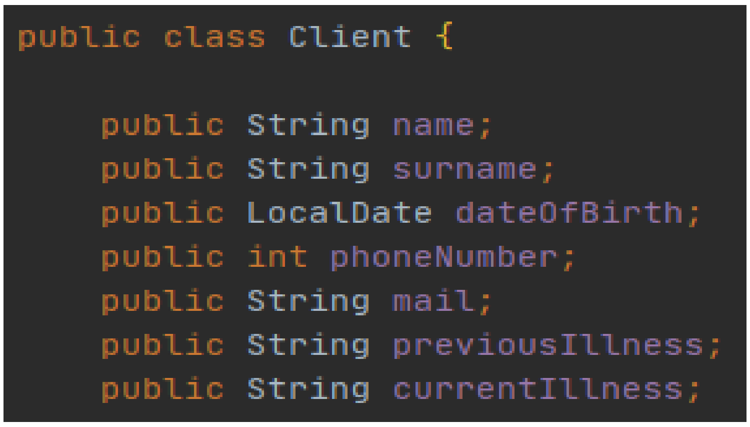
Criterion C: Development

Used Frontend/Backend dependencies and techniques used:

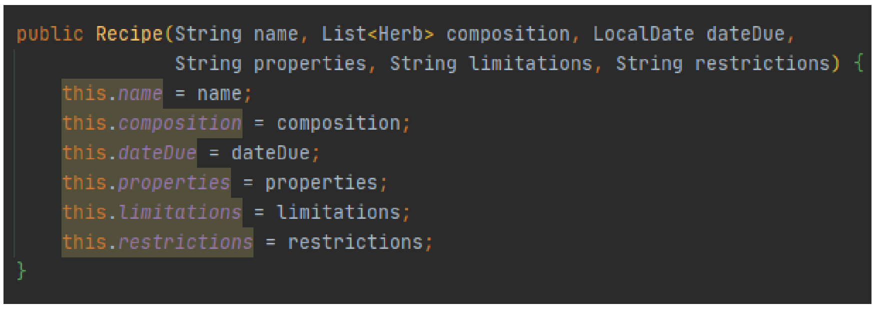
- POJO classes
- Deleting/adding data from/to SQL database
- “Assigning methods”
- Java libraries
- Displaying tables with sets of data
- For loops
- If statements/if else statements
- Nested loops
- Parallel arrays
- Try-catch statements
- @FXML annotation and Scene Builder
- Initialization
- Charts and series
- Calculation and sorting methods
- CSS stylesheets
- Personalized algorithms

POJO Classes

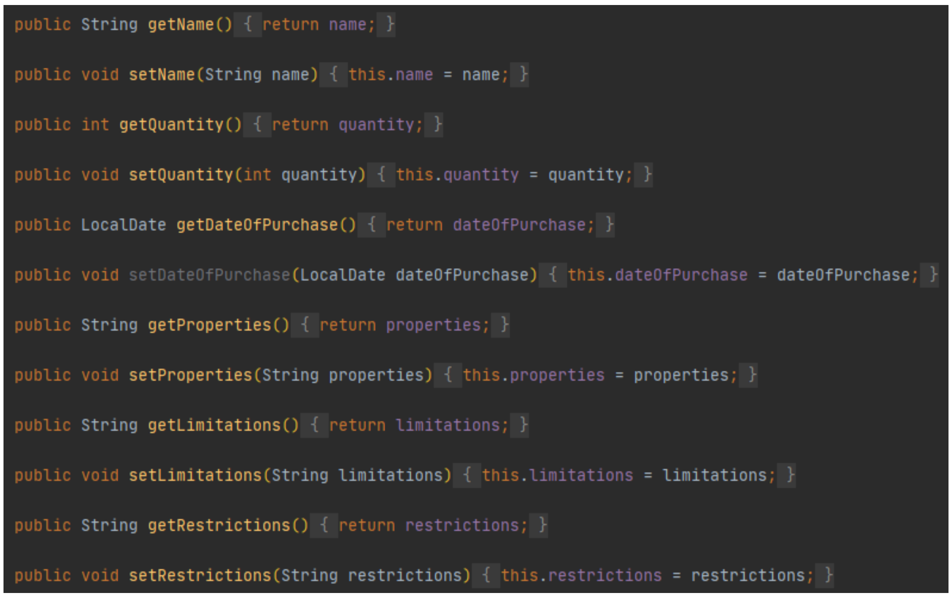
POJO classes were crucial for creation of this software. Classes such as “Client” use variables, getters, setters and toString methods which are used to modify or insert values of said variables. They were used to ease uploading or reading objects from the database.



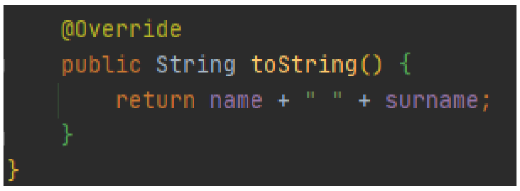
```java
public class Client {

    public String name;
    public String surname;
    public LocalDate dateOfBirth;
    public int phoneNumber;
    public String mail;
    public String previousIllness;
    public String currentIllness;
```



```java
public Recipe(String name, List<Herb> composition, LocalDate dateDue,
              String properties, String limitations, String restrictions) {
    this.name = name;
    this.composition = composition;
    this.dateDue = dateDue;
    this.properties = properties;
    this.limitations = limitations;
    this.restrictions = restrictions;
}
```



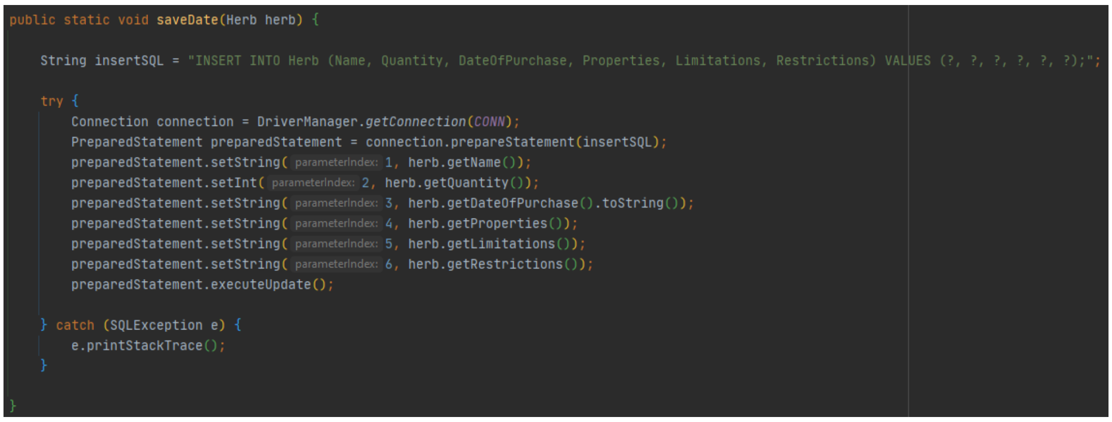
```java
public String getName() { return name; }

public void setName(String name) { this.name = name; }

public int getQuantity() { return quantity; }

public void setQuantity(int quantity) { this.quantity = quantity; }

public LocalDate getDateOfPurchase() { return dateOfPurchase; }

public void setDateOfPurchase(LocalDate dateOfPurchase) { this.dateOfPurchase = dateOfPurchase; }

public String getProperties() { return properties; }

public void setProperties(String properties) { this.properties = properties; }

public String getLimitations() { return limitations; }

public void setLimitations(String limitations) { this.limitations = limitations; }

public String getRestrictions() { return restrictions; }

public void setRestrictions(String restrictions) { this.restrictions = restrictions; }
```



```java
    @Override
    public String toString() {
        return name + " " + surname;
    }
}
```

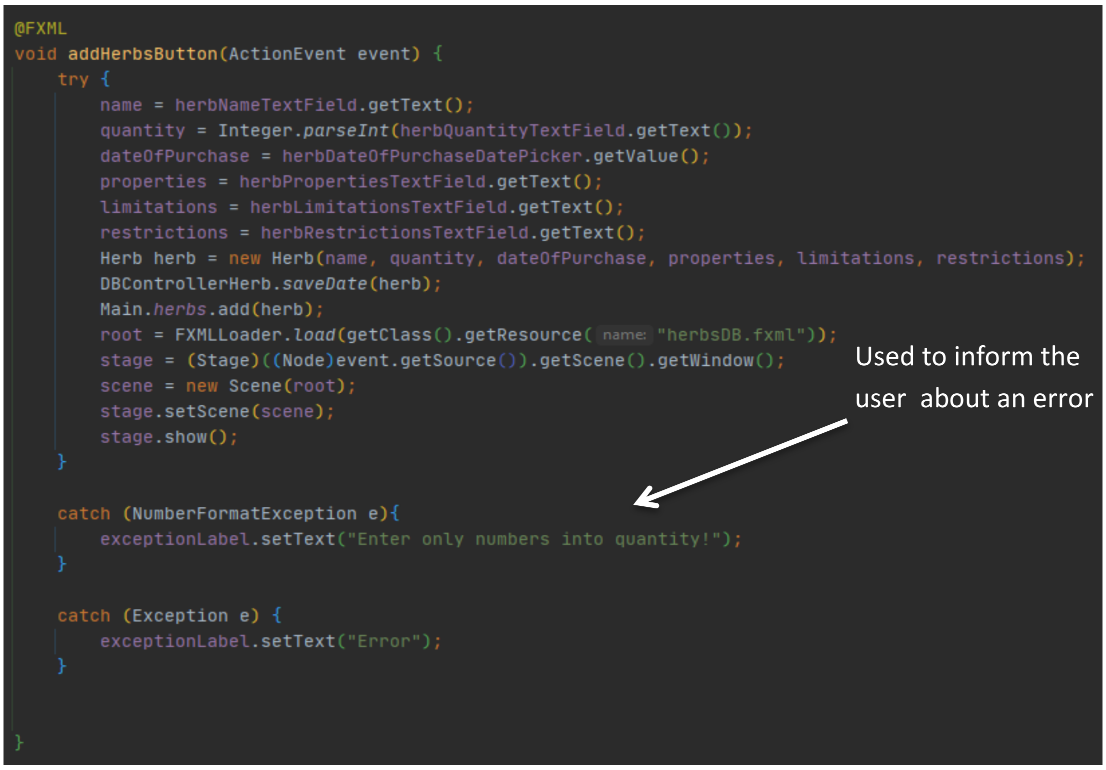
Adding data to the SQL database

This method (saveDate) enables the user to add new Herbs into the database. The name, quantity, date of purchase, properties, limitations and restrictions are input into the database with use of the SQL statement “INSERT INTO”. With use of this method new Herb is created with variables from Herbs class. In some cases the types of values are formatted e.g. “dateOfBirth”. It is changed from LocalDate into a String in order to be stored in the database.



```java
public static void saveDate(Herb herb) {

    String insertSQL = "INSERT INTO Herb (Name, Quantity, DateOfPurchase, Properties, Limitations, Restrictions) VALUES (?, ?, ?, ?, ?, ?);";

    try {
        Connection connection = DriverManager.getConnection(CONN);
        PreparedStatement preparedStatement = connection.prepareStatement(insertSQL);
        preparedStatement.setString( parameterIndex: 1, herb.getName());
        preparedStatement.setInt( parameterIndex: 2, herb.getQuantity());
        preparedStatement.setString( parameterIndex: 3, herb.getDateOfPurchase().toString());
        preparedStatement.setString( parameterIndex: 4, herb.getProperties());
        preparedStatement.setString( parameterIndex: 5, herb.getLimitations());
        preparedStatement.setString( parameterIndex: 6, herb.getRestrictions());
        preparedStatement.executeUpdate();

    } catch (SQLException e) {
        e.printStackTrace();
    }

}
```

This happens thanks to the “addHerbsButton” which is visible below. This method gathers the data from Text-Fields and Data-Pickers and if it fits the criteria, a new Herb is created in the database with input values and the scene is changed to the “HerbsDB” class in which it is visible. Try-catch statement is used to prevent any miss input such as Strings in Text-Fields dedicated to Integers.

