

# Documentation

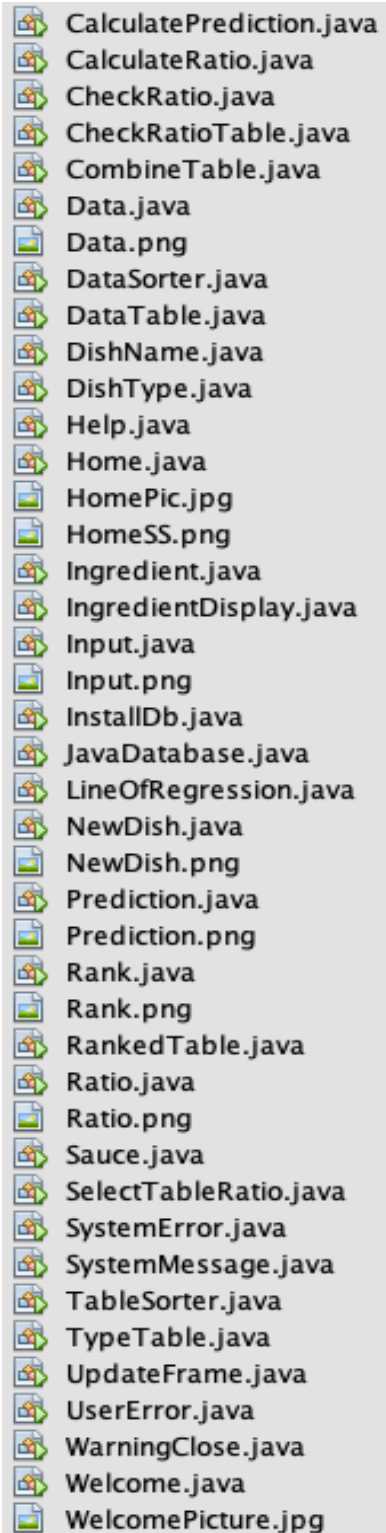
---

## **Table of Content**

1. Overall Structures
    - a. Classes List
    - b. GUI Frames and navigation
  2. Algorithms
    - a. Sorting ArrayList<ArrayList<Double>>
    - b. Converting ArrayList<ArrayList<Double>>
    - c. Summation calculate
    - d. Summation calculate of two set of data
    - e. Average value of a dataset calculate
    - f. Average value of one ingredient in all data calculate
    - g. Retrieving ingredient name data
    - h. Combining Table
    - i. Combining Ingredients
  3. Techniques use
    - a. Overloading
    - b. Encapsulation
    - c. Inheritance
    - d. Iteration
    - e. Exception Handling
    - f. Selection Sort
    - g. ArrayList
    - h. Array
    - i. If-Else condition
    - j. Accessing Database
  4. Tools
    - a. To 2d Array Method
    - b. Java Derby
    - c. Java API
-

# Overall Structure

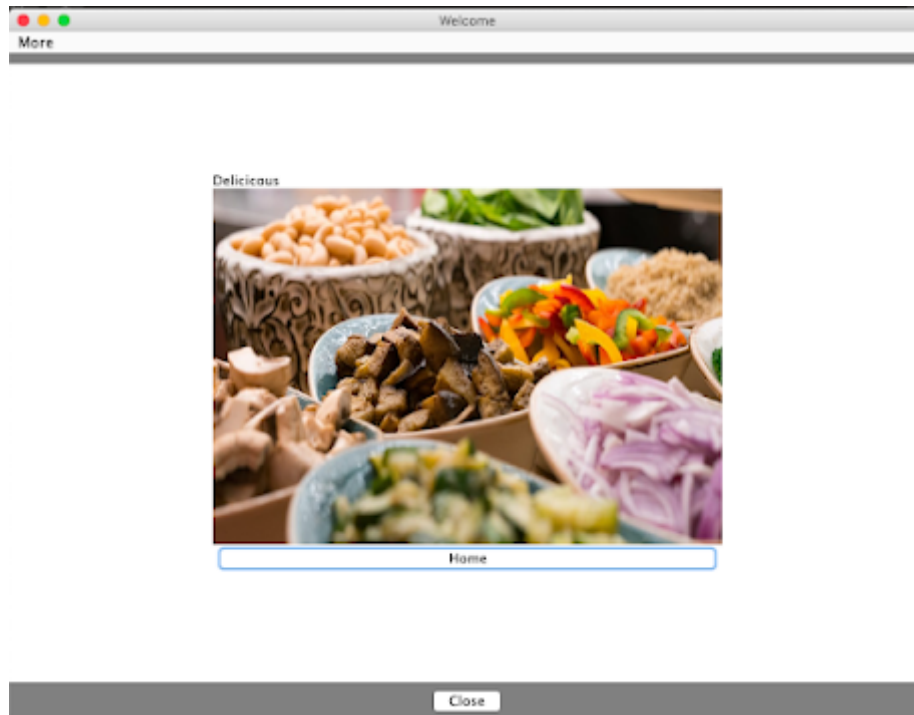
## Classes List



- CalculatePrediction.java
- CalculateRatio.java
- CheckRatio.java
- CheckRatioTable.java
- CombineTable.java
- Data.java
- Data.png
- DataSorter.java
- DataTable.java
- DishName.java
- DishType.java
- Help.java
- Home.java
- HomePic.jpg
- HomeSS.png
- Ingredient.java
- IngredientDisplay.java
- Input.java
- Input.png
- InstallDb.java
- JavaDatabase.java
- LineOfRegression.java
- NewDish.java
- NewDish.png
- Prediction.java
- Prediction.png
- Rank.java
- Rank.png
- RankedTable.java
- Ratio.java
- Ratio.png
- Sauce.java
- SelectTableRatio.java
- SystemError.java
- SystemMessage.java
- TableSorter.java
- TypeTable.java
- UpdateFrame.java
- UserError.java
- WarningClose.java
- Welcome.java
- WelcomePicture.jpg

Justification: For the program, I followed the OOP ( Object-oriented programming ) programming style. I used OOP mainly because I can break down the functionality, User interface and others features of the program into chunks.

