Criterium C

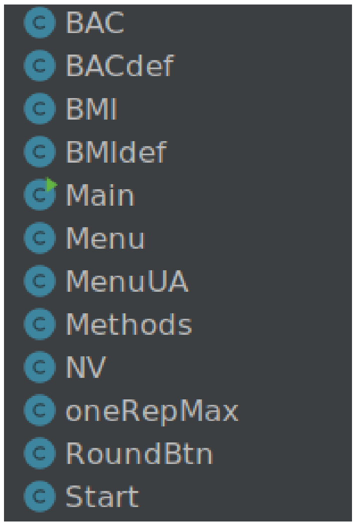
List of techniques:

- **Class Decomposition**
- **User input, GUI Interface**
- **Void and return methods**
- **Error checking (try-catch)**
- **Local and global variables**
- **Class extension and implementation of Action Listener**
- **Switch case**
- **Inheritance and Polymorphism**

Please see Appendix D where the whole code is attached.

Class Decomposition

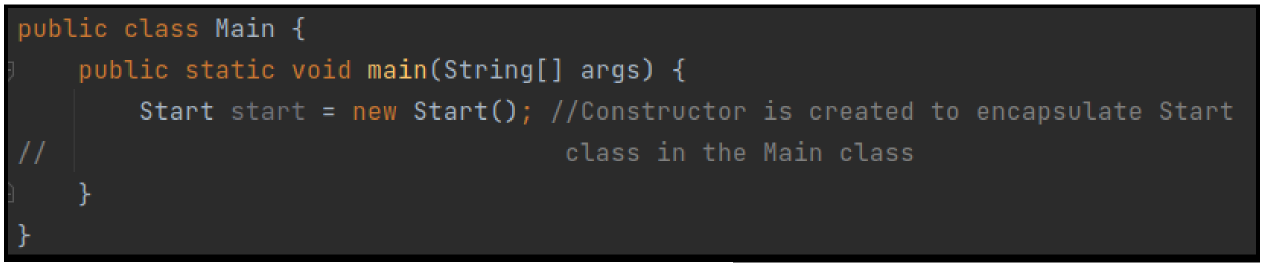
In order to make the program more readable for developers, the code is divided into multiple classes.
That allows to divide one huge problem into many smaller problems, instead of writing it as one huge block of code, which would be unfunctional - finding anything would be nearly impossible, and fixing errors would be extremely hard. Due to decomposition - each class has a certain role and performs its own tasks.
I divided the program into 12 classes, as shown in figure 1.



```
BAC
BACdef
BMI
BMIdef
Main
Menu
MenuUA
Methods
NV
oneRepMax
RoundBtn
Start
```

Figure 1. All classes used in the program.

Most classes are responsible for their own windows, despite three of them: **Main**, **RoundBtn**, and **Methods**. The **Main** is responsible for running the app(see Figure 2), the **RoundBtn** uses Java's Swing library and is responsible for rounding buttons easily(that way the app-using experience is much more pleasurable). The **Methods** has - as the name says - methods, responsible for creating components in other classes in an easier way. However, I am going to get deeply into that under the "Inheritance and polymorphism" section.



```
public class Main {
    public static void main(String[] args) {
        Start start = new Start(); //Constructor is created to encapsulate Start
//                                  class in the Main class
    }
}
```

Figure 2. Main class of the program responsible for running it

User input, GUI Interface

User input allows the user to input values into the app. GUI interface allows a user to connect with a computer using symbols, visual metaphors, and pointing devices.[1]

For the application to be usable by clients, they need to be able to input their own values. Thus, I used **JTextField**, **JComboBox**, and **JRadioButton** for that. I used various input types as the user is going to either choose one of two values(**JRadioButton**), one of many values(**JComboBox**) or write their own value(**JTextField**). That way, users can easily see and deduce what they are supposed to do.

