

Criterion C

Techniques used:

| | |
|--------------------------------|---|
| Graphical interface | 2 |
| Multiple windows | 2 |
| Formatting | 3 |
| Hints / Tooltips | 4 |
| Calculations | 4 |
| Object-oriented approach | 4 |
| Dynamic data structures | 5 |
| Binary search | 6 |
| Java database | 6 |
| Libraries | 6 |
| Database driver | 7 |
| Paths | 7 |
| SQL | 7 |
| Validation | 8 |

Word Count: 959

Graphical interface

The program's graphical interface (GUI) was developed using the NetBeans integrated development environment (IDE)¹. The graphical elements were created in the graphical view, although many of their properties and interactions were altered directly in the code.

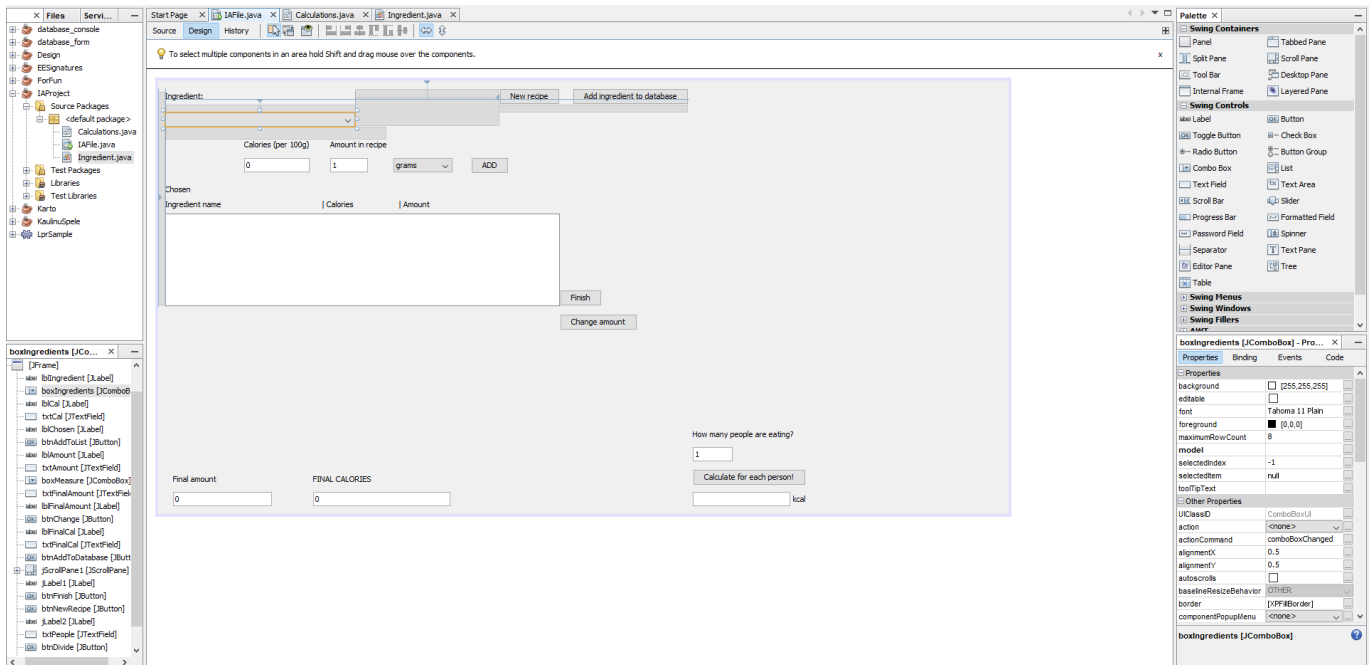


Figure 1 Creation of graphical user interface (GUI) using NetBeans' graphical elements

Multiple windows

In order for the user not to get lost in the variety of different buttons and labels, some buttons hide a different window which opens when the button is clicked, e.g., the button "Add ingredient to database" opens a dialog box² (in Figure 2) when clicked.