## Welcome Frame -> Home Frame

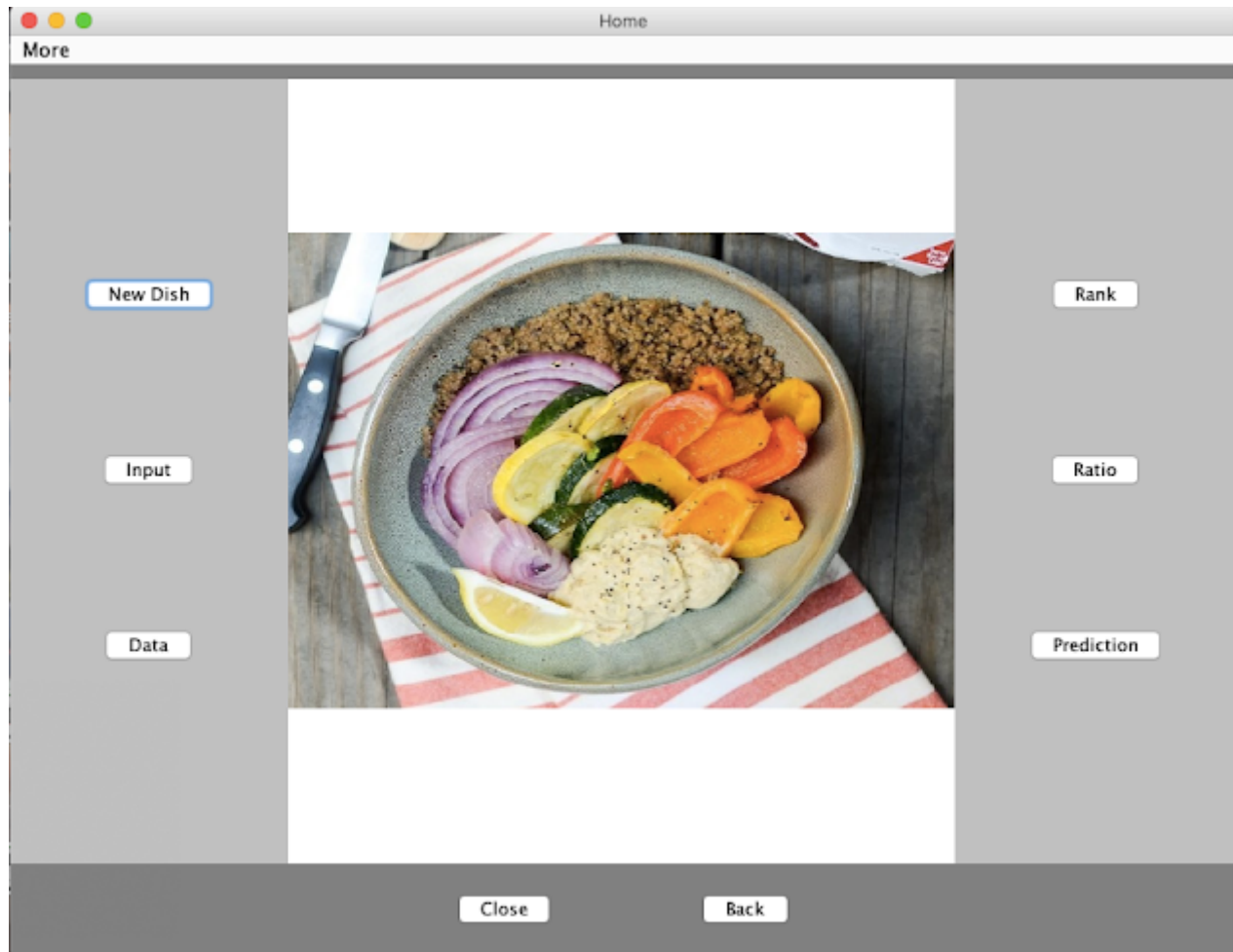


**Figure 1.** Welcome frame, present navigation tool to go to home frame or close the program

```
public void actionPerformed(ActionEvent e)
{
    String command = e.getActionCommand();
    if (command.equals("Close"))
    {
        System.exit(0);
    }
    else if (command.equals("Help"))
    {
        this.dispose();
        Help objHelp = new Help();
    }
    else if (command.equals("Home"))
    {
        this.dispose();
        Home objHome = new Home();
    }
    this.validate();
    this.repaint();
}
```

**Figure 2.** Buttons functionality of going to different frames

Home Frame -> New Dish frame, Input frame, Data Frame, Rank Frame, Ratio Frame, Prediction Frame.



**Figure 3.** Home frame, present navigation to all the main functionality of the program

- Buttons Functionality in Home frame
- New Dish Button pressed -> New Dish Frame
    - Input Button pressed -> Input Frame
    - Data Button pressed -> Data Frame
    - Rank Button pressed -> Rank Frame
    - Ratio Button pressed -> Ratio Frame
  - Prediction Button pressed -> Prediction Frame

```

public void actionPerformed(ActionEvent e)
{
    String command = e.getActionCommand();
    if (command.equals("Close"))
    {
        System.exit(0);
    } else if (command.equals("Back"))
    {
        this.dispose();
        Welcome objWelcome = new Welcome();
    } else if (command.equals("Help"))
    {
        this.dispose();
        Help objHelp = new Help();
    } else if (command.equals("Data"))
    {
        this.dispose();
        Data objData = new Data();
    } else if (command.equals("New Dish"))
    {
        this.dispose();
        NewDish objNewDish = new NewDish();
    } else if (command.equals("Input"))
    {
        this.dispose();
        Input objInput = new Input();
    } else if (command.equals("Rank"))
    {
        this.dispose();
        Rank objRank = new Rank();
    } else if (command.equals("Ratio"))
    {
        this.dispose();
        CheckRatio objRatio = new CheckRatio();
    } else if (command.equals("Prediction"))
    {
        this.dispose();
        Prediction objPrediction = new Prediction();
    }

    this.validate();
    this.repaint();
}

```

**Figure 4.** Home Frame's buttons functionality of going into different frames

## Inputting New Dish Frame

The screenshot shows a window titled "New Dish Input111". It features a "More" button at the top left. The main content area is divided into two columns. The left column contains a "Dish Name" label and a corresponding text input field. The right column is further divided into two sections: "Ingredients" and "Sauces and Spices". The "Ingredients" section includes a dropdown menu currently showing the number "5", followed by five empty text input fields. The "Sauces and Spices" section contains one empty text input field. To the right of this input field is a button labeled "Input Entree". At the bottom of the window, there are two buttons: "Close" and "Home".