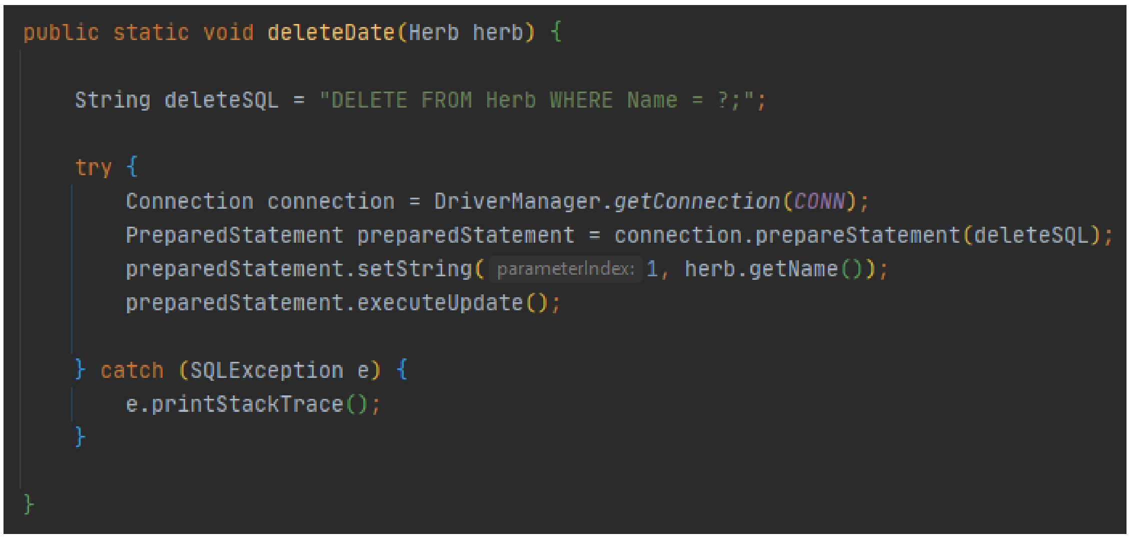


```java
@FXML
void addHerbsButton(ActionEvent event) {
    try {
        name = herbNameTextField.getText();
        quantity = Integer.parseInt(herbQuantityTextField.getText());
        dateOfPurchase = herbDateOfPurchaseDatePicker.getValue();
        properties = herbPropertiesTextField.getText();
        limitations = herbLimitationsTextField.getText();
        restrictions = herbRestrictionsTextField.getText();
        Herb herb = new Herb(name, quantity, dateOfPurchase, properties, limitations, restrictions);
        DBControllerHerb.saveDate(herb);
        Main.herbs.add(herb);
        root = FXMLLoader.load(getClass().getResource( name: "herbsDB.fxml"));
        stage = (Stage)((Node)event.getSource()).getScene().getWindow();
        scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }

    catch (NumberFormatException e){
        exceptionLabel.setText("Enter only numbers into quantity!");
    }

    catch (Exception e) {
        exceptionLabel.setText("Error");
    }


}
```

Used to inform the user about an error

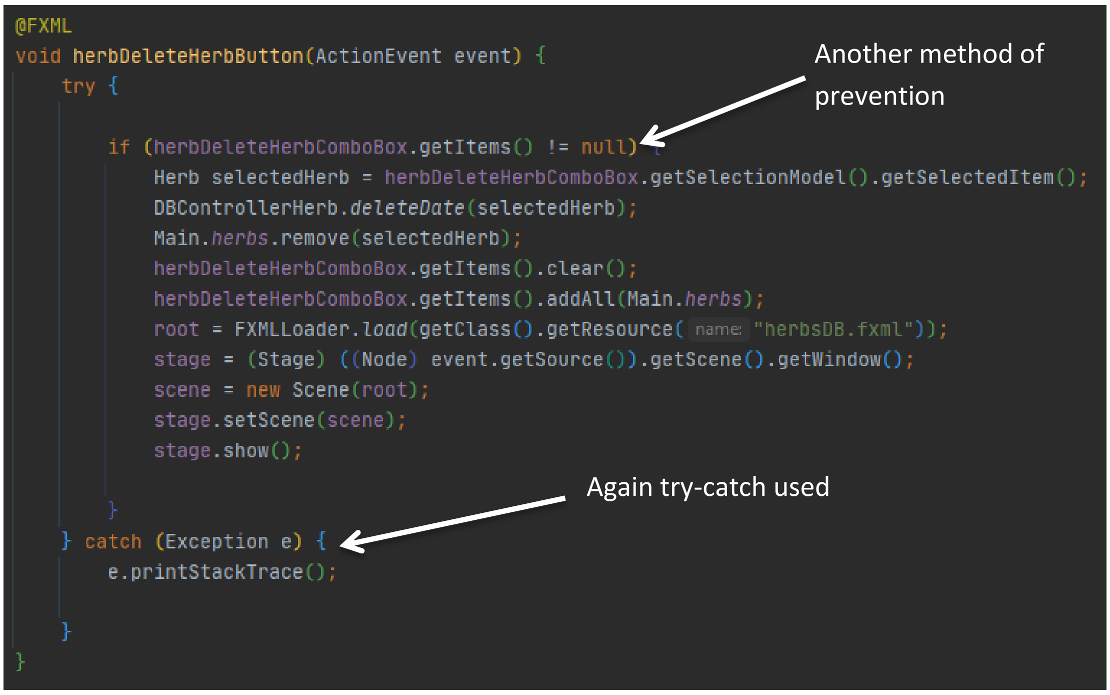
Deleting data from the SQL database

This method (deleteDate) enables the user to delete Herbs from the database. Herb as an object and all its data is deleted from the database with use of the SQL statement “DELETE FROM” only with use of its name.



```java
public static void deleteDate(Herb herb) {

    String deleteSQL = "DELETE FROM Herb WHERE Name = ?;";

    try {
        Connection connection = DriverManager.getConnection(CONN);
        PreparedStatement preparedStatement = connection.prepareStatement(deleteSQL);
        preparedStatement.setString( parameterIndex: 1, herb.getName());
        preparedStatement.executeUpdate();

    } catch (SQLException e) {
        e.printStackTrace();
    }

}
```

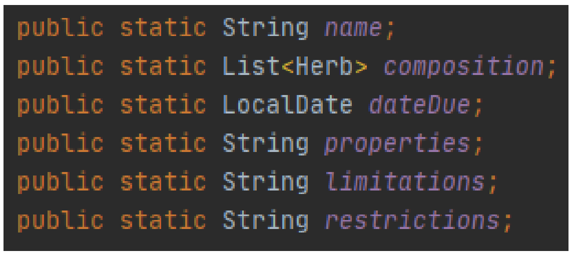
This happens thanks to the “deleteHerbsButton” which is visible below. This method gathers the data from Combo-Box with names of Herbs and after one is selected, it is deleted both from the table of Herbs and database with use of (deleteHerb).



```java
@FXML
void herbDeleteHerbButton(ActionEvent event) {
    try {

        if (herbDeleteHerbComboBox.getItems() != null) {
            Herb selectedHerb = herbDeleteHerbComboBox.getSelectionModel().getSelectedItem();
            DBControllerHerb.deleteDate(selectedHerb);
            Main.herbs.remove(selectedHerb);
            herbDeleteHerbComboBox.getItems().clear();
            herbDeleteHerbComboBox.getItems().addAll(Main.herbs);
            root = FXMLLoader.load(getClass().getResource( name: "herbsDB.fxml"));
            stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
            scene = new Scene(root);
            stage.setScene(scene);
            stage.show();

        }
    } catch (Exception e) {
        e.printStackTrace();

    }
}
```

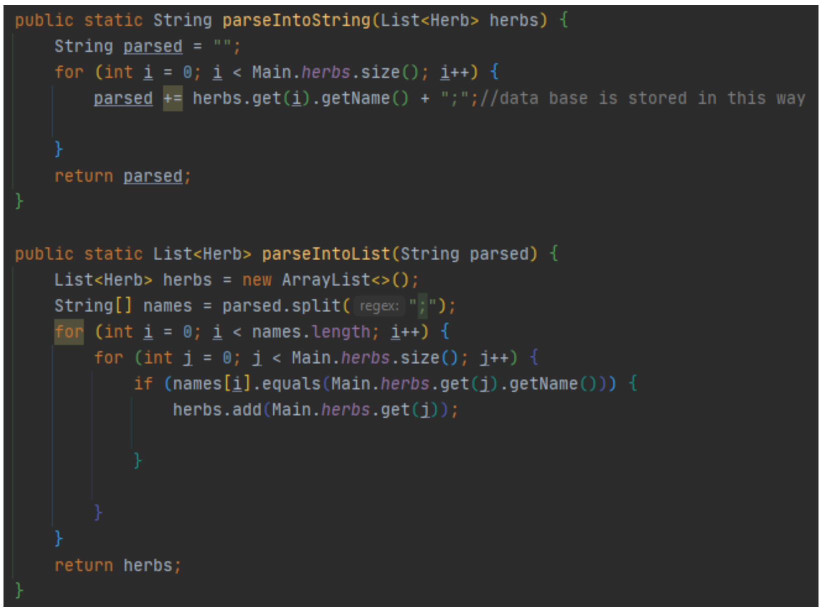
Adding a recipe

Recipes are a crucial matter in this software. They are composed of already existing herbs from the database. To utilize them composition list containing herbs was created.



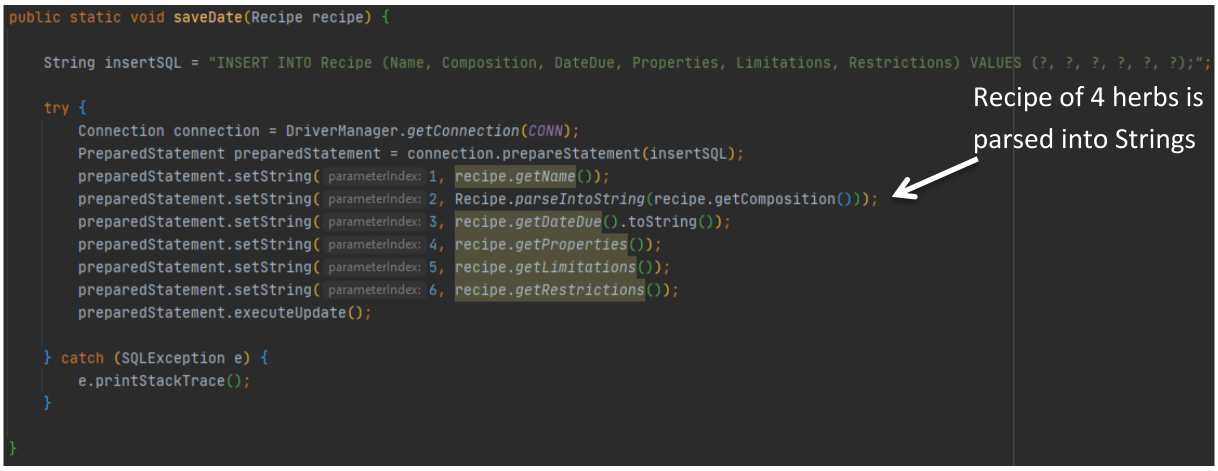
```java
public static String name;
public static List<Herb> composition;
public static LocalDate dateDue;
public static String properties;
public static String limitations;
public static String restrictions;
```

Then, with use of parsing methods visible below, one is able to parse the “4 herbs recipe” either into String when it is supposed to go into the database, or into a list when it is added into a table.



