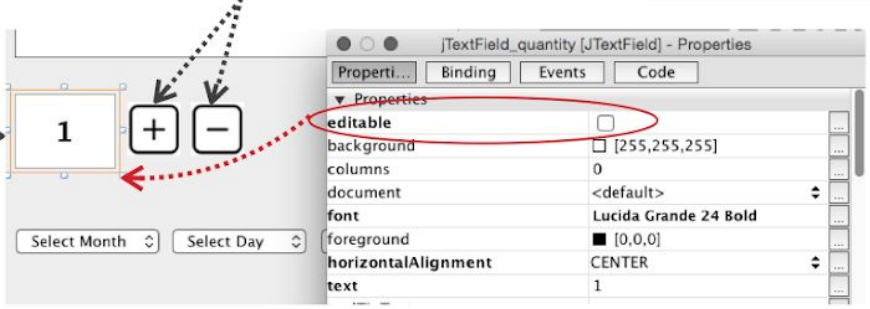
            } catch (MalformedURLException ex) {
                Logger.getLogger(DatesJFrame.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
```

b) Range Checker:

The use of buttons (Event Handling) to inc/dec quantity displayed in textfield is the only way for client to input the quantity of the product

thus can easily determine the range that user can input

Instead of validating user's input, not allowing user to physically type into textfield completely fixes any future errors in terms of quantity of products



Error #1 = Client tries to click the **decrease button** when quantity is '**0**'



when client clicks decrease button . . .

```
private void jButton_decreaseqActionPerformed(java.awt.event.ActionEvent evt) {
    //DECREASE the value in 'Quantity' field

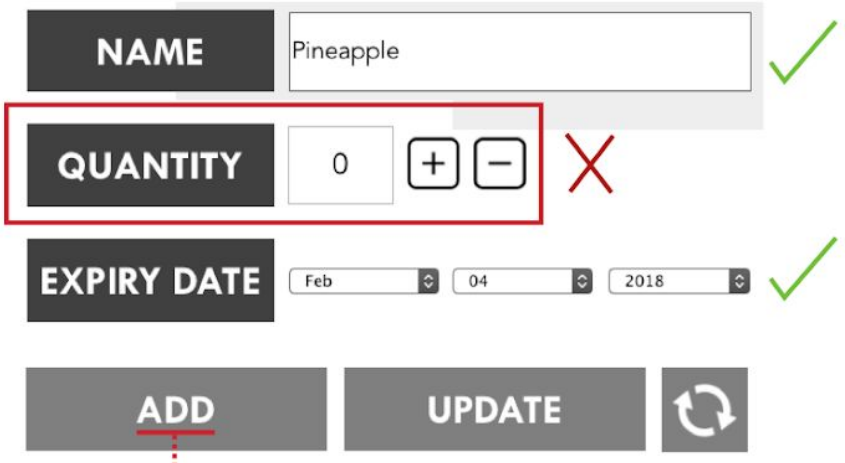
    //Retrieves the current number prsent in jTextField_quantity
    //and instantiates it
    String quantity = jTextField_quantity.getText();

    //Convert the String into an int so it can be validated
    int intQuantity = Integer.valueOf(quantity);

    //If number is more than 0, then a single click of jButton_decreaseq
    //will decrease the number by 1
    if (intQuantity > 0) {
        quantity = Integer.toString(--intQuantity);
        //Sets the new number back into jTextField_quanitity
        jTextField_quantity.setText(quantity);
        //If number is < 0, nothing will happen - it will stay as 0
        //not possible to have a 'negative number' of products
    } else {

    }
}
```

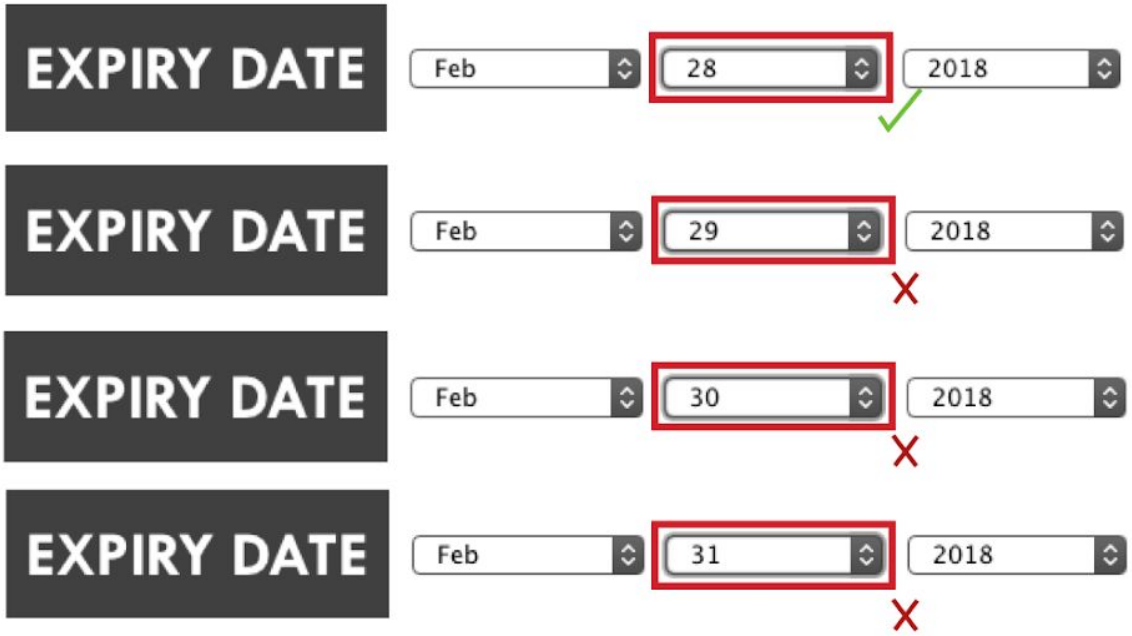
Error #2 = Client **adds** a product with a quantity of '**0**'



when client clicks 'ADD' button . . .