**Figure 5.** *New Dish Frame, provides a textfield and buttons for users to input into a database new dishes along with components of the dish.*

```
if (command.equals("Close"))
{
    System.exit(0);
} else if (command.equals("Home"))
{
    this.dispose();
    Home objHome = new Home();
} else if (command.equals("Help"))
{
    this.dispose();
    Help objHelp = new Help();
}
```

**Figure 6.** *New Dish Frame 's buttons functionality of going into different frames*

## Input Frame

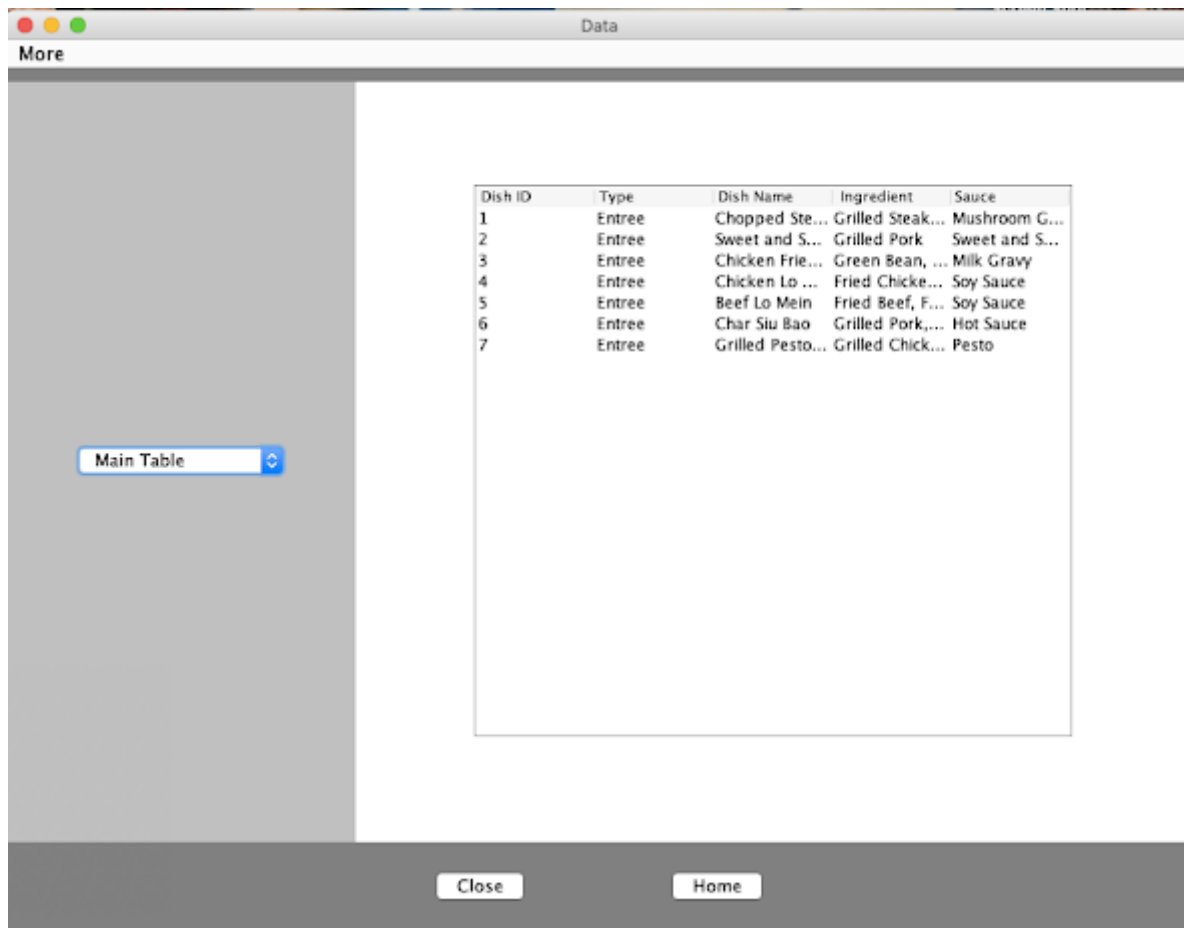
Dish ID	Dish Name	Type	Ingredient	Sauce
1	Entree	Chopped Ste...	Grilled Steak...	Mushroom G...
2	Entree	Sweet and S...	Grilled Pork	Sweet and S...
3	Entree	Chicken Frie...	Green Bean, ...	Milk Gravy
4	Entree	Chicken Lo ...	Fried Chicke...	Soy Sauce
5	Entree	Beef Lo Mein	Fried Beef, F...	Soy Sauce
6	Entree	Char Siu Bao	Grilled Pork,...	Hot Sauce
7	Entree	Grilled Pesto...	Grilled Chick...	Pesto

**Figure 7.** Input Frame allows the user to input the daily amount of food made and food left over, it also has a table for reference.

```
@Override
public void actionPerformed(ActionEvent e)
{
    String command = e.getActionCommand();
    double foodMade;
    double foodLeft;
    double ratio;
    int dishID;
    //Database Connection
    //Database Connection
    String dbName = "CUISINES";
    JavaDatabase objDb = new JavaDatabase(dbName);
    Connection myDbConn = null;
    myDbConn = objDb.getDbConn();
    Object[] [] dataTable;
    if (command.equals("Close"))
    {
        System.exit(0);
    } else if (command.equals("Home"))
    {
        this.dispose();
        Home objHome = new Home();
    } else if (command.equals("Help"))
    {
        this.dispose();
        Help objHelp = new Help();
    }
}
```