[1] (Levy, 2018)

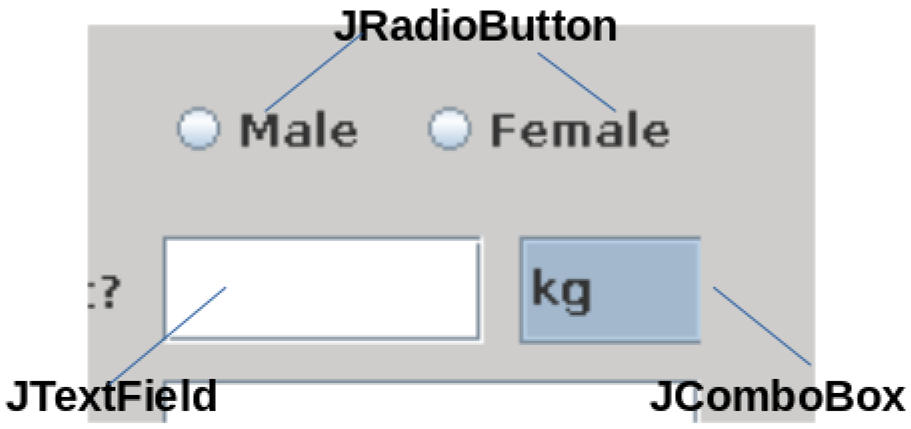
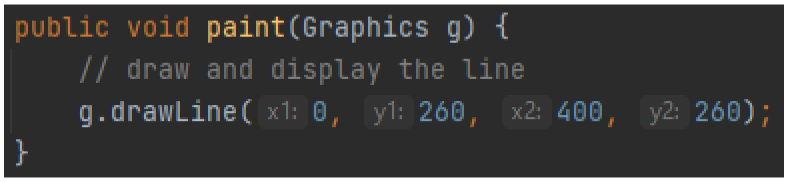


Figure 3. User input fields.

Implementation of those components is available due to Java's **GUI Swing** library. The program also uses a user-friendly GUI interface in order to both encourage the client to use the app and to make it much easier and simpler. That's the most suitable option, as using the app in a console only would be much harder and more complicated. There are also tips for users to use the program correctly, i.e. information to fill all the fields. Moreover, the GUI windows allow fulfilling the second, fourth, sixth, and eighth success criteria (see Crit. A), by allowing to display the result or by creating two different independent menus.
Introducing user input allows users to insert their data which enables the program to perform necessary calculations to obtain the required result. That way, the first, third, fifth, and seventh success criteria can be fulfilled (see Crit A).

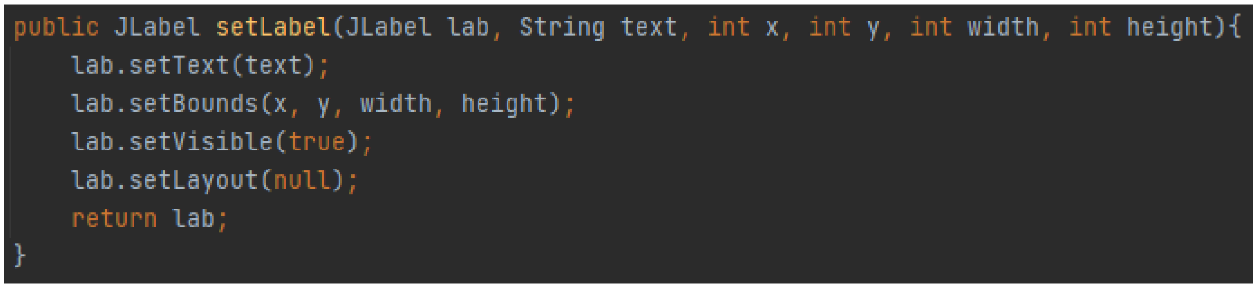
Void and return methods

As methods' purpose is to perform a task, performing anything would be impossible without them. Return methods are used in functions that need to return something, while void methods don't return anything. Thus, implementing the usage of void and return methods and functions was necessary for my program, as without them it simply wouldn't work or would work improperly. For instance, if there were only local variables, then classes such as **Methods** would have no use. Also, as the program consists of numerous functions working together, using only variables that can be used in their scope, the development of the app would be impossible. As for void methods, I used them for example while displaying lines, as that action requires no return (see figure 4). Return methods were used to avoid duplication of code by establishing some of the features of buttons, text fields, and labels. They are implemented in the aforementioned **Methods** class(see example in Figure 5). That way, a **JButton**, **JTextField**, **JRadioButton**, or many other components can be coded with just one line(see example in figure 6).