```
//fixes the accidental issue of client inputing '0' as a
//quantity (not possible) by sending a warning message to client
if (jTextField_quantity.getText().equals("0")) {
    JOptionPane.showMessageDialog(null, "ERROR. Please try again");
    //after warning message, mouse is re-focused back onto first jTextField,
    //allowing client to make changes
    someTextField.requestFocusInWindow();
```

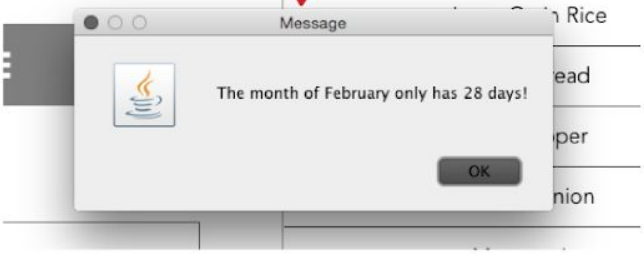
```
private void jButton_addActionPerformed(java.awt.event.ActionEvent evt) {
```



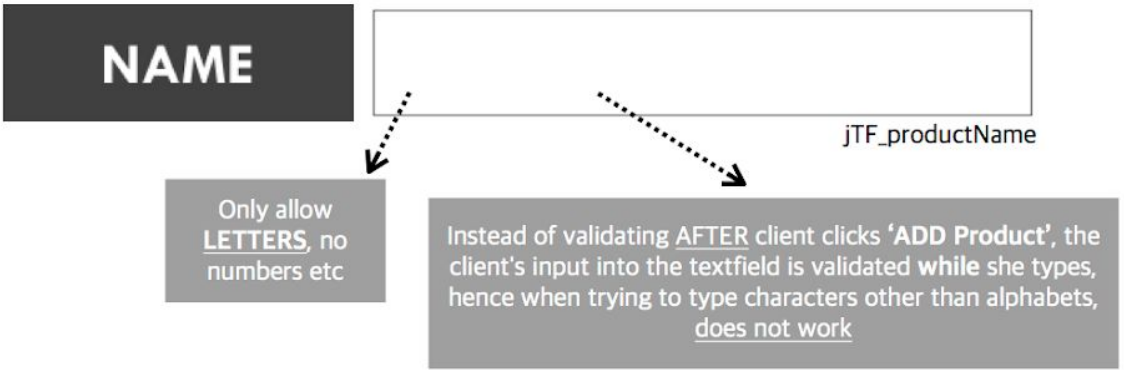
When client clicks the '**ADD**' jButton . . .

```
//fixes the accidental issue of client inputing dates past 28th February by sending a warning message to client
//(does not take into account leap years)
if (jComboBox_expiryDay.getSelectedItem().equals("31") && jComboBox_expiryMonth.getSelectedItem().equals("Feb")
        || jComboBox_expiryDay.getSelectedItem().equals("30") && jComboBox_expiryMonth.getSelectedItem().equals("Feb")
        || jComboBox_expiryDay.getSelectedItem().equals("29") && jComboBox_expiryMonth.getSelectedItem().equals("Feb")) {
    JOptionPane.showMessageDialog(null, "The month of February only has 28 days!");
    //after warning message, process of adding product is terminated
    //mouse is re-focused back onto first jTextField, allowing client to make changes
    someTextField.requestFocusInWindow();
```

if **0 < jComboBox_expiryDay <= 28**, then process of adding product can be continued on

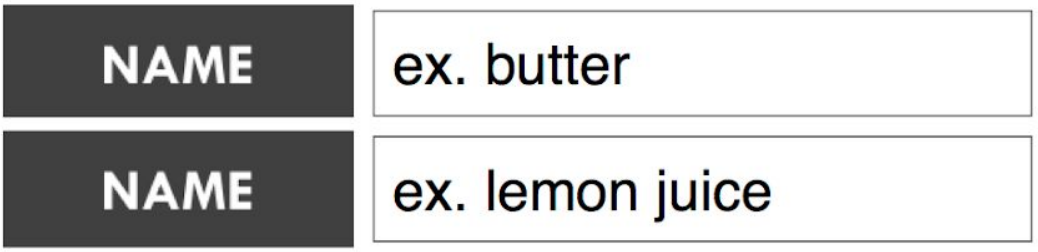


c) Type/Format Checker:




```
private void jTF_productNameKeyTyped(java.awt.event.KeyEvent evt) {
    char c = evt.getKeyChar();

    if (!(Character.isAlphabetic(c) || (c == KeyEvent.VK_BACKSPACE) ||
            c == KeyEvent.VK_DELETE || c == KeyEvent.VK_SPACE)) {
        evt.consume();
    }
}
```




```
//if productName inputed by client consists of multiple words, the first letter
//of each word is automatically capitalized for better appearance/aesthetics.

// words are split up (by spaces)
String[] words = nameText.split(" ");

//capitalize each word
for (int i = 0; i < words.length; i++) {
    words[i] = words[i].substring(0, 1).toUpperCase() + words[i].substring(1).toLowerCase();
}

//words are rejoin back into a sentence
nameText = String.join(" ", words);
```

for all text inputed by user,
first letter will always be made **capitalised**

- convenient for client who is not that familiar with technology
- removes any confusion with lower vs higher case letters with other interconnected processes within program

4) Encapsulation

The program consists of 3 main classes - **User, Shopping**, and **Recipe** and each had respective sets of accessors and mutators. The primary purpose of them is for the retrieval of each column heading from their respective tables in the SQLite database.

```
//Instantiates the column headers of the 'kitchen' table from SQLite database
class User {
    //private variables so not all external classes can have access to them
    //only a selected few
    private String productName; //name of the product
    private int amount; //how many of them
    private String expiryDay; //the expiration date of the product

    //Mutators - sets the values of an object's state
    public User(String productName, int amount, String expiryDay)
    {
        this.productName=productName;
        this.amount=amount;
        this.expiryDay=expiryDay;
    }

    //Accessors - return the values of an object's state
    //Purpose is for selected external classes to access these private variables
    public String getproductName(){
        return productName;
    }
    public int getamount(){
        return amount;
    }
    public String getexpiryDay(){
        return expiryDay;
    }

}
```

```
//Instantiates the column headers of the 'recipes' table from SQLite database
public class Recipe {

    private String mealName; //name of the meal
    private String recipe; //the recipes/instructions on how to make the meal
    private String ingredients; //what ingredients is needed to make the meal
    private String image; //an image of the meal itself
    private String mealType; //what food category does the meal fall in?