**Figure 8.** Input Frame 's buttons functionality of going into different frames

## Data Frame



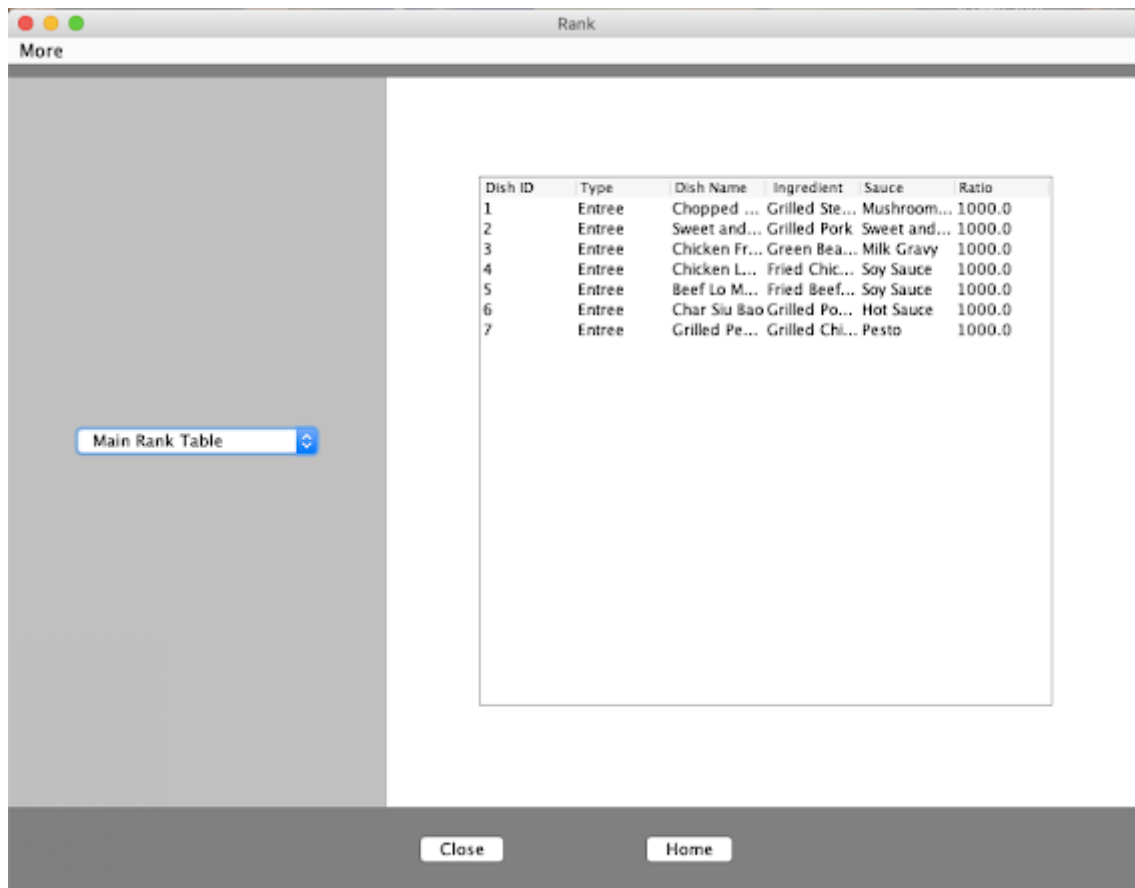
**Figure 9.** Data Frame allows the user to view data in the database, the data is also classified into different types of dishes.

```
@Override
public void actionPerformed(ActionEvent e)
{
    String command = e.getActionCommand();
    if (command.equals("Close"))
    {
        System.exit(0);
    } else if (command.equals("Home"))
    {
        this.dispose();
        Home objHome = new Home();
    } else if (command.equals("Help"))
    {
        this.dispose();
        Help objHelp = new Help();
    }
}
```