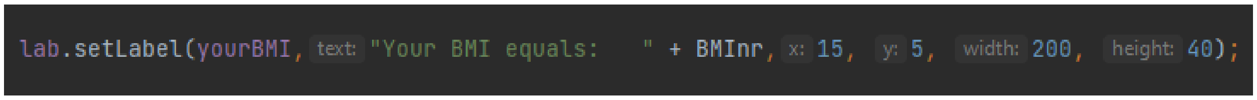
```
public void paint(Graphics g) {
    // draw and display the line
    g.drawLine( x1: 0,  y1: 260,  x2: 400,  y2: 260);
}
```

Figure 4. Example of void method - **paint()**



```
public JLabel setLabel(JLabel lab, String text, int x, int y, int width, int height){
    lab.setText(text);
    lab.setBounds(x, y, width, height);
    lab.setVisible(true);
    lab.setLayout(null);
    return lab;
}
```

Figure 5. Example of return method - **setLabel()**



```
lab.setLabel(yourBMI, text: "Your BMI equals:   " + BMInr, x: 15,  y: 5,  width: 200,  height: 40);
```

Figure 6. Example of coding a Label in one line using a return method(**setLabel()**) created in class **Methods**

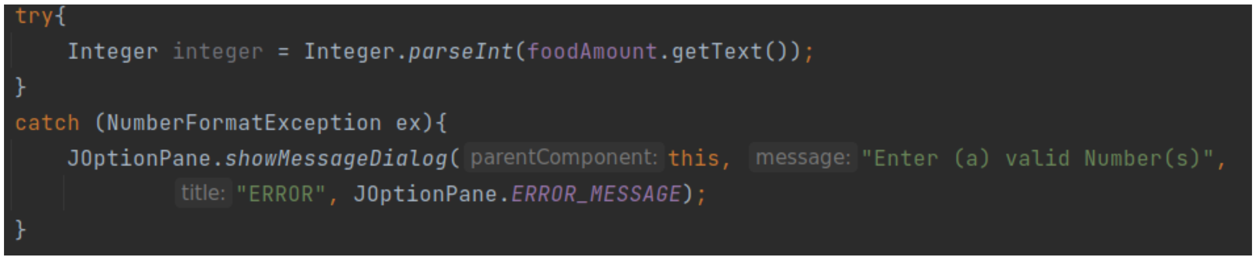
Error checking (try catch)

For error handling, I decided on using an error checking method called "**try…catch**". Its use allows for reducing errors to the minimum. Try catch is divided into two sections. A **try** block is a place for code that is checked for mistakes while program runs. If an error occurs in the try block, code from the **catch** block is executed.[2]

I implemented it so that each time when a user does not enter all variables or enters invalid value(not an integer), the error is shown informing the user to check whether all values are inputted correctly. In the case of my app, if the user misses a window or input's incorrect, then a window appears that says "Enter (a) valid Number(s)". That way, the ninth success criterion

[2] (*Java Exceptions (Try...Catch)*, 2020)

can be fulfilled (All criteria are in Crit A). I was also thinking about implementing the **Finally** statements, but as it runs after **try…catch** regardless of the result, it wasn't necessary because users do not need confirmation that they inputted data correctly - the result of calculations will be sufficient indicator of that.

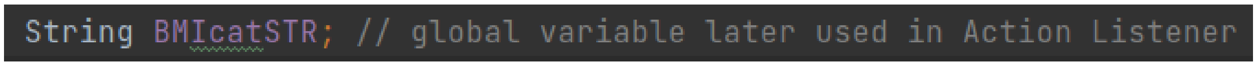


```
try{
    Integer integer = Integer.parseInt(foodAmount.getText());
}
catch (NumberFormatException ex){
    JOptionPane.showMessageDialog( parentComponent: this,  message: "Enter (a) valid Number(s)",
             title: "ERROR", JOptionPane.ERROR_MESSAGE);
}
```

Figure 7. Example of "**try…catch**" error checking

Local and global variables

Local variables are defined inside a function and have a scope that is limited to that function only, whereas global variables are defined outside of a function and have a global scope.
Without global variables, it would be impossible to use Action Listener, which is an essential part of the program and it would not work without it, as no button would work.

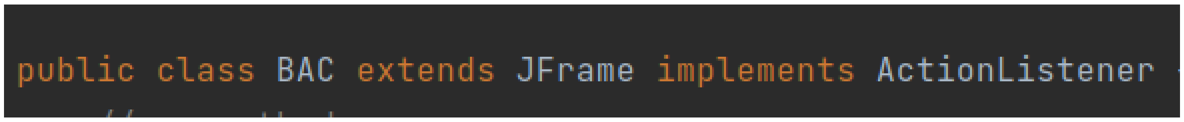


```
String BMIcatSTR; // global variable later used in Action Listener
```

Figure 8. Example of a global variable.