    //Mutators - sets the values of an object's state
    public Recipe(String mealName, String recipe, String ingredients,
            String image, String mealType) {
        this.mealName = mealName;
        this.recipe = recipe;
        this.ingredients = ingredients;
        this.image = image;
        this.mealType = mealType;
    }

    //Accessors - return the values of an object's state
    //Purpose is for selected external classes to access these private variables
    public String getmealName() {
        return mealName;
    }

    public String getrecipe() {
        return recipe;
    }

    public String getingredients() {
        return ingredients;
    }

    public String getimage() {
        return image;
    }

    public String getmealtype() {
        return mealType;
    }
}
```

```java
//Instantiates the column headers of the 'shopping' table from SQLite database
class Shopping {

    //out-of-stock and expired products that client can
    //choose from to be inputted into her shopping list
    private String product;

    //Mutators - sets the values of an object's state
    public Shopping(String product) {
        this.product = product;
    }

    //Accessors - return the values of an object's state
    //Purpose is for selected external classes to access these private variables
    public String getproduct() {
        return product;
    }

}
```

5) Design and Implementation of Swing GUI

Construction of jTable for SQLite Database:

```
try {
    Class.forName("org.sqlite.JDBC");
    c = DriverManager.getConnection("jdbc:sqlite:project.db");
    c.setAutoCommit(false);
    Statement s = c.createStatement();

    //Programmatically setting up properties/dimensions/appearance of jTable in GUI
    TableColumnModel columnModel = jTable_Display_User.getColumnModel();
    columnModel.getColumn(0).setPreferredWidth(1200);
    columnModel.getColumn(1).setPreferredWidth(400);
    columnModel.getColumn(2).setPreferredWidth(400);
    jTable_Display_User.setRowHeight(50);

    //Making column headers aligned in the centre
    TableCellRenderer rendererFromHeader = jTable_Display_User.getTableHeader().getDefaultRenderer();
    JLabel headerLabel = (JLabel) rendererFromHeader;
    headerLabel.setHorizontalAlignment(JLabel.CENTER);

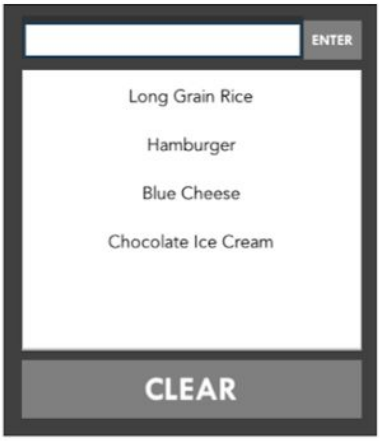
    //Setting the font of the column header
    jTable_Display_User.getTableHeader().setFont(new Font("Courier New", Font.BOLD, 16));

    //Setting dimensions/size of headers of jTable
    JTableHeader th = jTable_Display_User.getTableHeader();
    th.setPreferredSize(new Dimension(40, 40));

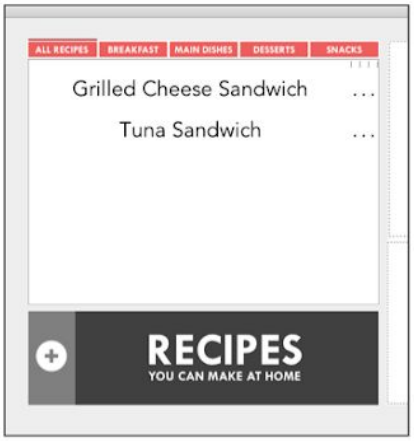
    //Retrieving ALL data from 'kitchen' table in SQLite database
    //so can be populated into jTable
    String querySelect = "SELECT * FROM kitchen";
    ResultSet rs = s.executeQuery(querySelect);
    User user;
    while (rs.next()) {
        user = new User(rs.getString("productName"), rs.getInt("amount"), rs.getString("expiryDay"));
        usersList.add(user);

    }
```

jTable to display SQLite database was created programatically into GUI



jTable in ShopListJFrame class

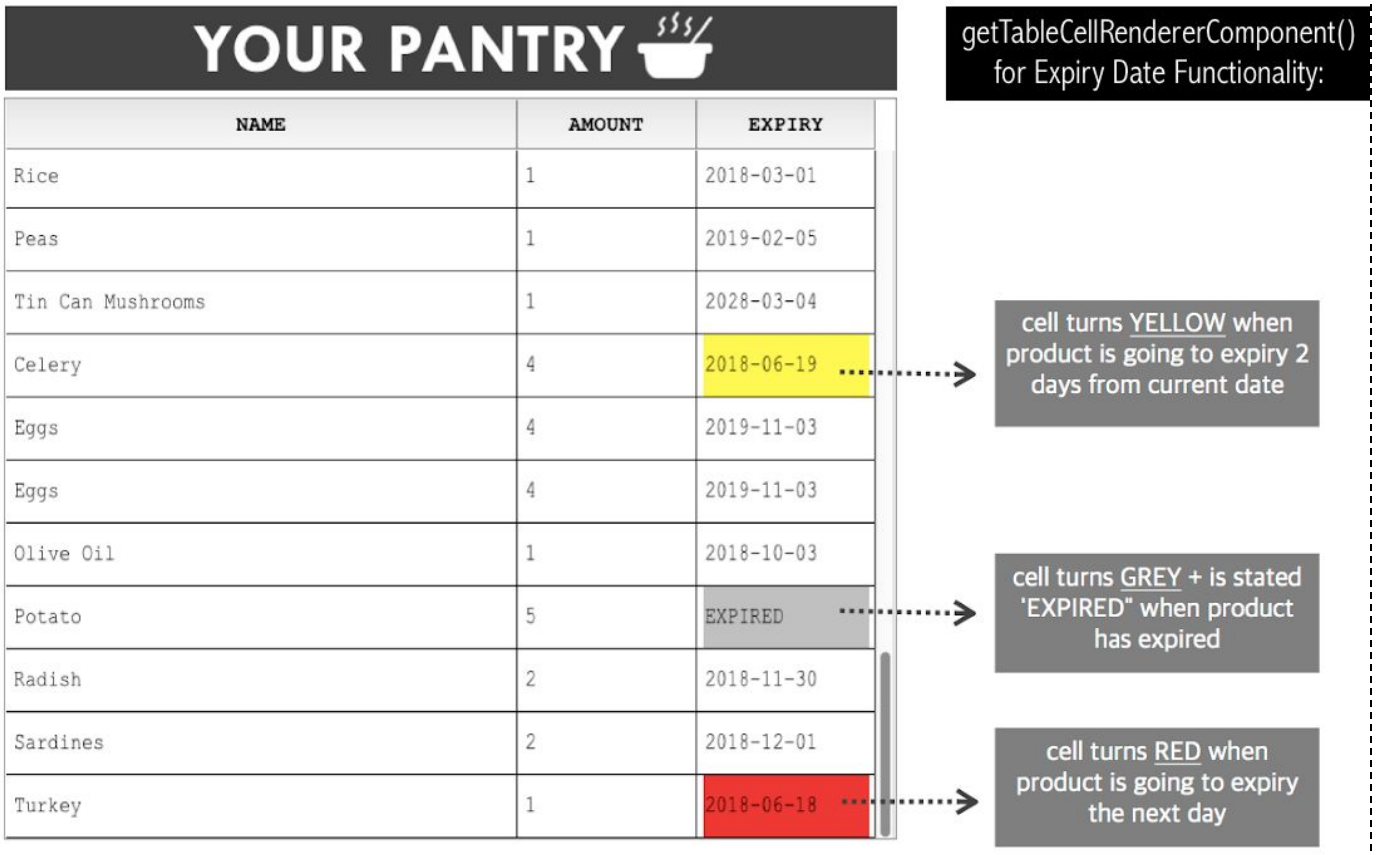


jTable in RecipeJFrame class



jTable in PantryJFrame class

The TableColorCellRenderer() class is used to colour specific cells/rows/columns of a jTable. This was implemented with the Expiry date warning system, by changing the colour of the rows containing products which are going to expiry/or has expired already.



The getTableCellRendererComponent() function (which is found in the TableColorCellRenderer class) was called on from the PantryJFrame class, as shown below:

```
TableColorCellRenderer renderer = new TableColorCellRenderer();
            jTable_Display_User.setDefaultRenderer(Object.class, renderer);
```

```java
public class TableColorCellRenderer implements TableCellRenderer {

    private static final TableCellRenderer RENDERER = new DefaultTableCellRenderer();

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value,
            boolean isSelected, boolean hasFocus, int row, int column) {
        Component c = RENDERER.getTableCellRendererComponent(table, value, isSelected, hasFocus, row, column);

        if (column == 2) { //expiry data's column location
            Object result = table.getModel().getValueAt(row, column);
            String average = result.toString();
            Color color = null;

            //current date
            LocalDateTime currentTime = LocalDateTime.now();

            //1 day after current date
            LocalDateTime pregame = currentTime.plusDays(1);

            //2 days after current date
            LocalDateTime game = currentTime.plusDays(2);
            DateTimeFormatter formatter2 = DateTimeFormatter.ofPattern("yyyy-MM-dd");

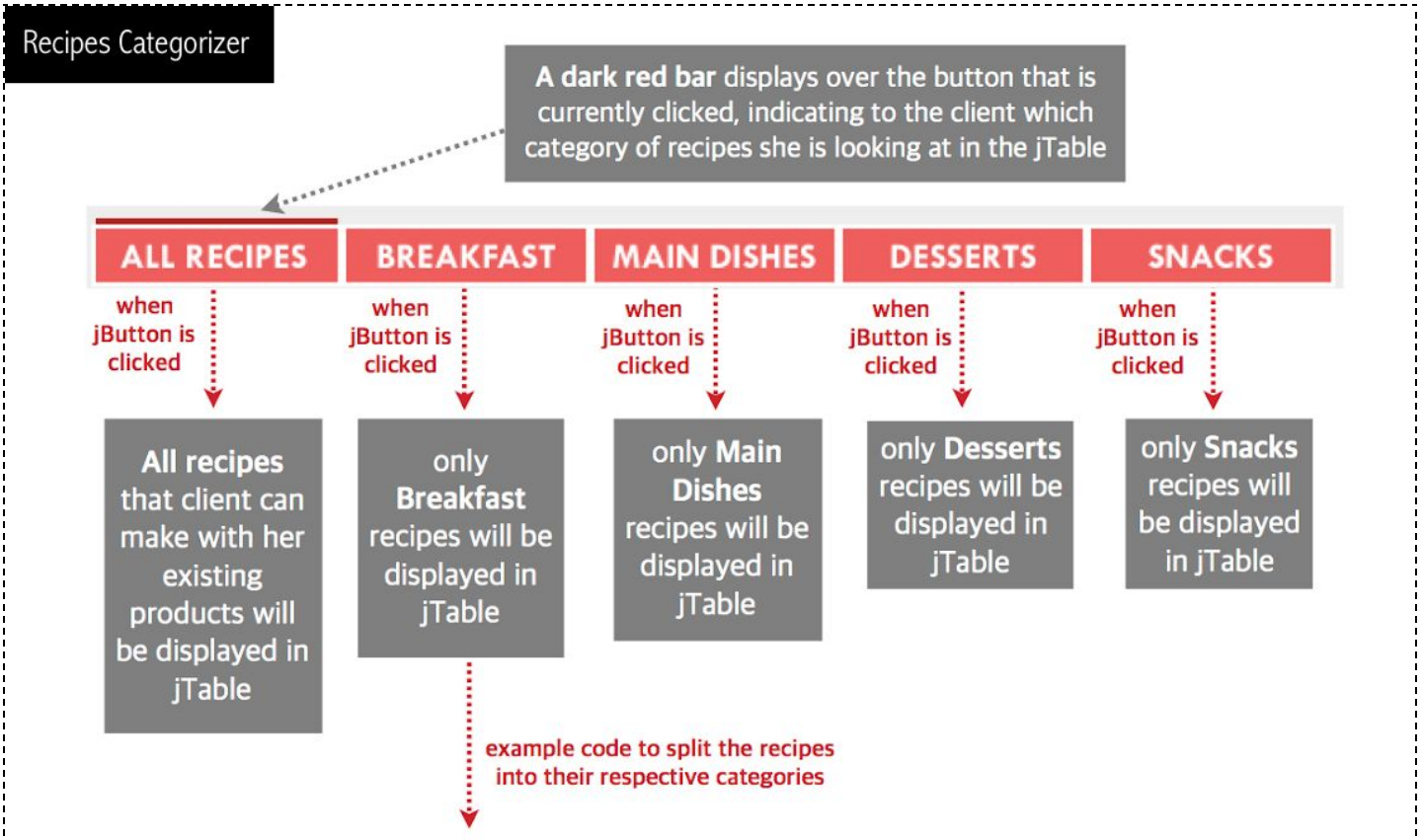
            String Edate1 = pregame.format(formatter2);
            String Edate2 = game.format(formatter2);
            String exd = "EXPIRED";
            String formatDateTime = currentTime.format(formatter2);