**Figure 10.** Data Frame 's buttons functionality of going into different frames



## Rank Frame

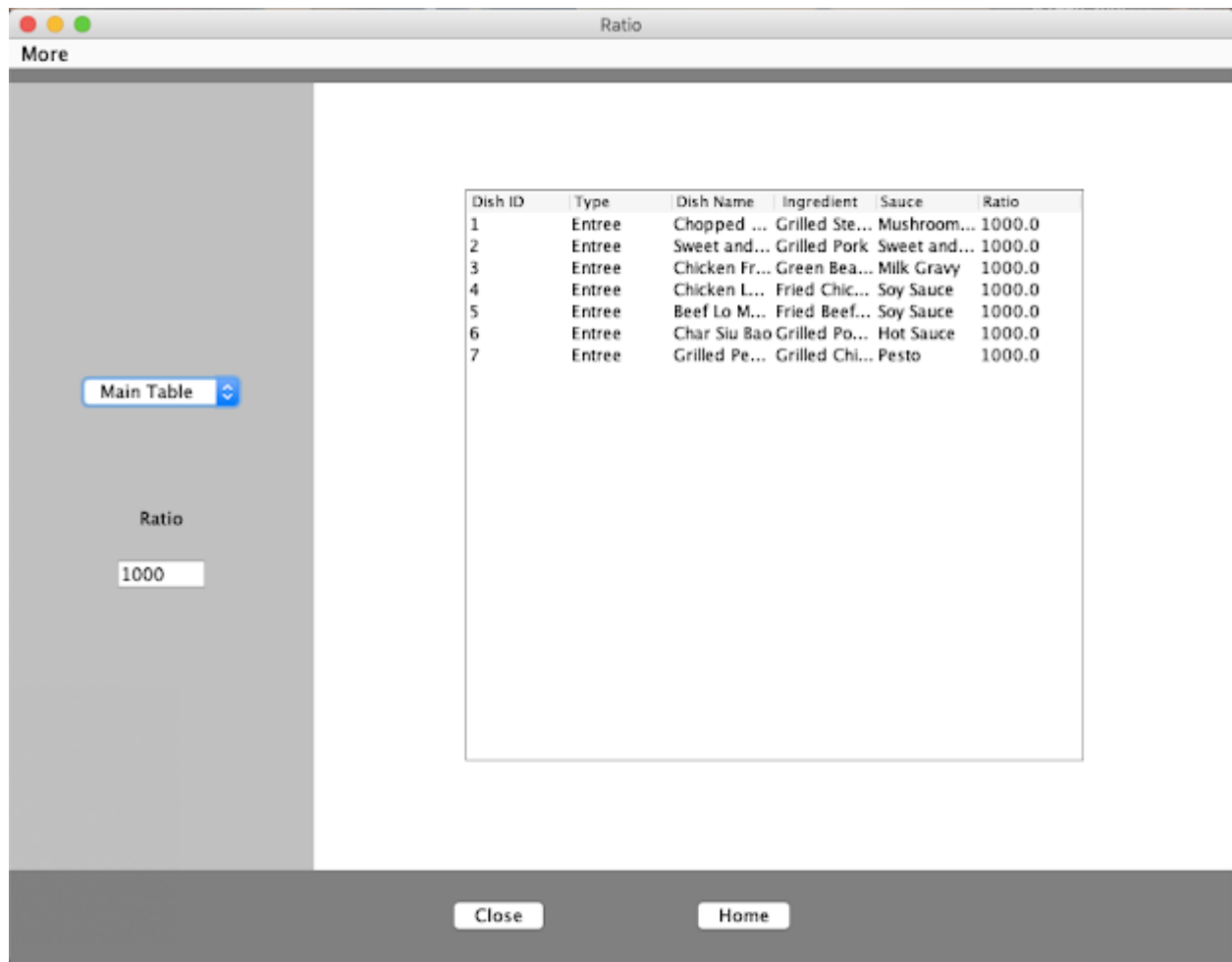


**Figure 11.** Data Frame allows the user to view data that is sorted in the database, the data is also classified into different types of dishes.

```
@Override
public void actionPerformed(ActionEvent e)
{
    String targetTable;
    String dbName = "CUISINES";
    String command = e.getActionCommand();
    Object[][] dataTable;
    if (command.equals("Close"))
    {
        System.exit(0);
    }
    if (command.equals("Home"))
    {
        this.dispose();
        Home objHome = new Home();
    }
    if (command.equals("Help"))
    {
        this.dispose();
        Help objHelp = new Help();
    }
}
```

**Figure 12.** Rank Frame 's buttons functionality of going into different frames

## Check Ratio Frame



**Figure 13.** Check Ratio Frame allows the user to view data that satisfied the ratio that the user want

```
@Override
public void actionPerformed(ActionEvent e)
{
    String targetTable;
    String dbName = "CUISINES";
    String command = e.getActionCommand();
    Object[][] dataTable;
    if (command.equals("Close"))
    {
        System.exit(0);
    }
    if (command.equals("Home"))
    {
        this.dispose();
        Home objHome = new Home();
    }
    if (command.equals("Help"))
    {
        this.dispose();
        Help objHelp = new Help();
    }
}
```

**Figure 14.** Check Ratio Frame 's buttons functionality of going into different frames

## Prediction Frame

The screenshot shows a window titled "Prediction" with a standard macOS-style title bar (red, yellow, green buttons). Below the title bar is a menu bar with the word "More". The main content area is white and contains the following elements:

- A label "Number of Ingredients" above a spin box containing the number "3".
- Below the spin box are three empty text input fields stacked vertically.
- A label "Ingredients" above a "Display" button.
- A "Predict" button to the right of the "Display" button.

At the bottom of the window is a dark gray footer bar containing two buttons: "Close" and "Home".

**Figure 15.** Prediction Frame allows user to see the predicted ratio of a dish base on the input ingredient

```
if (command.equals("Close"))
{
    System.exit(0);
} else if (command.equals("Home"))
{
    this.dispose();
    Home objHome = new Home();
} else if (command.equals("Help"))
{
    this.dispose();
    Help objHelp = new Help();
}
```

**Figure 16.** Prediction Frame 's buttons functionality of going into different frames

# Algorithms

## Sorting ArrayList<ArrayList<Double>> algorithm

```
public void setSortedData()
{
    Double[] arID = new Double[this.unsortedData.size()];
    Double[] arRatio = new Double[this.unsortedData.size()];
    for (int i = 0; i < this.unsortedData.size(); i++)
    {
        ArrayList<Double> tempData = this.unsortedData.get(i);

        arID[i] = tempData.get(0);
        arRatio[i] = tempData.get(1);
    }
    for (int i = 0; i < arRatio.length; i++)
    {
        int min = i;
        for (int j = i + 1; j < arRatio.length; j++)
        {
            if (arRatio[j] < arRatio[min])
            {
                min = j;
            }
        }
        //Swapping
        double temp = arRatio[min];
        arRatio[min] = arRatio[i];
        arRatio[i] = temp;

        double temp1 = arID[min];
        arID[min] = arID[i];
        arID[i] = temp1;
    }

    for (int i = 0; i < this.unsortedData.size(); i++)
    {
        ArrayList<Double> tempData = new ArrayList<>();
        tempData.add(arID[i]);
        tempData.add(arRatio[i]);
        this.sortedData.add(tempData);
    }
}
```

**Figure 17.** Algorithm sort data in ArrayList<ArrayList<String>>

The data is first transferred from ArrayList<ArrayList<String>> into two array Double[] . This is done to use a selection sort algorithm. In order to keep the dish ID matching with the correct ratio. Both Array will be switching places in synchronization.

## Converting ArrayList<ArrayList<Double>> Algorithm

```
public ArrayList<ArrayList<String>> intToStringArray(ArrayList<ArrayList<Double>> data)
{
    for (int i = 0; i < data.size(); i++)
    {
        ArrayList<String> tempString = new ArrayList<>();
        ArrayList<Double> tempInt = data.get(i);
        Double doubleDishID = tempInt.get(0);
        int intDishID = (int) Math.round(doubleDishID);
        String dishID = Integer.toString(intDishID);
        String ratio = Double.toString(tempInt.get(1));
        tempString.add(dishID);
        tempString.add(ratio);
        this.dataTable.add(tempString);
    }
    return this.dataTable;
}
```

**Figure 18.** Using For loop, rounding, and toString to convert ArrayList<ArrayList<Double>> to ArrayList<ArrayList<String>>

## Summation calculate Algorithm

```
//Calculate summation of x
public void setSumationX(Double[] xSet)
{
    try
    {
        for (int i = 0; i < xSet.length; i++)
        {
            this.sumationX = xSet[i] + this.sumationX;
        }
    } catch (Exception exceptionObj)
    {
        SystemError objError = new SystemError("Something went wrong in the system, please contact developer")
    }
}
```