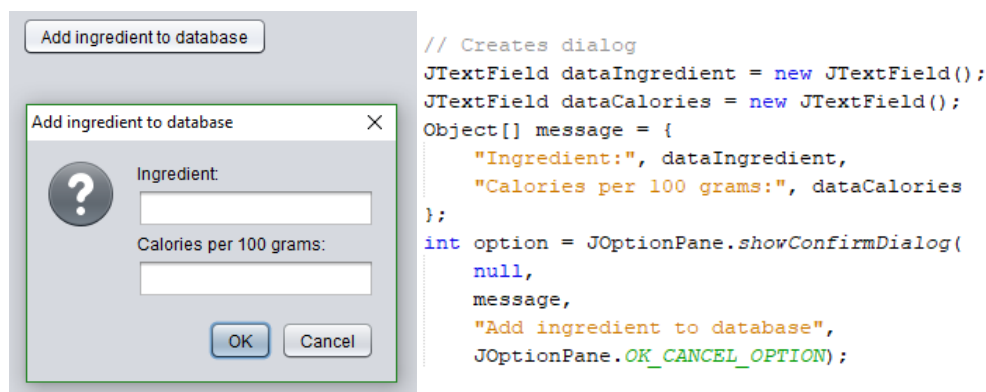


Figure 2 Dialog box shown asking the user to input the parameters of a new ingredient

¹ NetBeans IDE, Source: <https://netbeans.org/>. Last accessed 26 Jan, 2019.

² Oracle. "How to Make Dialogs", <https://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html>. Last accessed 27 Jan 2019.

Furthermore, whenever the user inputs something incorrectly, a message box warns them what exactly went wrong in a clear and understandable manner.

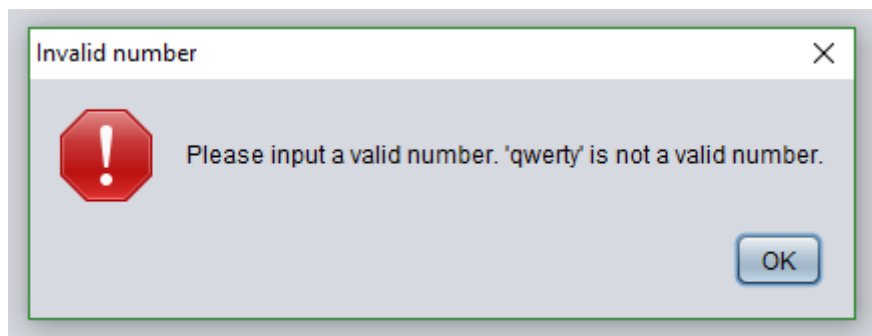


Figure 3 Message box displaying what the user did wrong

Formatting

For the user to better understand what numbers are displayed, they are formatted by the ***format*** method. This formatting specifically rounds numbers to two decimal places³ and places them with clearly visible gaps to separate them. It also uses the United States style of writing decimal numbers (with a dot for the separator)⁴ to keep the same formatting in any computer in which this program could be launched.

| Chosen | | |
|-----------------|----------|--------------|
| Ingredient name | Calories | Amount |
| Butter | 1792.00 | 250.00 grams |
| Flour | 1820.00 | 500.00 grams |
| Sugar | 193.35 | 50.00 grams |

Figure 4 Formatted JList with added ingredients

³ Mimoun-Prat, Vincent. "Double decimal formatting in Java", <https://stackoverflow.com/questions/12806278/double-decimal-formatting-in-java>. Last accessed 26 Jan 2019.

⁴ rodion. "Java: Float Formatting depends on Locale [duplicate]", <https://stackoverflow.com/questions/4553633/java-float-formatting-depends-on-locale>. Last accessed 26 Jan 2019

```

//Adding to final fields
String sumAmount = String.format(
    Locale.US, "%.2f",
    Calculations.sumAmount(ingredientArray)) + "";
String sumCalories = String.format(
    Locale.US, "%.2f",
    Calculations.sumCalories(ingredientArray)) + "";
txtFinalAmount.setText(sumAmount);
txtFinalCal.setText(sumCalories);

//Adding to list
String amountString = "" + String.format(
    Locale.US, "%.2f",
    amount);
String calcCalString = "" + String.format(
    Locale.US, "%.2f",
    calculatedCalories);
System.out.println("Amount: " + amountString + " added");
String dataString = String.format("%-69s", ingredient)
    + String.format("%-35s", calcCalString)
    + amountString
    + " " + "grams" + measurement;
listModel.addElement(dataString);

```

US locale for universal
standard in code

Two digits after decimal

A set amount of spaces

Figure 5 Formatting strings to round numbers and divide elements in the JList's row

Hints / Tooltips

To make the interface more understandable to a user not quite familiar with computers, each interactive element in the GUI displays a **tooltip**⁵ when hovered, which explains how the corresponding element works and what can be done with it, as seen in Figure 6.