            //if any expiry dates (displayed in jTable) are 1 day after current date
            if (average.equalsIgnoreCase(Edate1)) {
                color = Color.RED;

            //if any expiry dates (displayed in jTable) are 2 days after current date
            } else if (average.equalsIgnoreCase(Edate2)) {
                color = Color.YELLOW;

            //if any expiry dates (displayed in JTable) is the current date
            } else if (average.equalsIgnoreCase(exd)) {
                color = Color.lightGray;
            } else {
                color = null;
            }
```

The client has the option to view the list of recipes based on categories. In order to achieve this, each recipe had to validated against each food category (ex. Breakfast, Main Dishes) and if it matches, then that specific recipe will stay displayed while others will be hidden




```
//a model of recipesjTable is instantiated
DefaultTableModel modelRecipes = (DefaultTableModel) this.recipesjTable.getModel();

//the number of rows of recipesjTable is retrieved
int count = modelRecipes.getRowCount();

//loop through all rows of recipesjTable
for (int i = 0; i < modelRecipes.getRowCount(); i++) {

    //if a certain meal is a "Breakfast" recipe
    //(if a certain row contains 'Breakfast' in their 3rd column)
    //then it will stay displayed in jtable
    if (((String) modelRecipes.getValueAt(i, 3)).equals("Breakfast")) {

    //if meal/recipe is of another category, then the row will be
    //removed from jTable and will not be displayed to client
    } else {
        //removes row
        modelRecipes.removeRow(i);
        i--;
    }

}
```


ALL RECIPES
BREAKFAST
MAIN DISHES
DESSERTS
SNACKS
Grilled Cheese Sandwich
Guacamole
Tuna Sandwich
Pasta Al Pomodoro Fresco
Chicken Fajitas
Banana Pudding
RECIPES
YOU CAN MAKE AT HOME
ALL RECIPES
BREAKFAST
MAIN DISHES
DESSERTS
SNACKS
Grilled Cheese Sandwich
ALL RECIPES
BREAKFAST
MAIN DISHES
DESSERTS
SNACKS
Pasta Al Pomodoro Fresco
Chicken Fajitas
ALL RECIPES
BREAKFAST
MAIN DISHES
DESSERTS
SNACKS
Banana Pudding
ALL RECIPES
BREAKFAST
MAIN DISHES
DESSERTS
SNACKS
Guacamole
Tuna Sandwich

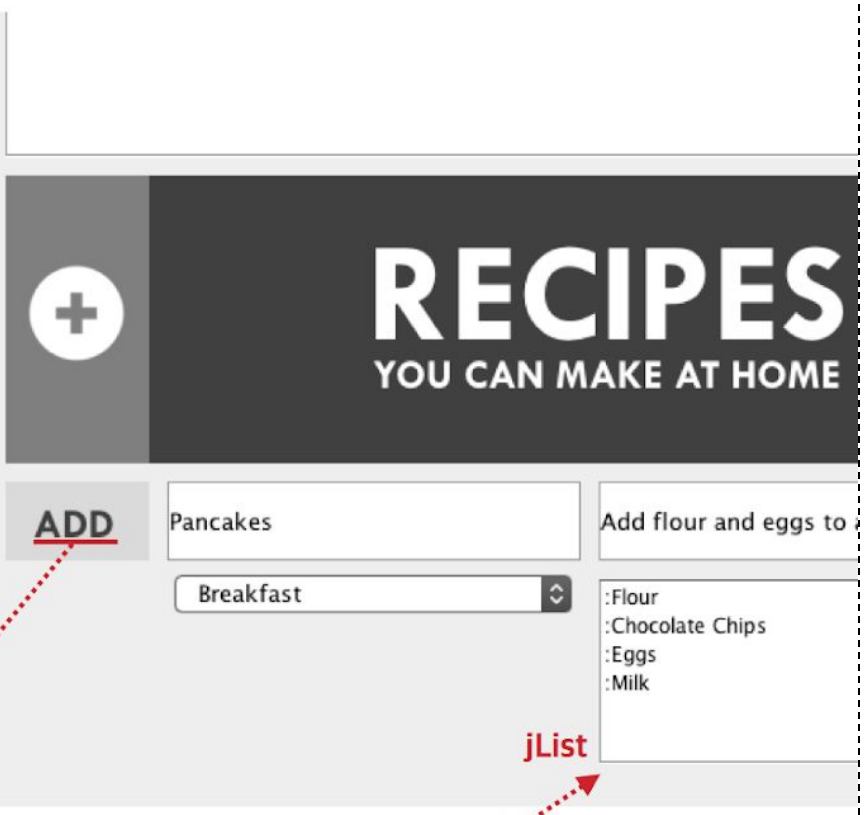
6) JSON Object and Array

Specifically for the recipe generator feature found in the RecipeJFrame class, an arraylist of ingredients for each meal was inputted, stored, and retrieved from the 'recipes' table in the SQLite database using JSON.

a) <u>INPUT of ingredients into JSON Array and stored in SQLite</u>:



Class: RecipeJFrame



when 'ADD' button is clicked . . .

jList

```
//new ArrayList for ingredients is instantiated
ArrayList<String> ing = new ArrayList<String>();

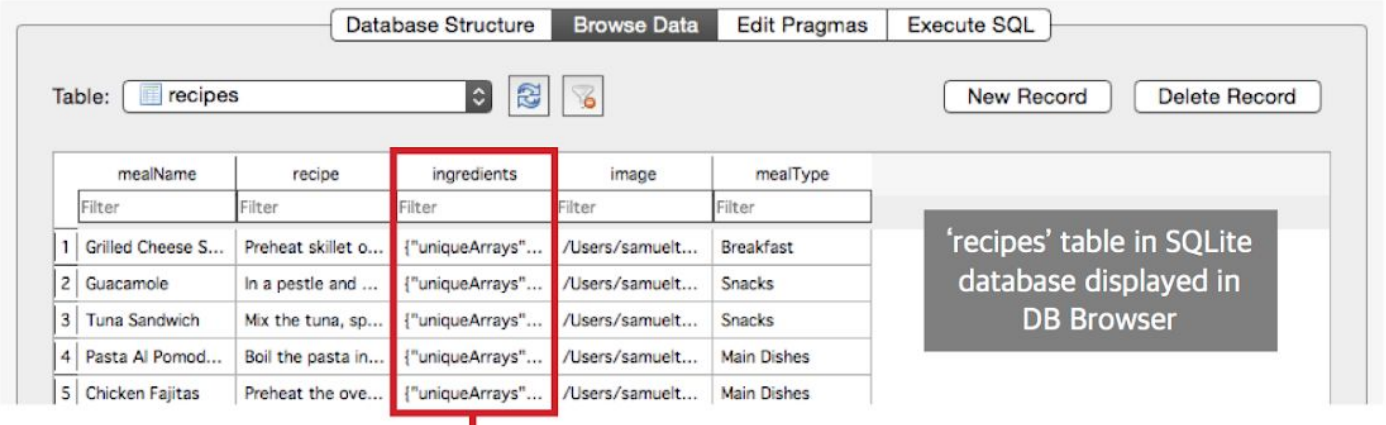
//retrieve all ingredients inputted into jList by client
//and store them into ArrayList 'ing'
for (int i = 0; i < ingInputjList.getModel().getSize(); i++) {
    ing.add((String) (ingInputjList.getModel().getElementAt(i)));
}