**Figure 19.** Using For loop to calculate the summation of all data in a data set

## Summation calculate of two set of data Algorithm

```
//Calculate sumation of x*y
public void setSumationXY(Double[] xSet, Double[] ySet)
{
    try
    {
        Double[] xySet = new Double[xSet.length];
        //Calculate xy set
        for (int i = 0; i < ySet.length; i++)
        {
            double var1 = xSet[i];
            double var2 = ySet[i];
            xySet[i] = var1 * var2;
        }
        //Calculate sumation of xy set
        for (int i = 0; i < xySet.length; i++)
        {
            this.sumationXY = this.sumationXY + xySet[i];
        }
    } catch (Exception exceptionObj)
    {
        SystemError objError = new SystemError("Something went wrong in the system, please contact developer")
    }
}
```

*Figure 20. Using For loop to calculate the summation of all data in two data sets*

## Average value of a dataset calculate algorithm

```
public double CalculateAvg(Double[] dataSet)
{
    double average = 0;
    double sum = 0;
    //Calculate sum
    for (int i = 0; i < dataSet.length; i++)
    {
        sum = dataSet[i] + sum;
    }
    //Calculate average
    average = (sum) / (int) (dataSet.length);
    //Return data
    return average;
}
```

*Figure 21. Using For loop to calculate the average of all data a two data*

## Average value of one ingredient in all data calculate algorithm

```

// calculate ingredient average
public void setIngredientAvg()
{
    try
    {
        ArrayList<Double> ratioHolder = new ArrayList<>();
        for (int i = 0; i < dataIngredient.size(); i++)
        {
            ArrayList<String> arrayIngredient = dataIngredient.get(i);
            String dishID1 = arrayIngredient.get(0);
            String ingredient = arrayIngredient.get(1);
            if (ingredient.equals(this.ingredientName))
            {
                for (int y = 0; y < dataRatioInt.size(); y++)
                {
                    ArrayList<Double> arrayRatio = dataRatioInt.get(y);
                    String dishID2 = Integer.toString((int) Math.round(arrayRatio.get(0)));
                    double ratio = arrayRatio.get(1);
                    if (dishID1.equals(dishID2))
                    {
                        if (1000.0 != arrayRatio.get(1))
                        {
                            ratioHolder.add(ratio);
                            break;
                        }
                    }
                }
            }
        }
        Double[] ratioSet = new Double[ratioHolder.size()];
        for (int i = 0; i < ratioHolder.size(); i++)
        {
            double ratio = ratioHolder.get(i);

            ratioSet[i] = ratio;
        }
        if (Double.isNaN(CalculateAvg(ratioSet)) == false)
        {
            this.ingredientAvgSum = CalculateAvg(ratioSet);
        }
    }
}

```

*Figure 22. Using For loop to calculate the average of one data in a data*

## Retrieving ingredient name data algorithm



```

public ArrayList<ArrayList<String>> getAllIngredientName()
{
    boolean repeat;
    for (int i = 0; i < this.data.size(); i++)
    {
        ArrayList<String> temp = new ArrayList<>();
        ArrayList<String> var1 = data.get(i);
        String ingredientName = var1.get(1);
        repeat = false;
        for (int z = 0; z < this.allIngredientName.size(); z++)
        {
            ArrayList<String> var2 = this.allIngredientName.get(z);
            if (ingredientName.equals(var2.get(0)))
            {
                repeat = true;
                break;
            }
        }
        if (repeat == true)
        {
            //Do nothing because the data already existed
        }
        if (ingredientName.equals(""))
        {
            //Do nothing because there are no ingredient for this dish
        }
        else
        {
            temp.add(ingredientName);
            this.allIngredientName.add(temp);
        }
    }
    return this.allIngredientName;
}

```

**Figure 23.** For loops and if statement to retrieve data and make sure that data are not repeated

## Combining Table

```

public class CombineTable
{
    private ArrayList<ArrayList<String>> dataCombine;
    private ArrayList<ArrayList<String>> dataTable1;
    private ArrayList<ArrayList<String>> dataTable2;
    private ArrayList<ArrayList<String>> dataTable3;
    private ArrayList<ArrayList<String>> dataTable4;
    private ArrayList<ArrayList<String>> dataTable5;

    private int tableCount;

    public CombineTable()
    {
        this.dataCombine = new ArrayList<>();
        this.dataTable1 = new ArrayList<>();
        this.dataTable2 = new ArrayList<>();
        this.dataTable3 = new ArrayList<>();
        this.dataTable4 = new ArrayList<>();
        this.dataTable5 = new ArrayList<>();
    }
}

```

**Figure 24.** attributes of the class

This class has 6 Array List. 5 Array List is responsible for holding the data got from the table in the database, and the last array List will hold the combination of the data in the tables. There is an integer for table count; it will identify the numbers of tables will be combining.

```

public void setDataCombine()
{
    // initialize data holder and ArrayList
    String dishID1;
    String dishID2;
    String dishID3;
    String dishID4;
    String dishID5;

    //Array List of Table 1
    ArrayList<ArrayList<String>> table1Temp = this.dataTable1;
    //Array List of Table 2
    ArrayList<ArrayList<String>> table2Temp = this.dataTable2;
    //Array List of Table 3
    ArrayList<ArrayList<String>> table3Temp = this.dataTable3;
    //Array List of Table 4
    ArrayList<ArrayList<String>> table4Temp = this.dataTable4;
    //Array List for Table 5
    ArrayList<ArrayList<String>> table5Temp = this.dataTable5;

    // Combining Table
    if (this.tableCount == 2)
    {
        for (int i = 0; i < table1Temp.size(); i++)
        {
            ArrayList<String> temp = new ArrayList();
            ArrayList<String> var1 = table1Temp.get(i);
            dishID1 = var1.get(0);
            for (int y = 0; y < table2Temp.size(); y++)
            {
                ArrayList<String> var2 = table2Temp.get(y);
                dishID2 = var2.get(0);
                if (dishID1.equals(dishID2))
                {
                    temp.add(dishID1);
                    temp.add(var1.get(1));
                    temp.add(var2.get(1));
                    this.dataCombine.add(temp);
                }
            }
        }
    }
    else if (this.tableCount == 3)

```

**Figure 25.** Combining data from tables ( 2 tables )

First the data will be transferred for a temporary ArrayList. Then the size of the first data will be for the counting group. Because of this, when using this method, the data that have the shortest size need to be in dataTable1 ArrayList because the data if the size is larger than the array inside the loop, the system will give out an error. Inside the loop, the ArrayList<String> of the ArrayList will be held in a temporary ArrayList. Then getting column 1 of the ArrayList and column 2. Column 1 represents the dishId and column 2 represents the data in that column.