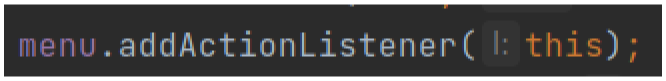
Class extension and implementation of Action Listener

The Action Listener extension is in charge of all action events, such as when a user clicks on a component.
Most classes are implemented with an **"Action Listener**" function. That allows for the usage of **JButtons**. Without that function, the program using GUI would not be functional at all, as it would be impossible to exit the start window, because no button would work. Moreover, as I already mentioned, using GUI is the most suitable, as using my app in a console without the GUI would be much harder and more complex. Implementation of the Action Listener allows to fully use and take advantage of the program.



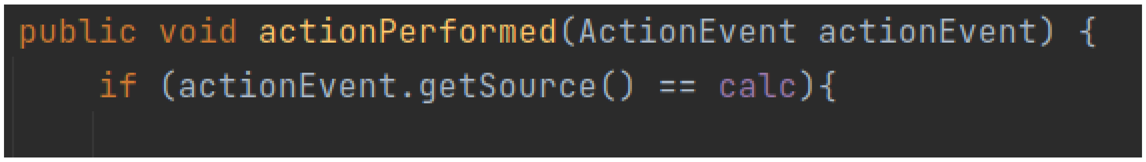
```
public class BAC extends JFrame implements ActionListener
```

Figure 9. Example of implementing **ActionListener** to a class - here, to class **BAC** (for other examples please see appendix D).



```
menu.addActionListener( l: this);
```

Figure 10. Example of implementing **ActionListener** to a button - here, a button called **menu**(for other examples please see appendix D).

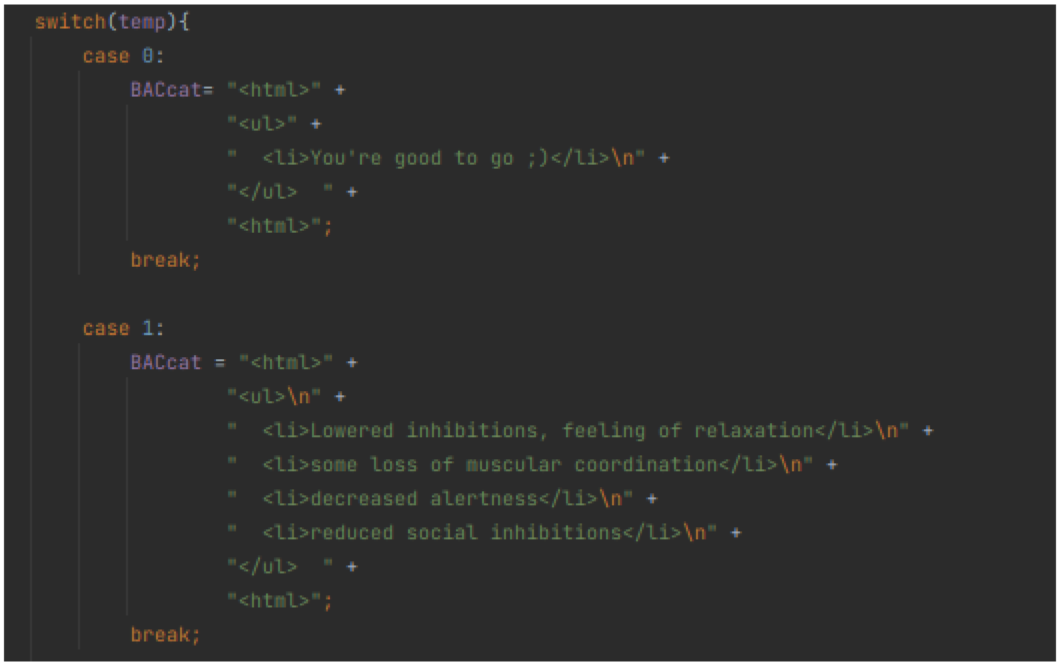


```
public void actionPerformed(ActionEvent actionEvent) {
    if (actionEvent.getSource() == calc){
```

Figure 11. Example of a conditional statement that allows **ActionListener** to work- here, with a button called **calc** (for other examples please see appendix D).