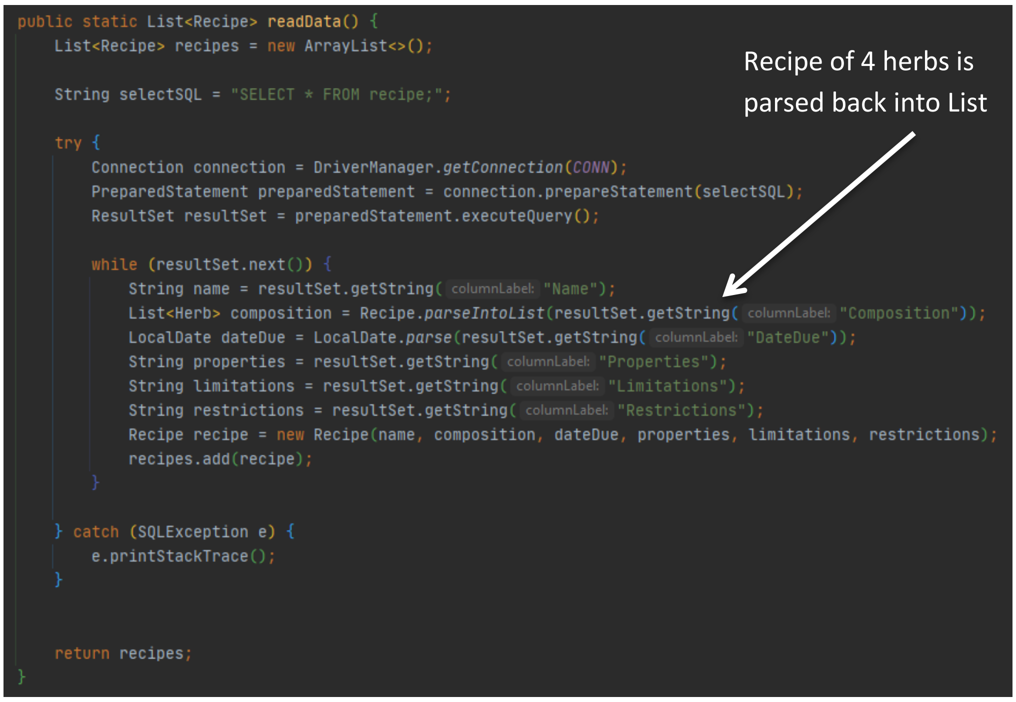
```java
public static String parseIntoString(List<Herb> herbs) {
    String parsed = "";
    for (int i = 0; i < Main.herbs.size(); i++) {
        parsed += herbs.get(i).getName() + ";";//data base is stored in this way

    }
    return parsed;
}

public static List<Herb> parseIntoList(String parsed) {
    List<Herb> herbs = new ArrayList<>();
    String[] names = parsed.split( regex: ";");
    for (int i = 0; i < names.length; i++) {
        for (int j = 0; j < Main.herbs.size(); j++) {
            if (names[i].equals(Main.herbs.get(j).getName())) {
                herbs.add(Main.herbs.get(j));

            }

        }
    }
    return herbs;
}
```

Example of this happening is visible whilst using the (saveDate) method that saves values into the database.



```java
public static void saveDate(Recipe recipe) {

    String insertSQL = "INSERT INTO Recipe (Name, Composition, DateDue, Properties, Limitations, Restrictions) VALUES (?, ?, ?, ?, ?, ?);";

    try {
        Connection connection = DriverManager.getConnection(CONN);
        PreparedStatement preparedStatement = connection.prepareStatement(insertSQL);
        preparedStatement.setString( parameterIndex: 1, recipe.getName());
        preparedStatement.setString( parameterIndex: 2, Recipe.parseIntoString(recipe.getComposition()));
        preparedStatement.setString( parameterIndex: 3, recipe.getDateDue().toString());
        preparedStatement.setString( parameterIndex: 4, recipe.getProperties());
        preparedStatement.setString( parameterIndex: 5, recipe.getLimitations());
        preparedStatement.setString( parameterIndex: 6, recipe.getRestrictions());
        preparedStatement.executeUpdate();

    } catch (SQLException e) {
        e.printStackTrace();
    }

}
```

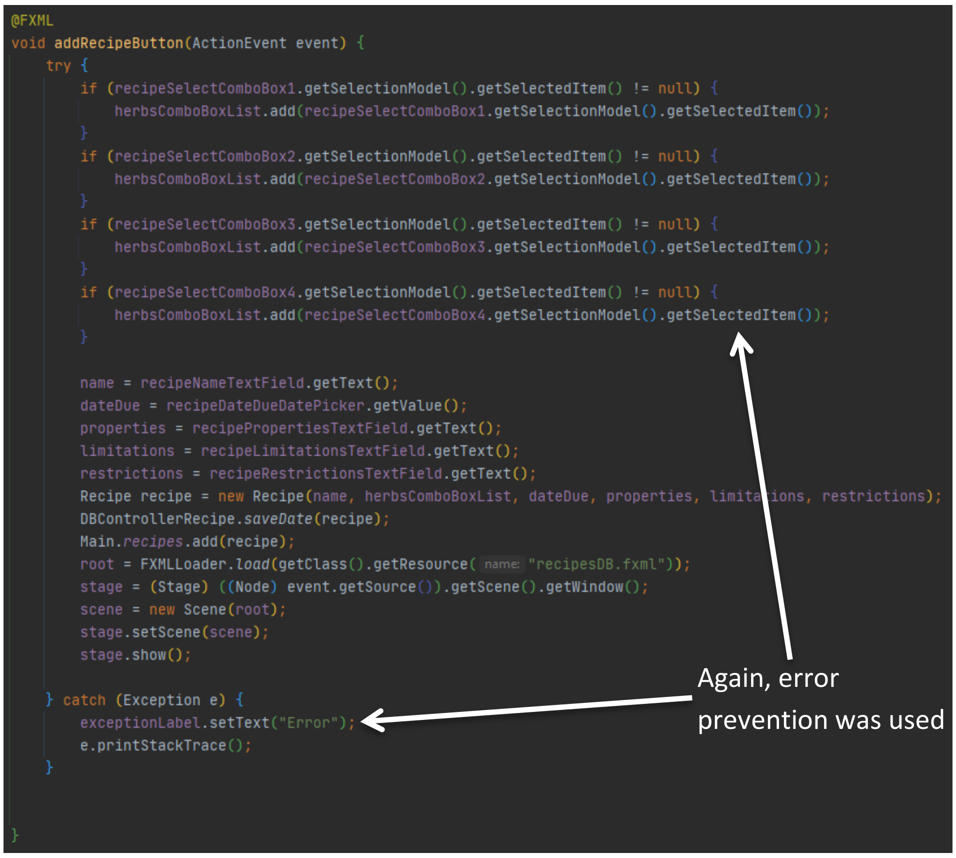
It can also happen the other way around while being read from the database into the table.



```
public static List<Recipe> readData() {
    List<Recipe> recipes = new ArrayList<>();

    String selectSQL = "SELECT * FROM recipe;";

    try {
        Connection connection = DriverManager.getConnection(CONN);
        PreparedStatement preparedStatement = connection.prepareStatement(selectSQL);
        ResultSet resultSet = preparedStatement.executeQuery();

        while (resultSet.next()) {
            String name = resultSet.getString( columnLabel: "Name");
            List<Herb> composition = Recipe.parseIntoList(resultSet.getString( columnLabel: "Composition"));
            LocalDate dateDue = LocalDate.parse(resultSet.getString( columnLabel: "DateDue"));
            String properties = resultSet.getString( columnLabel: "Properties");
            String limitations = resultSet.getString( columnLabel: "Limitations");
            String restrictions = resultSet.getString( columnLabel: "Restrictions");
            Recipe recipe = new Recipe(name, composition, dateDue, properties, limitations, restrictions);
            recipes.add(recipe);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }


    return recipes;
}
```

Recipe of 4 herbs is parsed back into List

This method is done with use of the “addRecipeButton” which collects the values, picked in Combo-Boxes (names of Herbs), creates a new object both in table and database and finally changes the scene to “recipiesDB” in which it is visible.

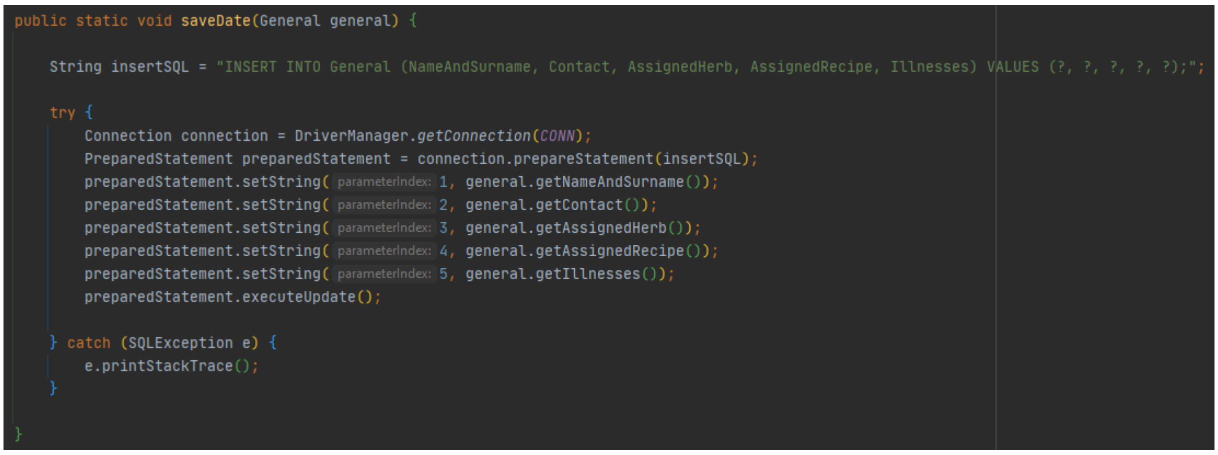


```
@FXML
void addRecipeButton(ActionEvent event) {
    try {
        if (recipeSelectComboBox1.getSelectionModel().getSelectedItem() != null) {
            herbsComboBoxList.add(recipeSelectComboBox1.getSelectionModel().getSelectedItem());
        }
        if (recipeSelectComboBox2.getSelectionModel().getSelectedItem() != null) {
            herbsComboBoxList.add(recipeSelectComboBox2.getSelectionModel().getSelectedItem());
        }
        if (recipeSelectComboBox3.getSelectionModel().getSelectedItem() != null) {
            herbsComboBoxList.add(recipeSelectComboBox3.getSelectionModel().getSelectedItem());
        }
        if (recipeSelectComboBox4.getSelectionModel().getSelectedItem() != null) {
            herbsComboBoxList.add(recipeSelectComboBox4.getSelectionModel().getSelectedItem());
        }

        name = recipeNameTextField.getText();
        dateDue = recipeDateDueDatePicker.getValue();
        properties = recipePropertiesTextField.getText();
        limitations = recipeLimitationsTextField.getText();
        restrictions = recipeRestrictionsTextField.getText();
        Recipe recipe = new Recipe(name, herbsComboBoxList, dateDue, properties, limitations, restrictions);
        DBControllerRecipe.saveDate(recipe);
        Main.recipes.add(recipe);
        root = FXMLLoader.load(getClass().getResource( name: "recipesDB.fxml"));
        stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
        scene = new Scene(root);
        stage.setScene(scene);
        stage.show();

    } catch (Exception e) {
        exceptionLabel.setText("Error");
        e.printStackTrace();
    }


}
```