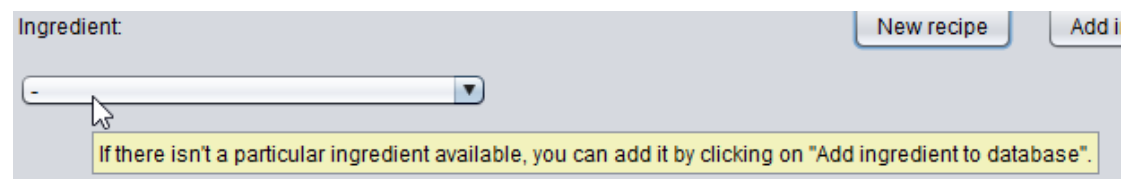


Figure 6 Helpful text showing up when user hovers over an element

```

String hint = "If you want to change an ingredient's amount, "
    + "simply remove it and add it with a different amount.";
listIngredients.setToolTipText(hint);

```

Calculations

Object-oriented approach

Since the program includes operations with ingredients, and each of them has multiple properties, that is, its name, amount and calories for that specific amount, the easiest way to deal with the problem of operating with multiple variables was to make a class of objects with these parameters called **Ingredient**. The class implements the **Comparable** interface to be able to sort and search a list of these objects.

⁵ dryairship. "Java get mouseover tooltip text", <https://stackoverflow.com/questions/36914969/java-get-mouseover-tooltip-text>. Last accessed 27 Jan 2019.

```

public class Ingredient implements Comparable{
    private String name;
    private double calories;
    private double amountIngredient;

    public Ingredient(String name, double calculatedCalories, double amount){
        this.name = name;
        this.calories = calculatedCalories;
        this.amountIngredient = amount;
    }

    public String getName(){
        return name;
    }

    public double getAmount(){
        return amountIngredient;
    }

    public double getCalories(){
        return calories;
    }

    public void setCalories(double calories) {
        this.calories = calories;
    }

    public void setAmountIngredient(double amountIngredient) {
        this.amountIngredient = amountIngredient;
    }

    @Override
    public int compareTo(Object i){
        Ingredient ing = (Ingredient)i;
        return this.getName().compareTo(ing.getName());
    }
}

```

The constructor

Getters and setters

Overriden compareTo method

Figure 7 *Ingredient()* class with appropriate getters and setters and implementation of Comparable interface

Dynamic data structures

Since there should be no limit to the number of ingredients the user adds to the recipe, and that specific number is unknown and can be varied, the ideal solution for making a list of these ingredients is through a dynamic data structure, particularly, an **ArrayList**. This **ArrayList** is made up of **Ingredient** objects and can be sorted and searched at leisure.

```
ArrayList<Ingredient> ingredientArray = new ArrayList<>();
```

Figure 8 ArrayList being initialised

```

//Adding object
Ingredient ing = new Ingredient(
    ingredient,
    calculatedCalories,
    amount);
ingredientArray.add(ing);

```

Figure 9 Creating object ing and adding it to the ArrayList

Binary search

The program required a method for removing an ingredient from a recipe without starting from the beginning either because they added the wrong one or accidentally input a wrong amount. The best way to achieve this was to use a search algorithm for **ingredientArray** in order to find the specific ingredient needing to be removed from the *ArrayList*. Sorting was possible due to the override of the *compareTo* method and the *Ingredient* class implementing *Comparable* in Figure 7. Then the search was done by binary search because it was part of the class curriculum.

```
Collections.sort(ingredientArray);  
int index = Arrays.binarySearch(  
    ingredientArray.toArray(),  
    new Ingredient(ingName, 0, 0));  
ingredientArray.remove(index);
```

Figure 10 Sorting and binary searching ArrayList of ingredients

Java database

The program stores details about ingredients (their names and calories per 100 grams) in a Java Derby **embedded database**⁶. Although the task could have been simplified by using some sort of text file to store the data, the Java Derby database provided a more sustainable and well-rounded framework with SQL support and a large storage.