After that, the algorithm will go through a second data loop and take out column 1 and column 2 of the second data. Then, the two dishID will be compared for column 1 of data 1 and column 2 of data 2. If they are equal then they will be added to a new ArrayList with column 1 of data1, column 2 of data 2, and column 2 of data 2. The ArrayList will then be added to the returning ArrayList. The process is represented until all the data that has similar dishID is combined.

```

} else if (this.tableCount == 5)
{
    for (int i = 0; i < table1Temp.size(); i++)
    {
        ArrayList<String> temp = new ArrayList();
        ArrayList<String> var1 = table1Temp.get(i);
        dishID1 = var1.get(0);
        for (int y = 0; y < table2Temp.size(); y++)
        {
            ArrayList<String> var2 = table2Temp.get(y);
            dishID2 = var2.get(0);
            ArrayList<String> var3 = table3Temp.get(y);
            dishID3 = var3.get(0);
            ArrayList<String> var4 = table4Temp.get(y);
            dishID4 = var4.get(0);
            ArrayList<String> var5 = table5Temp.get(y);
            dishID5 = var5.get(0);
            if (dishID1.equals(dishID2) && dishID1.equals(dishID3) && dishID1.equals(dishID4)
                && dishID1.equals(dishID5))
            {
                temp.add(dishID1);
                temp.add(var1.get(1));
                temp.add(var2.get(1));
                temp.add(var3.get(1));
                temp.add(var4.get(1));
                temp.add(var5.get(1));
                this.dataCombine.add(temp);
            }
        }
    }
}

```

**Figure 26.** Combining data from tables (5 tables)

## Combing Ingredients

```
public ArrayList<ArrayList<String>> CombineIngredient(ArrayList<ArrayList<String>> data)
{
    String check = "";
    boolean repeat = false;
    boolean singleData = false;
    int lastIndex = this.data.size() - 1;

    for (int i = 0; i < this.data.size(); i++)
    {
        singleData = false;
        ArrayList<String> temp = new ArrayList<>();
        ArrayList<String> var1 = data.get(i);
        String dishID1 = var1.get(0);
        String ingredient = var1.get(1);
        String combine = var1.get(1);

        for (int y = +i; y < this.data.size(); y++)
        {
            ArrayList<String> var2 = data.get(y);
            repeat = false;
            for (int z = 0; z < this.combineData.size(); z++)
            {
                ArrayList<String> var4 = this.combineData.get(z);
                if (var2.get(0).equals(var4.get(0)))
                {
                    repeat = true;
                    break;
                }
            }
            if (repeat == true)
            {
                break;
            }
            else if (var1.get(0).equals(var2.get(0)))
            {
                combine = combine + ", " + var2.get(1);
                check = var2.get(1);
            }
            else
            {
                singleData = true;
            }
        }
    }
}
```

```

    if (!ingredient.equals(check) && repeat == false)
    {
        temp.add(dishID1);
        temp.add(ingredient);
        this.combineData.add(temp);
    } else if (singleData == true)
    {
        ArrayList<String> var5 = data.get(i);
        String dishID = var5.get(0);
        String single = var5.get(1);
        temp.add(dishID);
        temp.add(single);
        this.combineData.add(temp);
    }
    else if (i == lastIndex && repeat == false)
    {
        temp.add(dishID1);
        temp.add(ingredient);
        this.combineData.add(temp);
    }
}
}

```

**Figure 27.** *Combining ingredients that have the same ID number*

This algorithm will group ingredients in the same dish under one String. First the limit of the loop will be the size of the data of the ingredient from the database. Then a temporary arraylist will be created to hold the new data that has combined ingredients. The loop will take each ArrayList<String> from the data, the first row of the ArrayList is the dish ID and the second row is the ingredient. The algorithm will take the dishID and measure it with the rest of the rest of the dishID with a loop that will start with the following index number. If the dishID matches the dishID of the following dishID then the ingredient of that index will be added to a String that holds all the names of ingredients with the same dishID. This process continues until there is no more match. Since the program is measuring each ingredient row by row, there will be repeated data or dishID. Therefore, firstly, the algorithm will check the ArrayList that holds the combined data to see if the dishID is already in there. If the dish is already in there then the ArrayList already has all the ingredients that have that ID, that loop will end. However, there will also be a problem where a dishID or a dish that only has 1 ingredient, so the program will detect that there is no dishID match with the rest of the dishID in the ArrayList and the algorithm will mark that index as single data. After the String that combines the ingredient is completed, the data will be added to the return ArrayList with combined String along with correlated dishID.

## Technique use

## Overloading

```
public void insertData(String tableName, int IDNumber, String columnName, String insert)
{
    try
    {
        String dbQuery1 = "INSERT INTO " + tableName + " VALUES (?,?)";
        PreparedStatement ps = dbConn.prepareStatement(dbQuery1);
        ps.setInt(1, IDNumber);
        ps.setString(2, insert);
        ps.executeUpdate();
    } catch (SQLException ex)
    {
        SystemError objError = new SystemError("Something went wrong in the system, please contact developer");
    }
}

public void insertData(String tableName, int IDNumber, String columnName, double insert)
{
    try
    {
        String dbQuery1 = "INSERT INTO " + tableName + " VALUES (?,?)";
        PreparedStatement ps = dbConn.prepareStatement(dbQuery1);
        ps.setInt(1, IDNumber);
        ps.setDouble(2, insert);
        ps.executeUpdate();
    } catch (SQLException ex)
    {
        SystemError objError = new SystemError("Something went wrong in the system, please contact developer");
    }
}

public void update(String tableName, int IDNumber, String columnName, String change)
{
    try
    {
        String dbQuery1 = "UPDATE " + tableName + " SET " + columnName + " = " + "'" + change + "'"
            + " WHERE DishID = " + IDNumber;
        PreparedStatement ps1 = dbConn.prepareStatement(dbQuery1);
        ps1.executeUpdate();
    } catch (SQLException ex)
    {
        SystemError objError = new SystemError("Something went wrong in the system, please contact developer");
    }
}

public void update(String tableName, int IDNumber, String columnName, int change)
{
    try
    {
        String dbQuery1 = "UPDATE " + tableName + " SET " + columnName + " = " + change +
            " WHERE DishID = " + IDNumber;
        PreparedStatement ps1 = dbConn.prepareStatement(dbQuery1);
        ps1.executeUpdate();
    } catch (SQLException ex)
    {
        SystemError objError = new SystemError("Something went wrong in the system, please contact developer");
    }
}
```

Same name

String type

double type

String Type

Int Stype

**Figure 28.** Example of Method with same name but different parameter



Insert data method, in JavaDatabase class, used overloading to have the same name but with different methods. Because there are columns in my database that can either be an integer or a string, it is easier and appropriate for me to have one method that has a same functionality and can be flexible in which type of data is imputed.