Again, error prevention was used

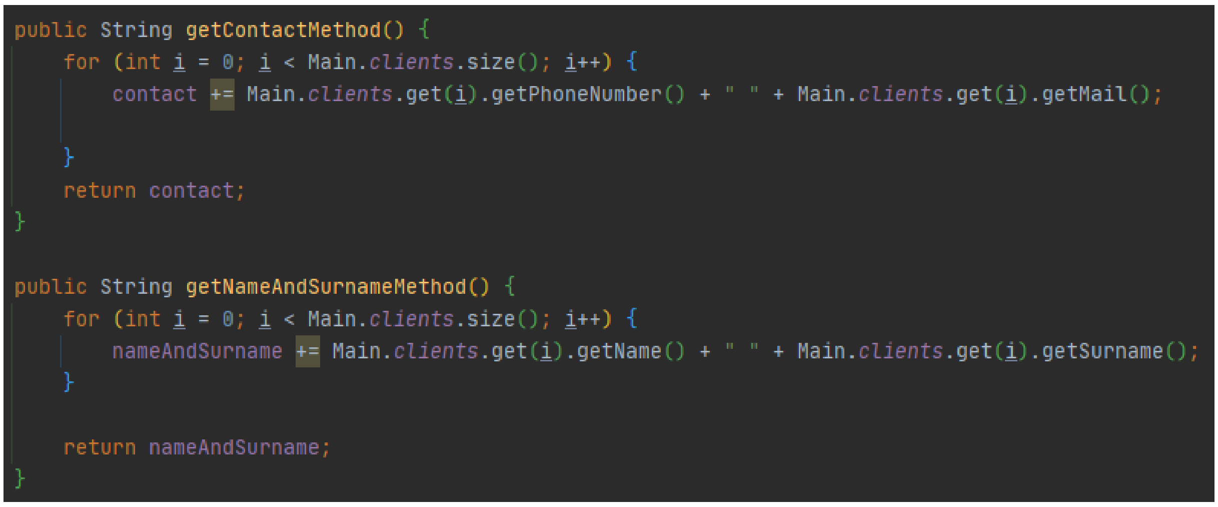
Assigning methods

This method works on the same basis as the aforementioned one (adding), but it uses data from three other Classes (Herbs, Recipes, Clients).



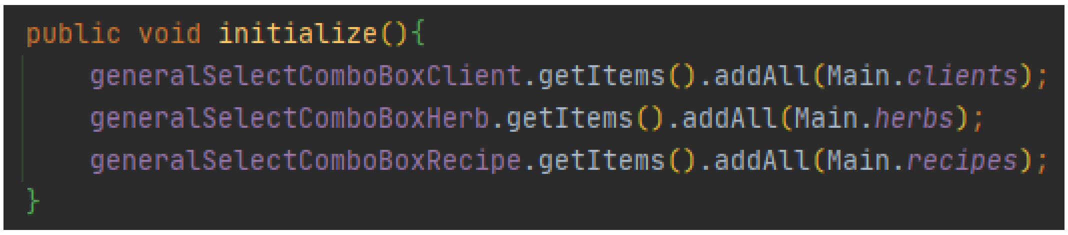
```java
public static void saveDate(General general) {

    String insertSQL = "INSERT INTO General (NameAndSurname, Contact, AssignedHerb, AssignedRecipe, Illnesses) VALUES (?, ?, ?, ?, ?);";

    try {
        Connection connection = DriverManager.getConnection(CONN);
        PreparedStatement preparedStatement = connection.prepareStatement(insertSQL);
        preparedStatement.setString( parameterIndex: 1, general.getNameAndSurname());
        preparedStatement.setString( parameterIndex: 2, general.getContact());
        preparedStatement.setString( parameterIndex: 3, general.getAssignedHerb());
        preparedStatement.setString( parameterIndex: 4, general.getAssignedRecipe());
        preparedStatement.setString( parameterIndex: 5, general.getIllnesses());
        preparedStatement.executeUpdate();

    } catch (SQLException e) {
        e.printStackTrace();
    }

}
```

Below, one can see that new Strings that use a different class was created.



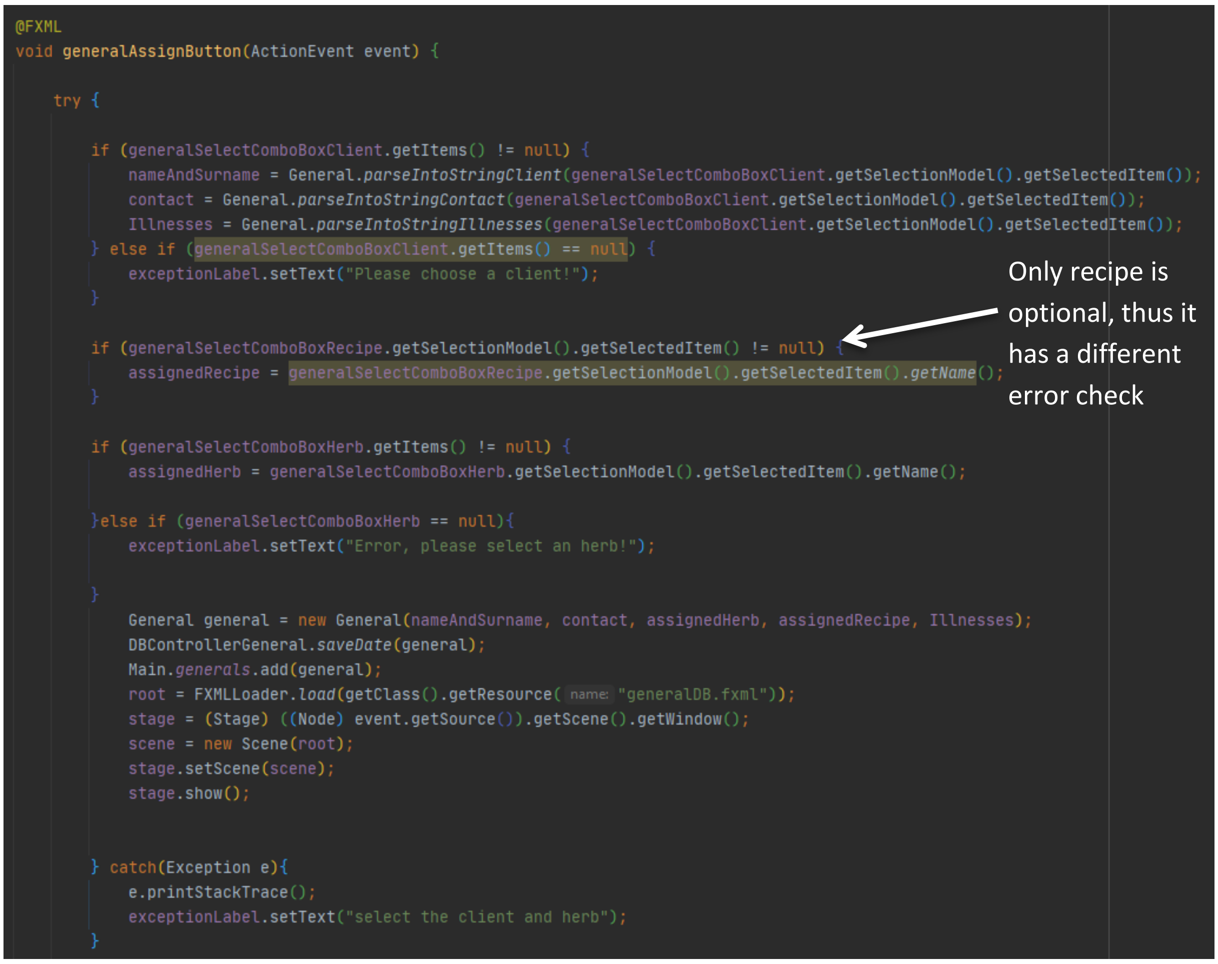
```java
public String getContactMethod() {
    for (int i = 0; i < Main.clients.size(); i++) {
        contact += Main.clients.get(i).getPhoneNumber() + " " + Main.clients.get(i).getMail();

    }
    return contact;
}

public String getNameAndSurnameMethod() {
    for (int i = 0; i < Main.clients.size(); i++) {
        nameAndSurname += Main.clients.get(i).getName() + " " + Main.clients.get(i).getSurname();
    }

    return nameAndSurname;
}
```

Another method used to initialize objects from different Classes into Combo-Boxes.



```java
public void initialize(){
    generalSelectComboBoxClient.getItems().addAll(Main.clients);
    generalSelectComboBoxHerb.getItems().addAll(Main.herbs);
    generalSelectComboBoxRecipe.getItems().addAll(Main.recipes);
}
```

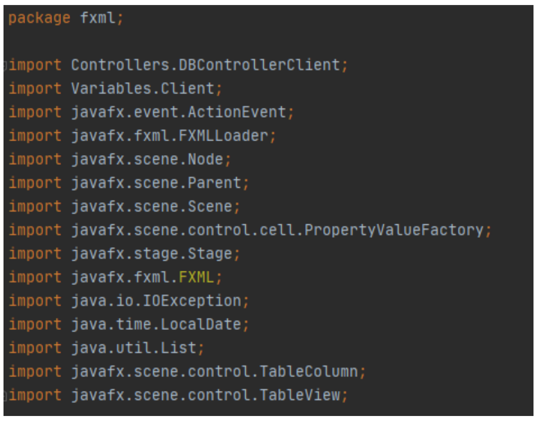
Finally with use of all methods visible above, after user clicks the “generalAssignButton” all items from Combo-Boxes are collected (with error checks) and then added to a new separate object called “General” it contains values such as the Name and Surname of a client, their diseases, contact information and assigned Herb/Recipe.



```java
@FXML
void generalAssignButton(ActionEvent event) {

    try {

        if (generalSelectComboBoxClient.getItems() != null) {
            nameAndSurname = General.parseIntoStringClient(generalSelectComboBoxClient.getSelectionModel().getSelectedItem());
            contact = General.parseIntoStringContact(generalSelectComboBoxClient.getSelectionModel().getSelectedItem());
            Illnesses = General.parseIntoStringIllnesses(generalSelectComboBoxClient.getSelectionModel().getSelectedItem());
        } else if (generalSelectComboBoxClient.getItems() == null) {
            exceptionLabel.setText("Please choose a client!");
        }

        if (generalSelectComboBoxRecipe.getSelectionModel().getSelectedItem() != null) {
            assignedRecipe = generalSelectComboBoxRecipe.getSelectionModel().getSelectedItem().getName();
        }

        if (generalSelectComboBoxHerb.getItems() != null) {
            assignedHerb = generalSelectComboBoxHerb.getSelectionModel().getSelectedItem().getName();

        }else if (generalSelectComboBoxHerb == null){
            exceptionLabel.setText("Error, please select an herb!");

        }
            General general = new General(nameAndSurname, contact, assignedHerb, assignedRecipe, Illnesses);
            DBControllerGeneral.saveDate(general);
            Main.generals.add(general);
            root = FXMLLoader.load(getClass().getResource( name: "generalDB.fxml"));
            stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
            scene = new Scene(root);
            stage.setScene(scene);
            stage.show();


        } catch(Exception e){
            e.printStackTrace();
            exceptionLabel.setText("select the client and herb");
        }
```