Switch case

A conditional statement, such as **Switch case** is a statement used to select one of many blocks of code to be executed. I decided on using the **Switch case** in some places instead of the **if** statement because it is more suitable for the program to run once instead of going through multiple **if** statements with nearly identical conditions. It allows comparing a variable to a list of values to see if they are equal. If they are, then, for instance, a variable can have different values assigned (see example in figure 9).



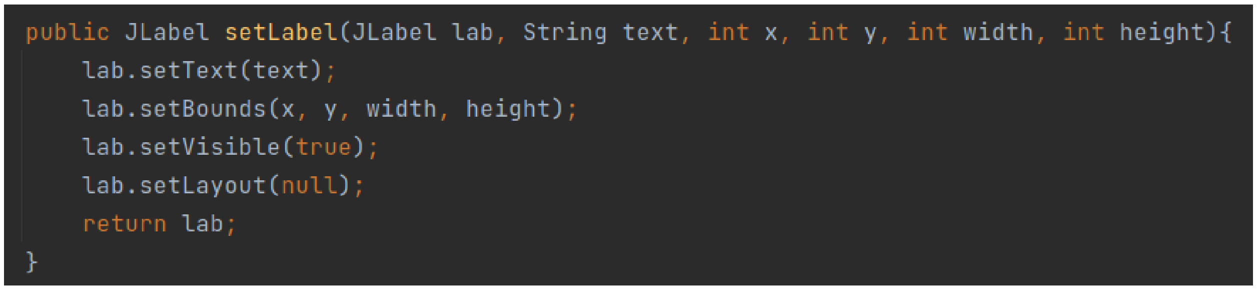
```
switch(temp){
    case 0:
        BACcat= "<html>" +
                "<ul>" +
                "   <li>You're good to go ;)</li>\n" +
                "</ul>   " +
                "<html>";
        break;

    case 1:
        BACcat = "<html>" +
                "<ul>\n" +
                "   <li>Lowered inhibitions, feeling of relaxation</li>\n" +
                "   <li>some loss of muscular coordination</li>\n" +
                "   <li>decreased alertness</li>\n" +
                "   <li>reduced social inhibitions</li>\n" +
                "</ul>   " +
                "<html>";
        break;
```

Figure 12. A part of the example of a **switch case** statement.

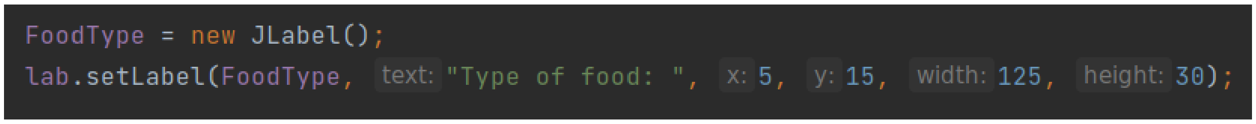
Inheritance and Polymorphism

Polymorphism is a situation in which many classes are related to each other through inheritance. Meanwhile, Inheritance is a technique by which one object inherits all of its parent's properties and actions.
As mentioned in the beginning, I used classes **RoundBtn**, and **Methods** in order to make the code more readable. More specifically, those methods are coded in only one class and not in each class where they are used, that way much less time, energy, and space in the IDE is used. Those methods are used numerous times across the program. Examples can be seen in figures 10 and 11.



```java
public JLabel setLabel(JLabel lab, String text, int x, int y, int width, int height){
    lab.setText(text);
    lab.setBounds(x, y, width, height);
    lab.setVisible(true);
    lab.setLayout(null);
    return lab;
}
```

Figure 13. Code retrieved from the Methods class, responsible for JLabels



```java
FoodType = new JLabel();
lab.setLabel(FoodType,  text: "Type of food: ",  x: 5,  y: 15,  width: 125,  height: 30);
```

Figure 14. Code retrieved from the class NV, that uses the snipped mentioned in figure 7.

Word count: 1144

References

Horstmann, C. S. (2019). *Core Java*. Pearson.

Java Exceptions (Try...Catch). (2020). W3schools.com.
https://www.w3schools.com/java/java_try_catch.asp

Levy, S. (2018). Graphical user interface | computing. In *Encyclopædia Britannica*.
https://www.britannica.com/technology/graphical-user-interface

Wright, C. (2003). *Java*. Hodder Headline.