## Encapsulation

```
public class CombineTable
{
    private ArrayList<ArrayList<String>> dataCombine;
    private ArrayList<ArrayList<String>> dataTable1;
    private ArrayList<ArrayList<String>> dataTable2;
    private ArrayList<ArrayList<String>> dataTable3;
    private ArrayList<ArrayList<String>> dataTable4;
    private ArrayList<ArrayList<String>> dataTable5;

    private int tableCount;

    public CombineTable()
    {
        this.dataCombine = new ArrayList<>();
        this.dataTable1 = new ArrayList<>();
        this.dataTable2 = new ArrayList<>();
        this.dataTable3 = new ArrayList<>();
        this.dataTable4 = new ArrayList<>();
        this.dataTable5 = new ArrayList<>();
    }
}
```

**Figure 29.** Example of encapsulation by declaring data private

```

public void setTableCount(int tableCount)
{
    this.tableCount = tableCount;
}

public ArrayList<ArrayList<String>> getDataCombine()
{
    return dataCombine;
}

public ArrayList<ArrayList<String>> getDataTable1()
{
    return dataTable1;
}

public ArrayList<ArrayList<String>> getDataTable2()
{
    return dataTable2;
}

public ArrayList<ArrayList<String>> getDataTable3()
{
    return dataTable3;
}

public ArrayList<ArrayList<String>> getDataTable4()
{
    return dataTable4;
}

public ArrayList<ArrayList<String>> getDataTable5()
{
    return dataTable5;
}

public int getTableCount()
{
    return tableCount;
}

```

**Figure 30.** Example of encapsulation, using getter and setter to access the data

Encapsulation is used to hide data from other classes. It helps with managing data. It also helps with fixing the program or method because only one change is needed to change all the associated methods.

## Inheritance

```

public class Welcome extends JFrame implements ActionListener

```

**Figure 31.** Example of Inheritance, Welcome class inherit JFrame class

## Iteration

```
//Combining Table back in order
if (columnCount == 3)
{
    for (int i = 0; i < selectedRatio.size(); i++)
    {
        ArrayList<String> ratioDishID = selectedRatio.get(i);
        String dishID1 = ratioDishID.get(0);
        String ratio = ratioDishID.get(1);
        for (int y = 0; y < dataHolder.size(); y++)
        {
            ArrayList<String> temp = new ArrayList<>();
            ArrayList<String> unsortData = dataHolder.get(y);
            String dishID2 = unsortData.get(0);
            if (dishID1.equals(dishID2))
            {
                temp.add(dishID1);
                temp.add(unsortData.get(1));
                temp.add(ratio);
                this.SelectedData.add(temp);
            }
        }
    }
}
```

**Figure 32.** Example of iteration, using loops to combine data accordingly to the dishID order

## Exception Handling

Exception handling is used to handle exceptions in the program. This is used to create robustness. It detects if the user has entered a wrong input format or forgot to input. It is also used to handle anything that goes wrong in the program.

## Selection Sort

```

public void setSortedData()
{
    Double[] arID = new Double[this.unsortedData.size()];
    Double[] arRatio = new Double[this.unsortedData.size()];
    for (int i = 0; i < this.unsortedData.size(); i++)
    {
        ArrayList<Double> tempData = this.unsortedData.get(i);

        arID[i] = tempData.get(0);
        arRatio[i] = tempData.get(1);
    }
    for (int i = 0; i < arRatio.length; i++)
    {
        int min = i;
        for (int j = i + 1; j < arRatio.length; j++)
        {
            if (arRatio[j] < arRatio[min])
            {
                min = j;
            }
        }
        //Swapping
        double temp = arRatio[min];
        arRatio[min] = arRatio[i];
        arRatio[i] = temp;

        double temp1 = arID[min];
        arID[min] = arID[i];
        arID[i] = temp1;
    }

    for (int i = 0; i < this.unsortedData.size(); i++)
    {
        ArrayList<Double> tempData = new ArrayList<>();
        tempData.add(arID[i]);
        tempData.add(arRatio[i]);
        this.sortedData.add(tempData);
    }
}

```

**Figure 33.** Selection sort is used to sort an ArrayList

Using the Selection sort algorithm to sort Array in order of number from high to low. Selection sort is used because it is simple and easy to construct. It is used to sort ArrayList because it can manipulate multiple columns at once just by sorting one column.

## ArrayList

ArrayList is used to store data and mostly used to store data in databases. It is flexible and dynamic, which means that the size of the ArrayList doesn't have to be declared. Therefore, it is ideal to store manipulated data where the size is not predefined.

## Array

This is used to store numerical data to be manipulated. Array is usually used for data that will be sorted through a sorting algorithm or be calculated such as the summation and the average.

### **If-Else condition**

If-Else condition is used to execute different operations based on if certain conditions are met. This is used for the algorithm to go through various cases as well as algorithms and execute a suitable algorithm for that case.

### **Accessing Database**

I used Database to store all my data in a table in the database. Accessing Database technique helps me to retrieve data for display anytime as well as manipulate data such as update or delete.

## Tools

## **To 2d Array Method**

I learned how to turn ArrayList into a two dimensional array in my computer science class> I used this method throughout most of my entire program. I used it to display data on JTable and print data in the terminal for testing purposes.

## **Java Derby**

Java Derby is used to store information in databases. Java Derby is also used to get data from the database. It is essential for this product because all data is stored in the database.

## **Java API**

Java API is used to create GUI, JFrame.