Only recipe is optional, thus it has a different error check

Java Libraries

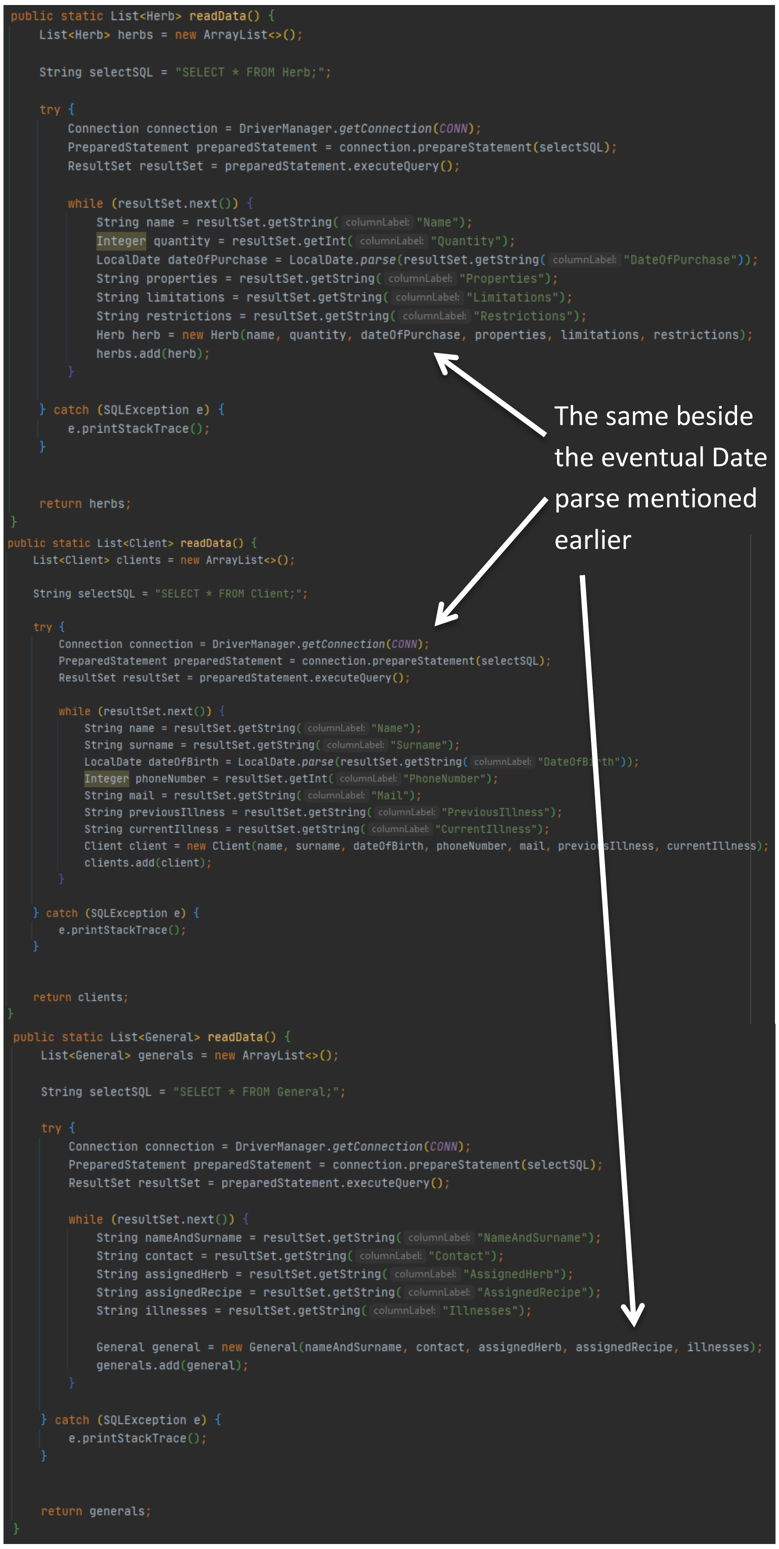


```java
package fxml;

import Controllers.DBControllerClient;
import Variables.Client;
import javafx.event.ActionEvent;
import javafx.fxml.FXMLLoader;
import javafx.scene.Node;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.stage.Stage;
import javafx.fxml.FXML;
import java.io.IOException;
import java.time.LocalDate;
import java.util.List;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
```

These libraries enabled to design the app with e.g. use of JavaFX.

Displaying tables with sets of data

In total, 4 separate tables of different objects were created (Herbs, Recipes, Clients, Generals). Herbs, Clients and Generals have the same basis visible below. They are selected form the database using the SQL Statement “SELECT FROM”.

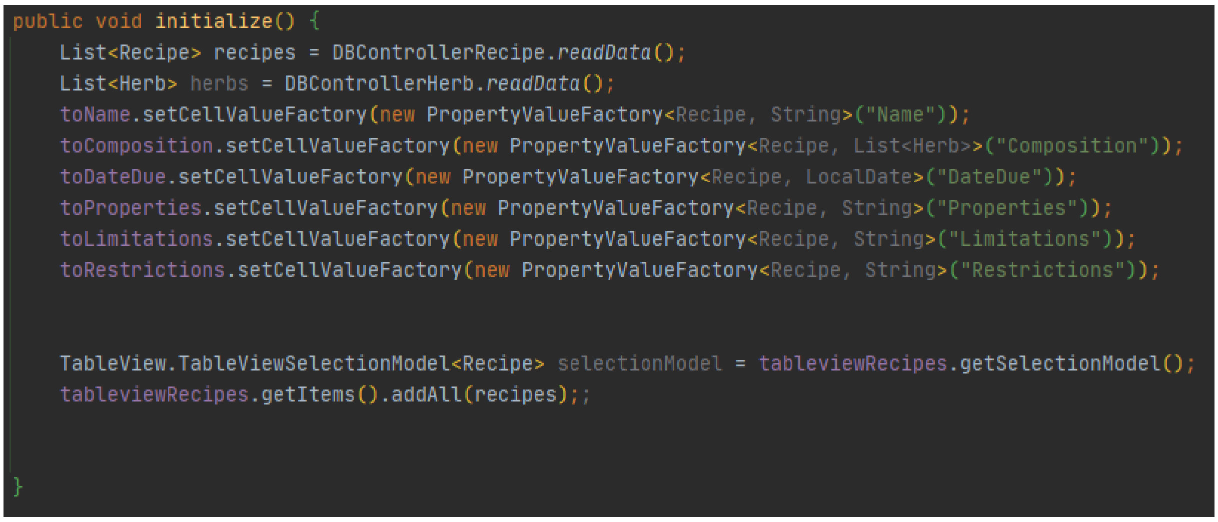


```java
public static List<Herb> readData() {
    List<Herb> herbs = new ArrayList<>();

    String selectSQL = "SELECT * FROM Herb;";

    try {
        Connection connection = DriverManager.getConnection(CONN);
        PreparedStatement preparedStatement = connection.prepareStatement(selectSQL);
        ResultSet resultSet = preparedStatement.executeQuery();

        while (resultSet.next()) {
            String name = resultSet.getString( columnLabel: "Name");
            Integer quantity = resultSet.getInt( columnLabel: "Quantity");
            LocalDate dateOfPurchase = LocalDate.parse(resultSet.getString( columnLabel: "DateOfPurchase"));
            String properties = resultSet.getString( columnLabel: "Properties");
            String limitations = resultSet.getString( columnLabel: "Limitations");
            String restrictions = resultSet.getString( columnLabel: "Restrictions");
            Herb herb = new Herb(name, quantity, dateOfPurchase, properties, limitations, restrictions);
            herbs.add(herb);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }


    return herbs;
}
public static List<Client> readData() {
    List<Client> clients = new ArrayList<>();

    String selectSQL = "SELECT * FROM Client;";

    try {
        Connection connection = DriverManager.getConnection(CONN);
        PreparedStatement preparedStatement = connection.prepareStatement(selectSQL);
        ResultSet resultSet = preparedStatement.executeQuery();

        while (resultSet.next()) {
            String name = resultSet.getString( columnLabel: "Name");
            String surname = resultSet.getString( columnLabel: "Surname");
            LocalDate dateOfBirth = LocalDate.parse(resultSet.getString( columnLabel: "DateOfBirth"));
            Integer phoneNumber = resultSet.getInt( columnLabel: "PhoneNumber");
            String mail = resultSet.getString( columnLabel: "Mail");
            String previousIllness = resultSet.getString( columnLabel: "PreviousIllness");
            String currentIllness = resultSet.getString( columnLabel: "CurrentIllness");
            Client client = new Client(name, surname, dateOfBirth, phoneNumber, mail, previousIllness, currentIllness);
            clients.add(client);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }


    return clients;
}
public static List<General> readData() {
    List<General> generals = new ArrayList<>();

    String selectSQL = "SELECT * FROM General;";

    try {
        Connection connection = DriverManager.getConnection(CONN);
        PreparedStatement preparedStatement = connection.prepareStatement(selectSQL);
        ResultSet resultSet = preparedStatement.executeQuery();

        while (resultSet.next()) {
            String nameAndSurname = resultSet.getString( columnLabel: "NameAndSurname");
            String contact = resultSet.getString( columnLabel: "Contact");
            String assignedHerb = resultSet.getString( columnLabel: "AssignedHerb");
            String assignedRecipe = resultSet.getString( columnLabel: "AssignedRecipe");
            String illnesses = resultSet.getString( columnLabel: "Illnesses");

            General general = new General(nameAndSurname, contact, assignedHerb, assignedRecipe, illnesses);
            generals.add(general);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }


    return generals;
}
```

Recipes work differently. It is based on parsing the List of "Herbs" into a String when inserted into database and vice versa when read onto the table. (visible in point "Adding a recipe")

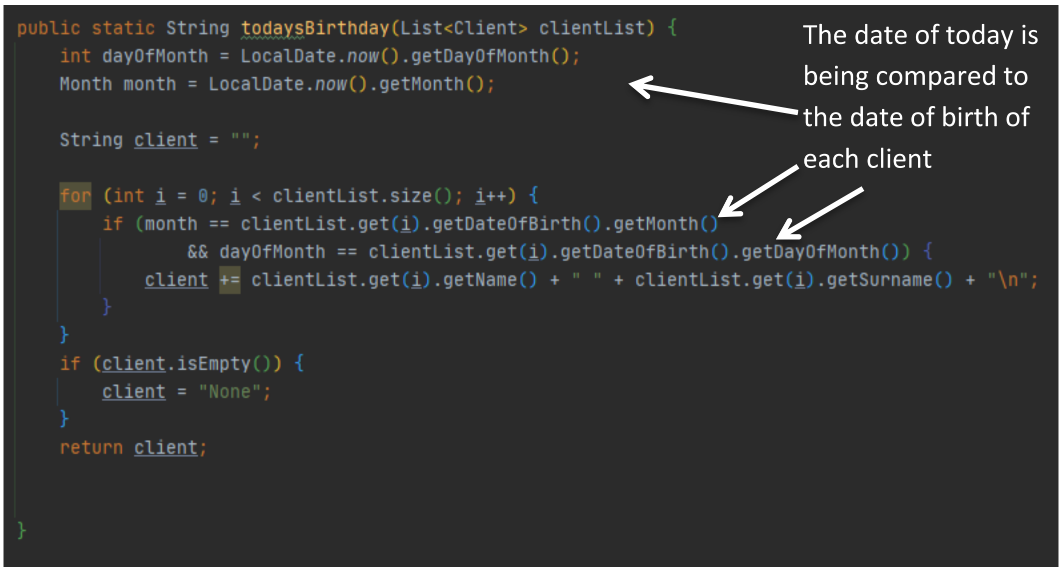
Firstly the data had to be retrieved from the database using (readData) and then the table had to be filled with values respective of their types and column names in the database from which they are read ("Name") and columns in the table which they are going to fill (toName).