//new JSON object is instantiated
JSONObject json = new JSONObject();

//Arraylist 'ing' is converted into a JSON Array
json.put("uniqueArrays", new JSONArray(ing));

//JSON Array is stored into a string so it can be
//inputted into SQLite database
String ingJSONArray = json.toString();
```

```
s.executeUpdate("INSERT INTO recipes (mealName, recipe, ingredients,image, mealType)"
        + "values('" + userInput + "','" + recipeInput + "','" + ingJSONArray + "','" + imgP + "','" + category + "')");
```



example of JSON Array storing ingredients

```
{"uniqueArrays":[":Onions",":Red pepper",":Wholemeal tortillas",":Olive
oil",":Chicken breast",":Tomatoes",":Garlic"]}
```

b) RETRIEVAL of JSONArray to be validated against existing products

```
//retrieve the number id of row selected by client
int row = recipesjTable.getSelectedRow();

//Retrieve the selected JSONArray from the jTable from column 2
//and store it in a String
String ingJSONArray = (recipesjTable.getModel().getValueAt(row, 2).toString());

//Retrieved JSONArray of ingredients is stored into new
//JSON Object
JSONObject json = new JSONObject(ingJSONArray);

//Gets an optional JSONArray associated with a key
//returns null if there is no such key
JSONArray jArray = json.optJSONArray("uniqueArrays");

//loop through JSON Array of ingredients to store each
//ingredient into a string
for (int i = 0; i < jArray.length(); i++) {
    String str_value = jArray.optString(i);

    //Split each string by ":"
    String splitter = ":";
    String[] newValue = str_value.split(splitter);

    //Store new ingredient values into new List
    List<String> stringList = new ArrayList<String>(Arrays.asList(newValue));

    //Retrieve each ingredient from List 'stringList'
    String ing = stringList.get(1);

    //'SELECT' SQL statement to check if ingredient(s) exists in client's pantry
    ResultSet rot = s.executeQuery("SELECT EXISTS(SELECT 1 FROM kitchen "
            + "WHERE productName LIKE '" + ing + "')AS amount");

    while (rot.next()) {
        int ingredientExist = rot.getInt("amount");
        //if the ingredients exists in 'kitchen' table in SQLite database, the respective meal will
        //stay displayed in jTable
        if (ingredientExist == 1) {

        } else {
            //if any ingredients does not exist in 'kitchen' table, meal is removed from
            //jTable and will not be displayed
            DefaultTableModel model = (DefaultTableModel) this.recipesjTable.getModel();
            int[] rows = recipesjTable.getSelectedRows();
            for (int e = 0; e < rows.length; e++) {
                model.removeRow(rows[e] - e);
            }
            start--;
        }
    }
}
```

Justification: Due to the nature of SQLite, ArrayLists cannot be directly stored into the database. Approaches to achieve this included using JSON Array (as shown above), or by converting Java objects from the ArrayList into a storable transmittable format (ex. stream of bytes) and stored into the database as a BLOB. When retrieving a JSON array, it does not need to be converted back to have its stored values validated (unlike the stream of bytes where it has to convert it back) and thus it would be beneficial as it would go straight into a for loop. This means it can load data quicker.

7) External Libraries

a) Javax Mail:

In order for the client to access the shopping list she created in the ShopListJFrame class, the list must be sent to her via email. This was accomplished through the import of the JavaxMail library.





Class: ShopListJFrame

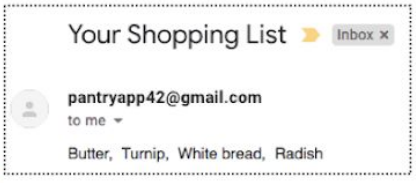
when 'EMAIL LIST' button is clicked . . .

```
String host = "smtp.gmail.com";
//Instantiates the Gmail account that is sending the shopping list
String user = "pantryapp42@gmail.com";
//Instantiates the password for 'user' gmail account to gain access
String pass = "prepare1";
//Instantiates the Gmail account that shopping list is being sent to
String to = "sonia.teh@gmail.com";
String from = "pantryapp42@gmail.com";
//Instantiates the subject of the mail being sent
String subject = "Your Shopping List";
//Instantiates the message of the mail being sent
String messageText = (String) jTextArea3.getText();
boolean sessionDebug = false;
Properties props = System.getProperties();
//Establishing the Simple Mail Transfer Protocol (smtp) connection
props.put("mail.smtp.starttls.enable", "true");
//the smtp host is gmail
props.put("mail.smtp.host", host);
props.put("mail.smtp.port", "587");
props.put("mail.smtp.auth", "true");
props.put("mail.smtp.starttls.required", "True");

java.security.Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
javax.mail.Session mailSession = javax.mail.Session.getDefaultInstance(props, null);
mailSession.setDebug(sessionDebug);
javax.mail.Message msg = new MimeMessage(mailSession);
msg.setFrom(new InternetAddress(from));
InternetAddress[] address = {new InternetAddress(to)};

//Sets all the information for the mail before proceeding to sending
msg.setRecipients(javax.mail.Message.RecipientType.TO, address);
msg.setSubject(subject);
msg.setSentDate(new Date());
msg.setText(messageText);