Libraries

In order for the database to work, it has to use resources not available to NetBeans naturally, so the solution is to use **libraries** specifically dedicated to building and sustaining databases with information about user connections and possible database errors that could occur.

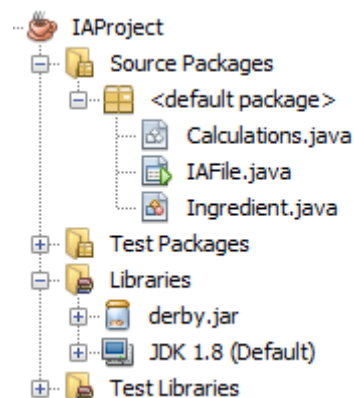


Figure 11 The layout of the files and folders of the program including a library folder with derby.jar

⁶ Apache Derby, Source: https://db.apache.org/derby/papers/DerbyTut/embedded_intro.html. Last accessed 26 Jan 2019.

Database driver

The database needs a **driver** to be built on, and each driver meets certain specific needs. This program's purpose is to work on any computer (with Java installed), no matter its connectivity to the Internet, therefore, an **embedded driver** was best suitable for these circumstances.

```
Driver derbyEmbeddedDriver = new EmbeddedDriver();
DriverManager.registerDriver(d DerbyEmbeddedDriver);

con = DriverManager.getConnection(
    "jdbc:derby:Ingredients;create=true",
    "username",
    "password");
con.setAutoCommit(false);
```

Figure 12 Registering driver and connecting to the database

Paths

Since the program depends on an **external database**, its location must be constant and always known to other parts of the code, i.e., its dependencies. Libraries also require **classpath**s, which is why the libraries are **stored relatively** (in the program's folder) to easily export it elsewhere.

```
String userHomeDir = System.getProperty("user.home", ".");
System.out.println("userHomeDir: " + userHomeDir);
String systemDir = userHomeDir + "/Recipes";

// Setting the db system directory
System.setProperty("derby.system.home", systemDir);

Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

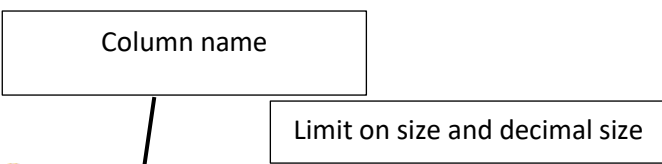
Creates a specific path
regardless of the computer

Figure 13 The directory of the database being specified⁷

SQL

The Java Derby database system supports **Structured Query Language (SQL)**, which means that tables in the database can be created with a variety of rules and factors to keep the table structured as precisely as possible, for example, with limits on characters or numbers. SQL adds the ability to define the limits with ease, as is done in Figure 13.

⁷ O'Conner, John. "Using Java DB in Desktop Applications", <https://www.oracle.com/technetwork/articles/java/javadb-141163.html>. Last accessed 27 Jan 2019.



```
String createSQL = "create table Ing3 ("
+ "id integer not null generated always as"
+ " identity (start with 1, increment by 1), "
+ "Ingredient_Name varchar(40) not null, Cal_Standard decimal(30, 2),"
+ "constraint primary_key primary key (id))";
```

Figure 14 Creation of table by specifying parameters⁸

```
pstmt = con.prepareStatement("insert into Ing3 (Ingredient_Name, Cal_Standard) values(?,?)");
pstmt.setString(1, "Butter");
pstmt.setDouble(2, 716.8);
pstmt.executeUpdate();
```

Figure 15 Inserting initial data into table

Validation

To equip the program with precautions and potential user mistakes, each field **validates** the data input to be able to operate with it afterwards. The program implements both limits for numbers and number validations.

```
public Double tryParse(String text) {
    try {
        works = true;
        return Double.parseDouble(text);
    }
    catch (NumberFormatException e) {
        JOptionPane.showMessageDialog(null,
            "Please input a valid number. '"
            + text
            + "' is not a valid number.",
            "Invalid number",
            JOptionPane.ERROR_MESSAGE);
        works = false;
        return null;
    }
}
```

Figure 16 Checking whether input text is a number

⁸ Debnath, Manoj. "Working with Embedded Databases in Java", <https://www.developer.com/java/j2me/working-with-embedded-databases-in-java.html>. Last accessed 27 Jan 2019.