For all objects the initialization looks almost the same, but with varying types of data from different objects e.g. in "Recipe", list of "Herbs", LocalDate and Strings.



```java
public void initialize() {
    List<Recipe> recipes = DBControllerRecipe.readData();
    List<Herb> herbs = DBControllerHerb.readData();
    toName.setCellValueFactory(new PropertyValueFactory<Recipe, String>("Name"));
    toComposition.setCellValueFactory(new PropertyValueFactory<Recipe, List<Herb>>("Composition"));
    toDateDue.setCellValueFactory(new PropertyValueFactory<Recipe, LocalDate>("DateDue"));
    toProperties.setCellValueFactory(new PropertyValueFactory<Recipe, String>("Properties"));
    toLimitations.setCellValueFactory(new PropertyValueFactory<Recipe, String>("Limitations"));
    toRestrictions.setCellValueFactory(new PropertyValueFactory<Recipe, String>("Restrictions"));


    TableView.TableViewSelectionModel<Recipe> selectionModel = tableviewRecipes.getSelectionModel();
    tableviewRecipes.getItems().addAll(recipes);;


}
```

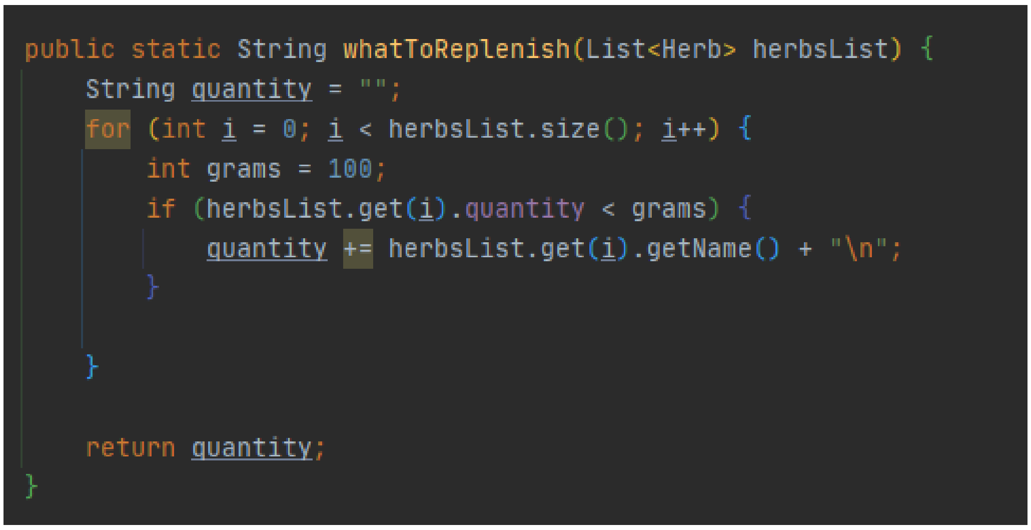
Calculations and sorting methods



```java
public static String todaysBirthday(List<Client> clientList) {
    int dayOfMonth = LocalDate.now().getDayOfMonth();
    Month month = LocalDate.now().getMonth();

    String client = "";

    for (int i = 0; i < clientList.size(); i++) {
        if (month == clientList.get(i).getDateOfBirth().getMonth()
                && dayOfMonth == clientList.get(i).getDateOfBirth().getDayOfMonth()) {
            client += clientList.get(i).getName() + " " + clientList.get(i).getSurname() + "\n";
        }
    }
    if (client.isEmpty()) {
        client = "None";
    }
    return client;


}
```

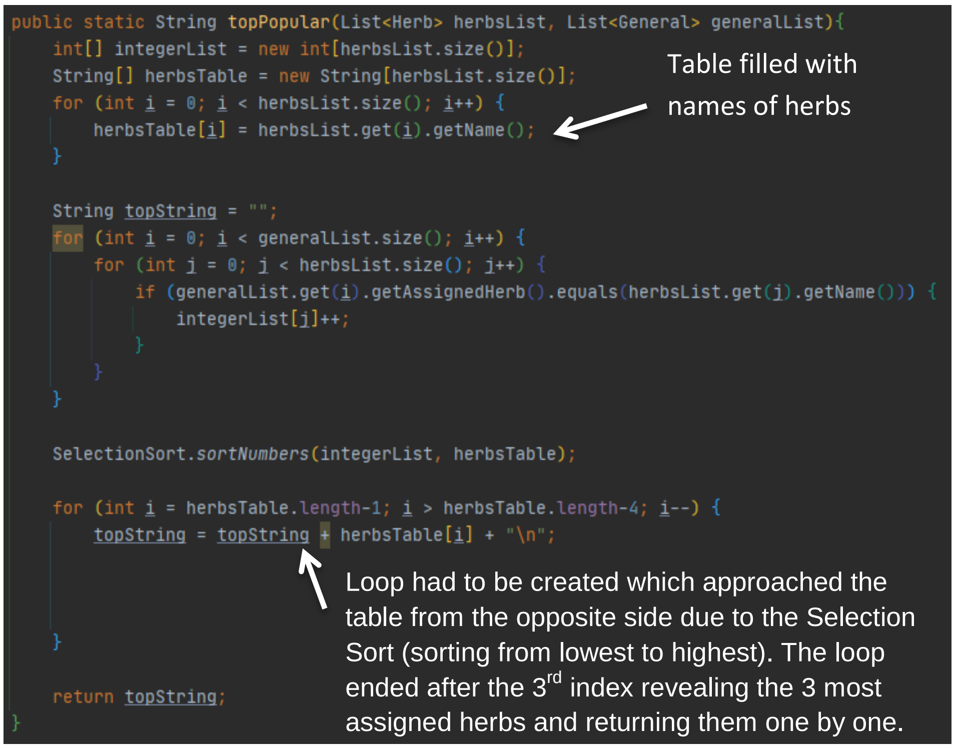
The date of today is being compared to the date of birth of each client

"todaysBirthday" searches for a client whose birthday is in the current day. At first today's date is created (LocalDate.now). Then with "for i" loop, it loops the list containing all clients. If the date of today matches with one of the client's date of birth, the algorithm returns said client's name and surname. Otherwise it returns "none".



```
public static String whatToReplenish(List<Herb> herbsList) {
    String quantity = "";
    for (int i = 0; i < herbsList.size(); i++) {
        int grams = 100;
        if (herbsList.get(i).quantity < grams) {
            quantity += herbsList.get(i).getName() + "\n";
        }

    }

    return quantity;
}
```

"whatToReplenish" searches for herbs which need replenishing. It's a pretty simple algorithm which loops the list of herbs and searches for ones with quantity smaller than "100". If it finds one it gets added to a list. If it finds more than one, the next will be added below. This method returns quantity which is the list of found herbs that need replenishing

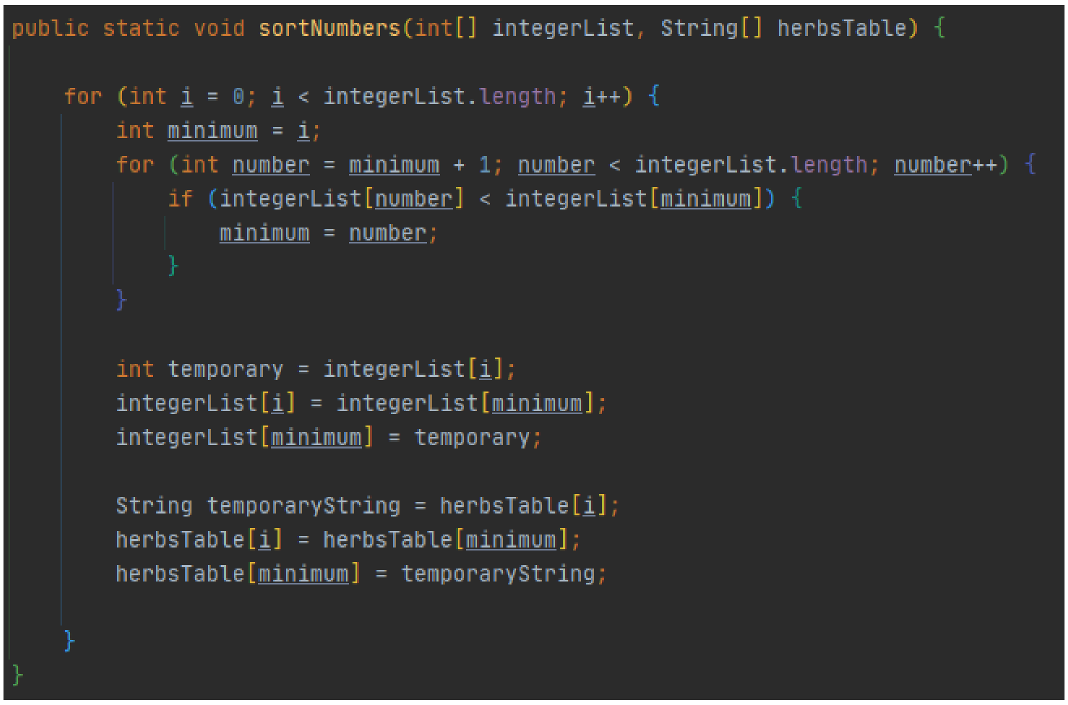


```
public static String topPopular(List<Herb> herbsList, List<General> generalList){
    int[] integerList = new int[herbsList.size()];
    String[] herbsTable = new String[herbsList.size()];
    for (int i = 0; i < herbsList.size(); i++) {
        herbsTable[i] = herbsList.get(i).getName();
    }

    String topString = "";
    for (int i = 0; i < generalList.size(); i++) {
        for (int j = 0; j < herbsList.size(); j++) {
            if (generalList.get(i).getAssignedHerb().equals(herbsList.get(j).getName())) {
                integerList[j]++;
            }
        }
    }

    SelectionSort.sortNumbers(integerList, herbsTable);

    for (int i = herbsTable.length-1; i > herbsTable.length-4; i--) {
        topString = topString + herbsTable[i] + "\n";


    }

    return topString;
}
```

Table filled with names of herbs

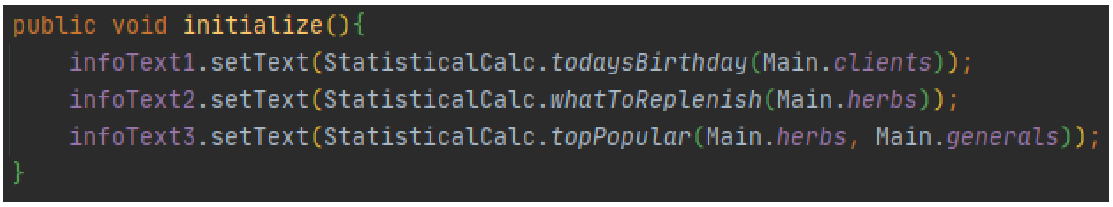
Loop had to be created which approached the table from the opposite side due to the Selection Sort (sorting from lowest to highest). The loop ended after the 3rd index revealing the 3 most assigned herbs and returning them one by one.