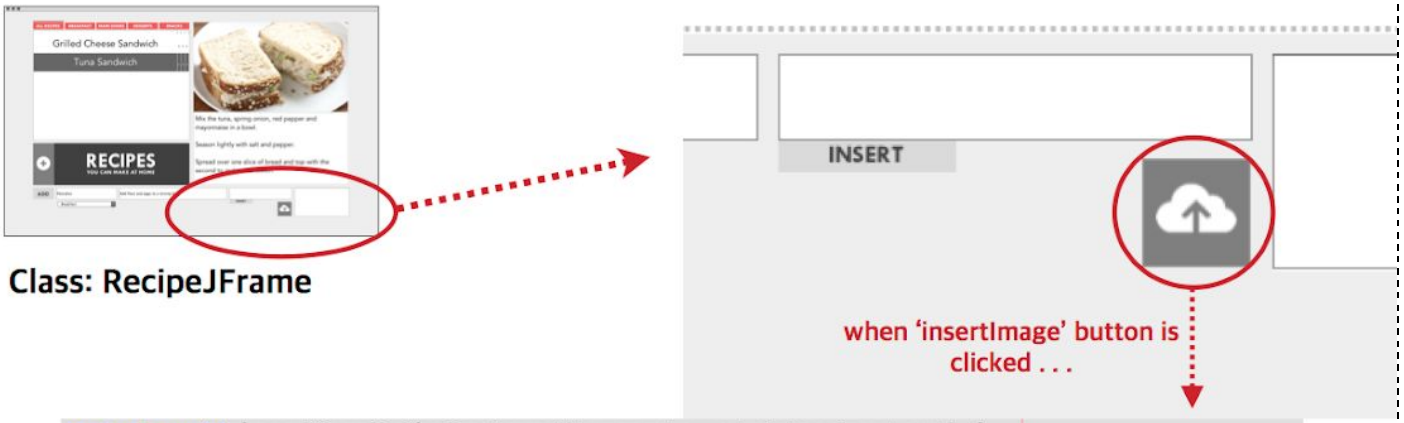
//Establish the connection between the gmail accounts and thus
//the message will be sent
Transport transport = mailSession.getTransport("smtp");
transport.connect(host, user, pass);
transport.sendMessage(msg, msg.getAllRecipients());
transport.close();
//indicates to the client if shopping list was successfully sent
int dialogButton = JOptionPane.YES_NO_OPTION;
//sets the image icon that will be displayed on jOptionPane
final ImageIcon icon3 = new ImageIcon(new URL(
        "file:///Users/samueltoi/Desktop/Screen%20Shot%202018-02-18%20at%2012.22.05%20am.png"));
//sets what will be displayed to client on jOptionPane
int dialogResult = JOptionPane.showConfirmDialog(this, "[Completed] Your shopping list has been sent "
        + "to your gmail. Would you like to make a new list?", "Notice",
        dialogButton, JOptionPane.YES_NO_OPTION, icon3);
```





b) JFileChooser:

Inside the RecipeJFrame class, when adding her own meal, client can also choose an image of the meal from her computer to upload. This is accomplished through JFileChooser.




```
private void insertImageActionPerformed(java.awt.event.ActionEvent evt) {
    //JFileChooser is part of the Java Swing package which allows client to select files
    //new JFileChooser file is instantiated
    JFileChooser file = new JFileChooser();

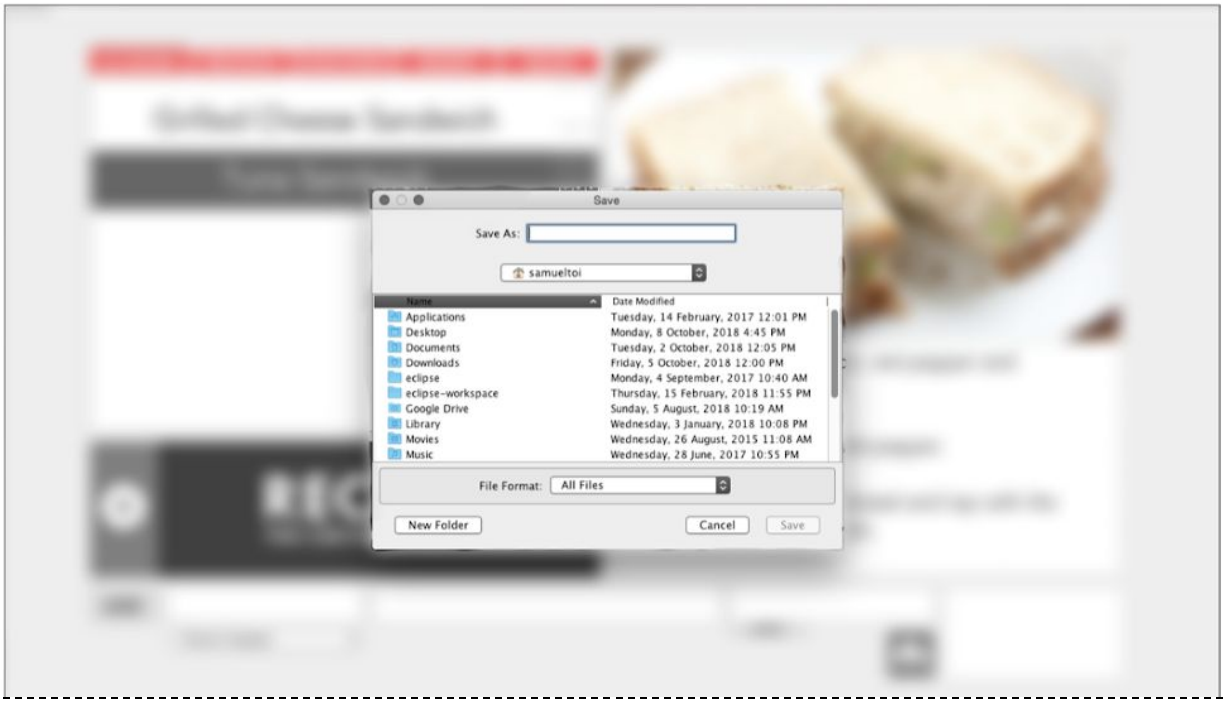
    //Sets the computer directory from which files can be accessed and selected from
    file.setCurrentDirectory(new File(System.getProperty("user.home")));

    //filters the type of files client can select
    FileNameExtensionFilter filter = new FileNameExtensionFilter("*.Images", "jpg", "gif", "png");
    file.addChoosableFileFilter(filter);
    int result = file.showSaveDialog(null);