"topPopular" is more complex than the two mentioned. It is based on parallel arrays and nested loops. With this method the aim is to get the names and frequency of the 3 herbs that have been assigned to clients the most. To achieve this, 2 parallel tables of integers (integerList) and Strings (herbsTable) for later (sorting) had to be created. Then, nested loops were used in order to find equal terms in the listof assignment and the list of herbs containing the names. If the initial condition was met, "+1" was added to the index of the herb that was found.



```java
public static void sortNumbers(int[] integerList, String[] herbsTable) {

    for (int i = 0; i < integerList.length; i++) {
        int minimum = i;
        for (int number = minimum + 1; number < integerList.length; number++) {
            if (integerList[number] < integerList[minimum]) {
                minimum = number;
            }
        }

        int temporary = integerList[i];
        integerList[i] = integerList[minimum];
        integerList[minimum] = temporary;

        String temporaryString = herbsTable[i];
        herbsTable[i] = herbsTable[minimum];
        herbsTable[minimum] = temporaryString;

    }
}
```

Then, Selection Sort was used. It enables to sort the numbers from the lowest to highest in an int table (integerList). In this case it goes through every item in the array and brings the smallest to the front and because it is sorted, excludes it from the loop. With the smallest integer must also come respective data from the parallel array, thus the swapping technique was also used for Strings (names of herbs).

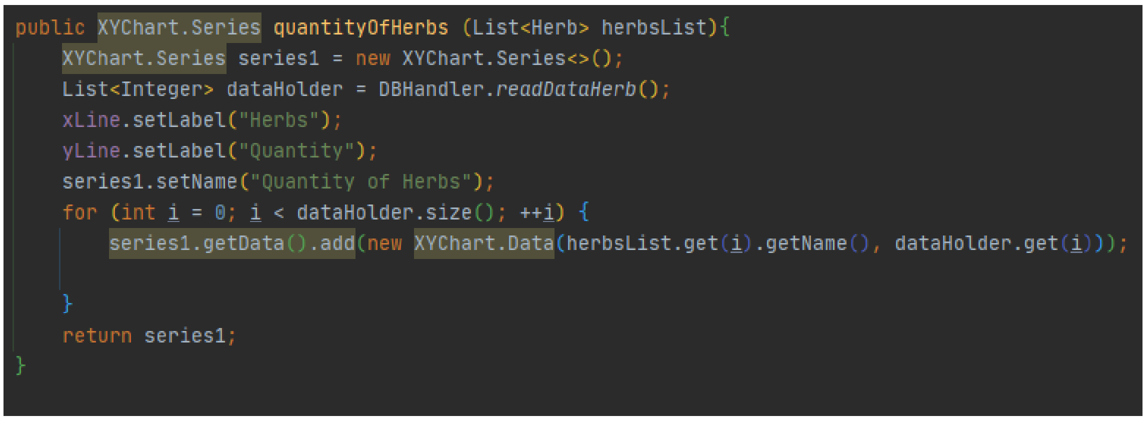


```java
public void initialize(){
    infoText1.setText(StatisticalCalc.todaysBirthday(Main.clients));
    infoText2.setText(StatisticalCalc.whatToReplenish(Main.herbs));
    infoText3.setText(StatisticalCalc.topPopular(Main.herbs, Main.generals));
}
```

Both of these are then initialized and displayed on the screen as seen above.

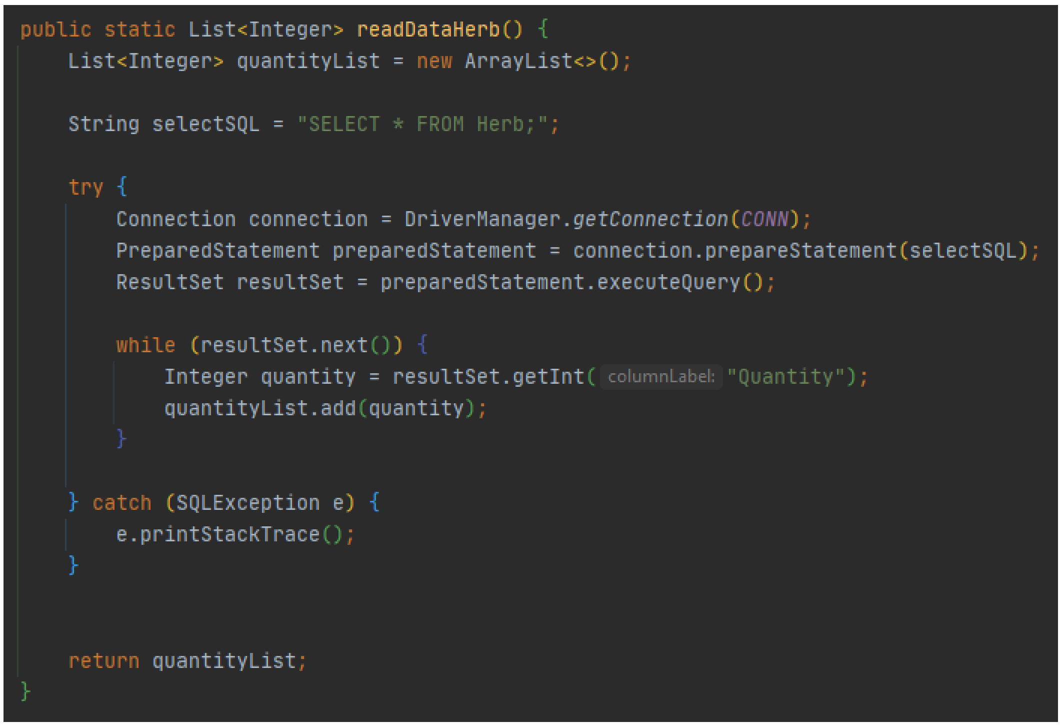
Charts and series

Two charts were created for this software.



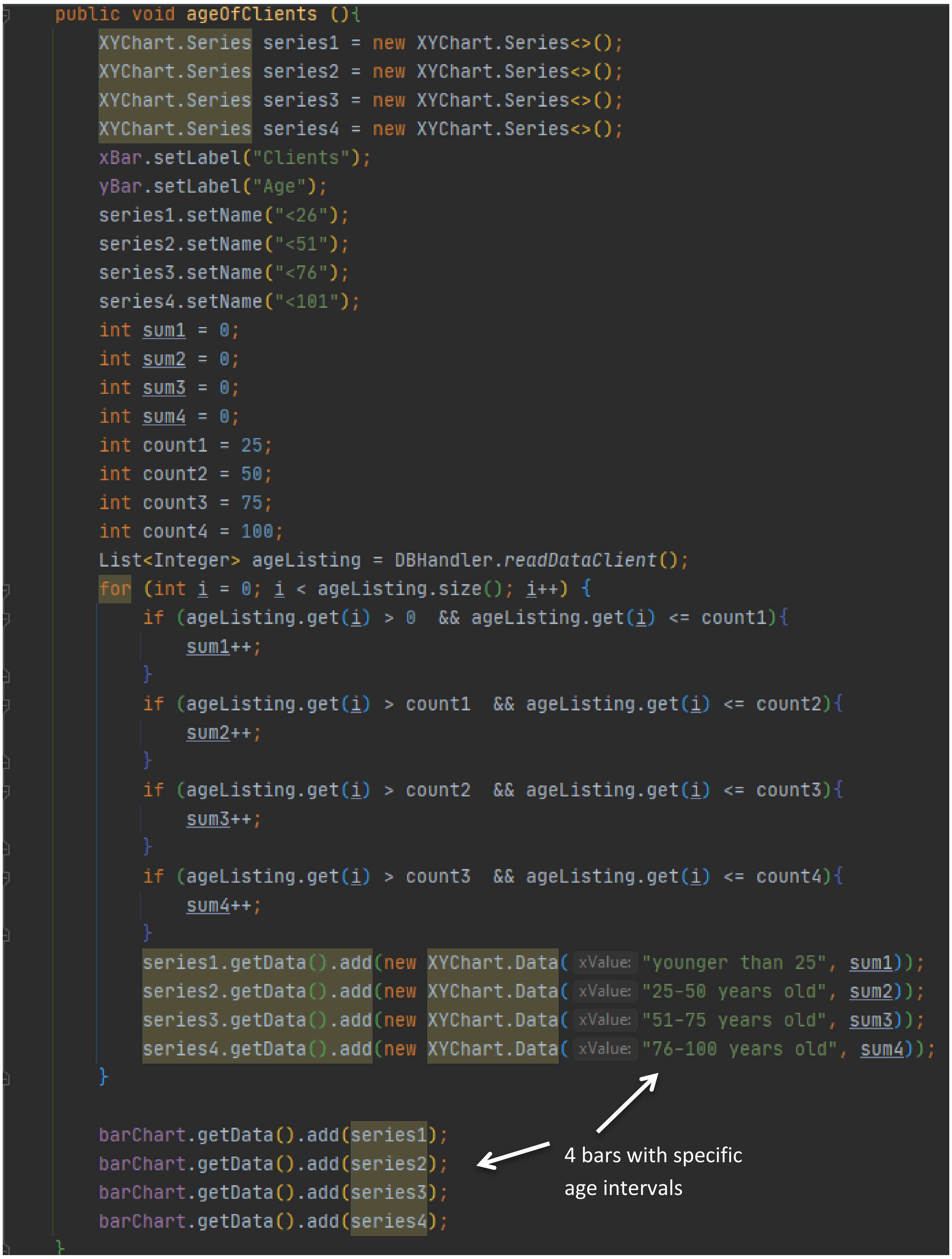
```java
public XYChart.Series quantityOfHerbs (List<Herb> herbsList){
    XYChart.Series series1 = new XYChart.Series<>();
    List<Integer> dataHolder = DBHandler.readDataHerb();
    xLine.setLabel("Herbs");
    yLine.setLabel("Quantity");
    series1.setName("Quantity of Herbs");
    for (int i = 0; i < dataHolder.size(); ++i) {
        series1.getData().add(new XYChart.Data(herbsList.get(i).getName(), dataHolder.get(i)));

    }
    return series1;
}
```

“quantityOfHerbs” uses data from “dataHolder” visible below, and displayes each herb by their quantity on a LineChart.



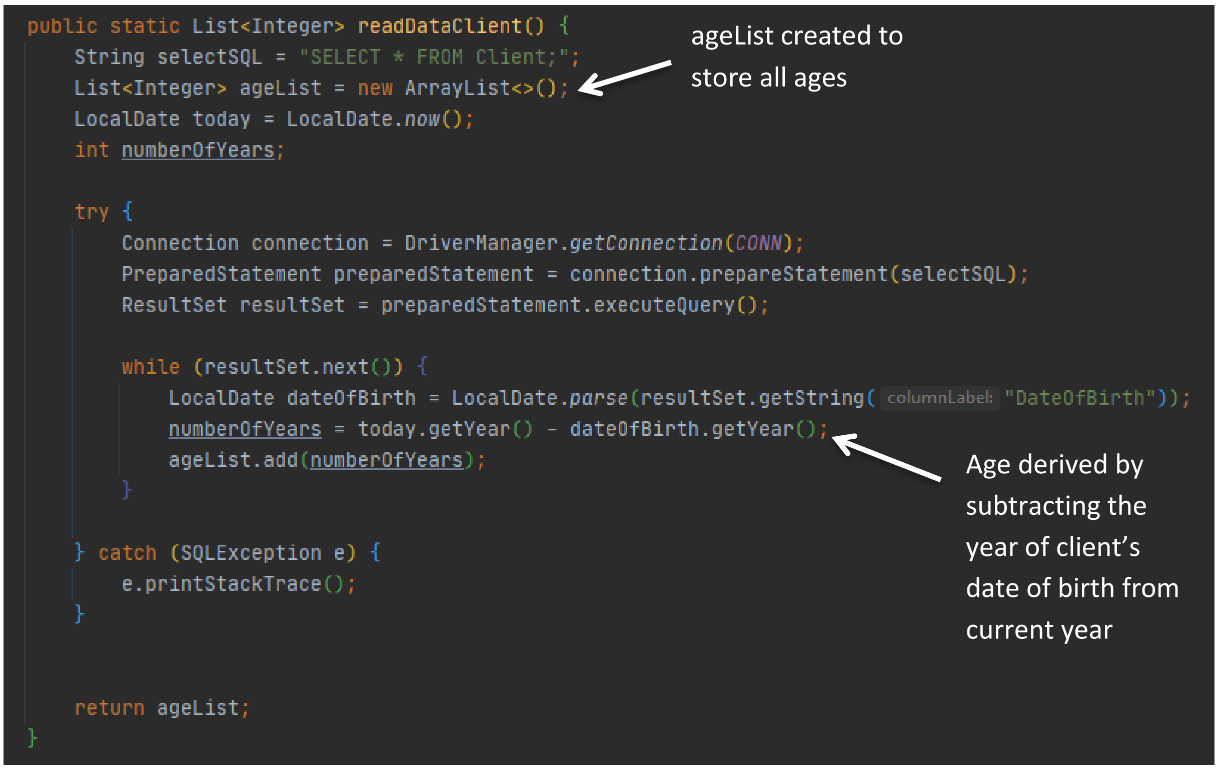
```java
public static List<Integer> readDataHerb() {
    List<Integer> quantityList = new ArrayList<>();

    String selectSQL = "SELECT * FROM Herb;";

    try {
        Connection connection = DriverManager.getConnection(CONN);
        PreparedStatement preparedStatement = connection.prepareStatement(selectSQL);
        ResultSet resultSet = preparedStatement.executeQuery();

        while (resultSet.next()) {
            Integer quantity = resultSet.getInt( columnLabel: "Quantity");
            quantityList.add(quantity);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }


    return quantityList;
}
```

It also uses SQL Statements and selects quantity value from herbs in the database.



```java
public void ageOfClients (){
    XYChart.Series series1 = new XYChart.Series<>();
    XYChart.Series series2 = new XYChart.Series<>();
    XYChart.Series series3 = new XYChart.Series<>();
    XYChart.Series series4 = new XYChart.Series<>();
    xBar.setLabel("Clients");
    yBar.setLabel("Age");
    series1.setName("<26");
    series2.setName("<51");
    series3.setName("<76");
    series4.setName("<101");
    int sum1 = 0;
    int sum2 = 0;
    int sum3 = 0;
    int sum4 = 0;
    int count1 = 25;
    int count2 = 50;
    int count3 = 75;
    int count4 = 100;
    List<Integer> ageListing = DBHandler.readDataClient();
    for (int i = 0; i < ageListing.size(); i++) {
        if (ageListing.get(i) > 0  && ageListing.get(i) <= count1){
            sum1++;
        }
        if (ageListing.get(i) > count1  && ageListing.get(i) <= count2){
            sum2++;
        }
        if (ageListing.get(i) > count2  && ageListing.get(i) <= count3){
            sum3++;
        }
        if (ageListing.get(i) > count3  && ageListing.get(i) <= count4){
            sum4++;
        }
        series1.getData().add(new XYChart.Data( xValue: "younger than 25", sum1));
        series2.getData().add(new XYChart.Data( xValue: "25-50 years old", sum2));
        series3.getData().add(new XYChart.Data( xValue: "51-75 years old", sum3));
        series4.getData().add(new XYChart.Data( xValue: "76-100 years old", sum4));
    }

    barChart.getData().add(series1);
    barChart.getData().add(series2);
    barChart.getData().add(series3);
    barChart.getData().add(series4);
}
```

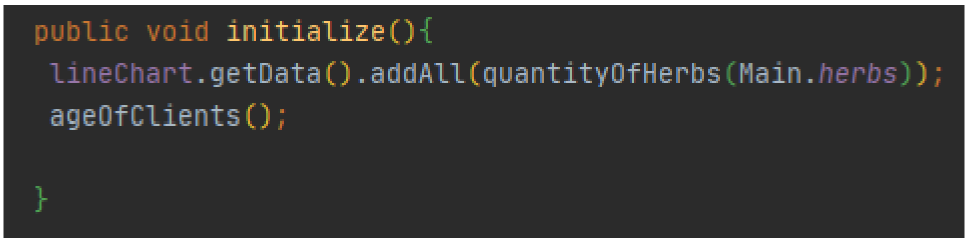
“ageOfClients” also uses data from “dataHolder”, visible below. It was used in order to display the number of people between specified age intervals on a BarChart.



```
public static List<Integer> readDataClient() {
    String selectSQL = "SELECT * FROM Client;";
    List<Integer> ageList = new ArrayList<>();
    LocalDate today = LocalDate.now();
    int numberOfYears;

    try {
        Connection connection = DriverManager.getConnection(CONN);
        PreparedStatement preparedStatement = connection.prepareStatement(selectSQL);
        ResultSet resultSet = preparedStatement.executeQuery();

        while (resultSet.next()) {
            LocalDate dateOfBirth = LocalDate.parse(resultSet.getString( columnLabel: "DateOfBirth"));
            numberOfYears = today.getYear() - dateOfBirth.getYear();
            ageList.add(numberOfYears);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }


    return ageList;
}
```

ageList created to store all ages
Age derived by subtracting the year of client's date of birth from current year

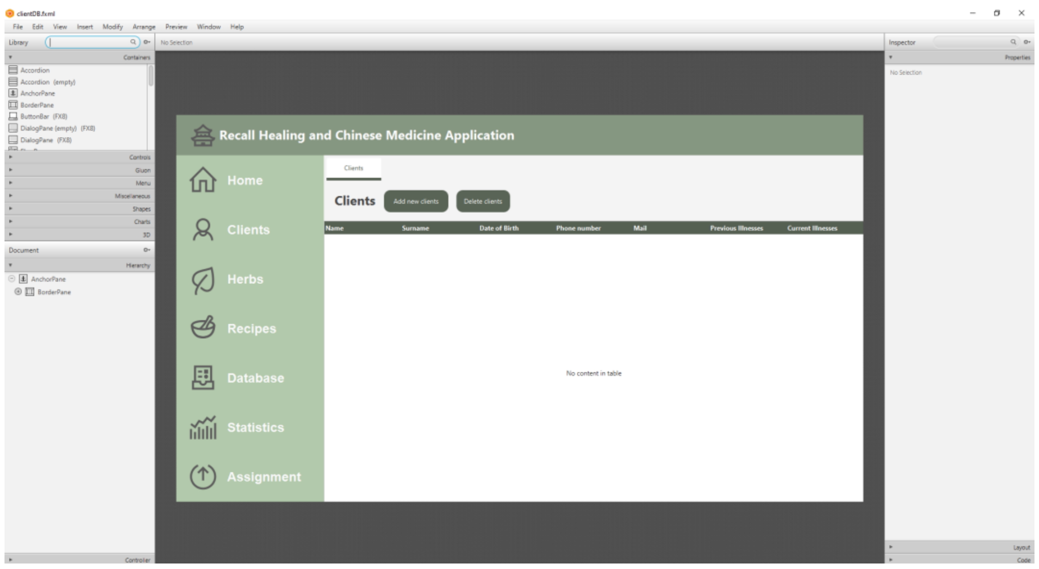
For this method a personalized List of integers was created that stores all ages. Then their age is derived from “numberOfYears” method and put into the aforementioned ageList which is then returned.



```
public void initialize(){
 lineChart.getData().addAll(quantityOfHerbs(Main.herbs));
 ageOfClients();

}
```

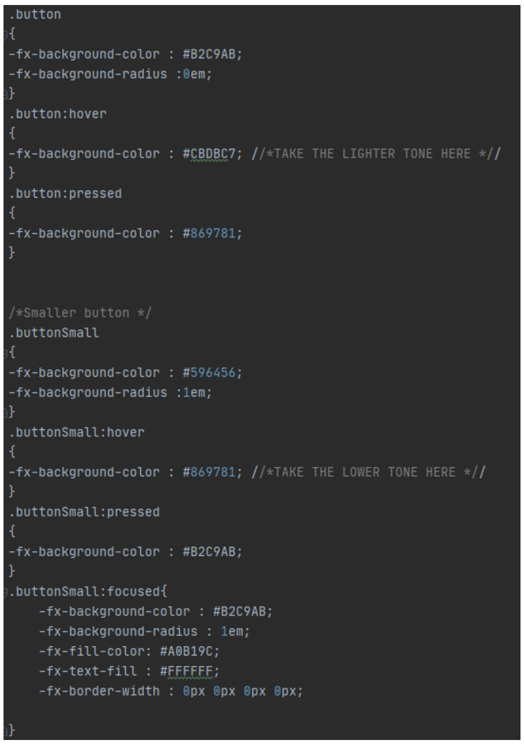
In the end both are initialized.

Scene Builder



With use of Scene Builder I was able to put together with ease every scene of the application. Buttons, Labels, Text-Fields, DataPickers, Tables, Charts, Combo-Boxes and other functions were added due to this software

CSS stylesheets



```
.button
{
-fx-background-color : #B2C9AB;
-fx-background-radius :0em;
}
.button:hover
{
-fx-background-color : #CBDBC7; //*TAKE THE LIGHTER TONE HERE *//
}
.button:pressed
{
-fx-background-color : #869781;
}


/*Smaller button */
.buttonSmall
{
-fx-background-color : #596456;
-fx-background-radius :1em;
}
.buttonSmall:hover
{
-fx-background-color : #869781; //*TAKE THE LOWER TONE HERE *//
}
.buttonSmall:pressed
{
-fx-background-color : #B2C9AB;
}
.buttonSmall:focused{
    -fx-background-color : #B2C9AB;
    -fx-background-radius : 1em;
    -fx-fill-color: #A0B19C;
    -fx-text-fill : #FFFFFF;
    -fx-border-width : 0px 0px 0px 0px;

}
```

CSS has enabled the software to not only be useful, but aesthetic. On the example a normal and small sized button are visible. They are coded in a way to slightly change colors when hovered over or clicked.

Word count 1145