    //When user saves selected image file, image path is retrieved
    if (result == JFileChooser.APPROVE_OPTION) {
        File selectedFile = file.getSelectedFile();
        String path = selectedFile.getAbsolutePath();
        //icon of jLabel 'imageDisplay'is set to image path
        imageDisplay.setIcon(ResizeImage(path));
        //Store the retrieved image path into a textfield
        //to be inserted into database later on
        imagePath.setText(path);

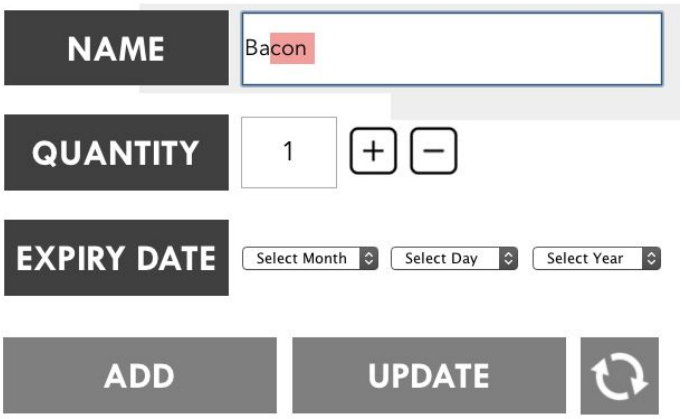
        //client can cancel from selecting a file
    } else if (result == JFileChooser.CANCEL_OPTION) {
        System.out.println("No File Select");
    }

}
```



8) TextField Auto-Complete

Whenever the client is required to input a value (ex.product name, recipe ingredients, groceries into shopping list) a text field autocomplete algorithm was implemented.









dict.txt is a text file containing a large library of words associated with food

```
//a new Arraylist of type String is instantiated for the purpose of storing
//words from dict.txt
private ArrayList<String> listSomeString = new ArrayList<String>();

//an instance of the class 'Java2sAutoTextField' is created
private Java2sAutoTextField someTextField
        = new Java2sAutoTextField(listSomeString);
```

Java2sAutoTextField is the class that holds all the actual methods for this auto-complete feature

```
//Retrieve dict.txt file from computer using it's file path
File file = new File("/Users/samueltoi/Desktop/dict.txt");
try {
    //Reads the file and adds every single word present in the
    //txt file into an ArrayList
    BufferedReader br = new BufferedReader(new FileReader(file));
    String word;
    while ((word = br.readLine()) != null) {
        listSomeString.add(word);
    }

    //catches any errors/exceptions that arises from reading the
    //dict.txt file from the computer
} catch (Exception e) {

}
```

Justification: The auto-complete suggested words were initially going to be stored inside an ArrayList, but realised it would be inefficient, considering the large quantities of words needed to stored. Thus, the list of words were instead kept inside a txt. file , and through using BufferedReader (as shown above) to retrieve them, it made it more practical and efficient.

It consists of **3 main methods** (all found in class Java2sAutoTextField) :

METHOD 1

```
//finding a match to the word client is inputting
private String getMatch(String s) {
    //loop through the whole list of words to find a match
    for (int i = 0; i < dataList.size(); i++) {
        //Retrieve each word into a string
        String s1 = dataList.get(i).toString();

        //Validates each word that is not empty/blank
        if (s1 != null) {
            //if client input is upper case, it changes to lower case to match
            //the auto-complete word
            if (!isCaseSensitive
                    && s1.toLowerCase().startsWith(s.toLowerCase())) {
                return s1;
            }
            //if client input completely matches an auto-complete word in
            //iterms of format (upper/lower case), then word is returned.
            if (isCaseSensitive && s1.startsWith(s)) {
                return s1;
            }
        }
    }

    return null;
}
```

METHOD 2

```
public Java2sAutoTextField(List list) {

    //if isCaseSensitive = false, means client can type either with caps on or off, and
    //auto-complete would still work and suggest the words
    //if true, means client must only type with lower case to allow auto-complete
    //to work
    isCaseSensitive = false;

    //if isStrict = false, means client can continue to type words even if
    //it does not match any auto-complete words
    //if true, means client must only type words/letters that match the
    //auto-complete words
    isStrict = false;

    //Error Check:
    //check if list is empty and there is no words present in it
    if (list == null) {
        throw new IllegalArgumentException("values cannot be null");
    } else {
        dataList = list;
        init();
        return;
    }
}
```

METHOD 3

```
    //method for inserting the new auto-complete word
    public void insertString(int i, String s, AttributeSet attributeset)
            throws BadLocationException {

        //checks if client has not inputted any letter/word
        if (s == null || "".equals(s)) {
            return;
        }

        //Retrieves the word/letter getting inputted by client
        String s1 = getText(0, i);
        //Retrieves the matching word to the word inputted by client
        String s2 = getMatch(s1 + s);
        //get letter count
        int j = (i + s.length()) - 1;

        //ERROR check - if matching word is empty
        if (isStrict && s2 == null) {
            s2 = getMatch(s1);
            j--;

        //auto-complete suggestion is inputed
        } else if (!isStrict && s2 == null) {
            super.insertString(i, s, attributeset);
            return;
        }

        //resets after each letter is inputted
        //so the auto-suggestion can suggest the next closest word
        super.remove(0, getLength());
        super.insertString(0, s2, attributeset);
        setSelectionStart(j + 1);
        setSelectionEnd(getLength());

    }

}
```

Mutators/accessors were also implemented so methods can be accessed by outside classes (ex. PantryJFrame, RecipeJFrame, ShopListJFrame) that uses this textfield autocomplete feature.

```
    //Accessor
    public boolean isCaseSensitive() {
        return isCaseSensitive;
    }

    //Mutator
    public void setCaseSensitive(boolean flag) {
        isCaseSensitive = flag;
    }

    //Accessor
    public boolean isStrict() {
        return isStrict;
    }

    //Mutator
    public void setStrict(boolean flag) {
        isStrict = flag;
    }

    //Accessor
    public List getDataList() {
        return dataList;
    }

    //Mutator
    public void setDataList(List list) {
        //Error Check:
        //check if list is empty and there is no words present in it
        if (list == null) {
            throw new IllegalArgumentException("values cannot be null");
        } else {
            //Store the list of words into dataList
            dataList = list;
            return;
        }
    }
```

References:

https://stackoverflow.com/

www.sqlitetutorial.net

https://www.tutorialspoint.com

WORD COUNT: 1208