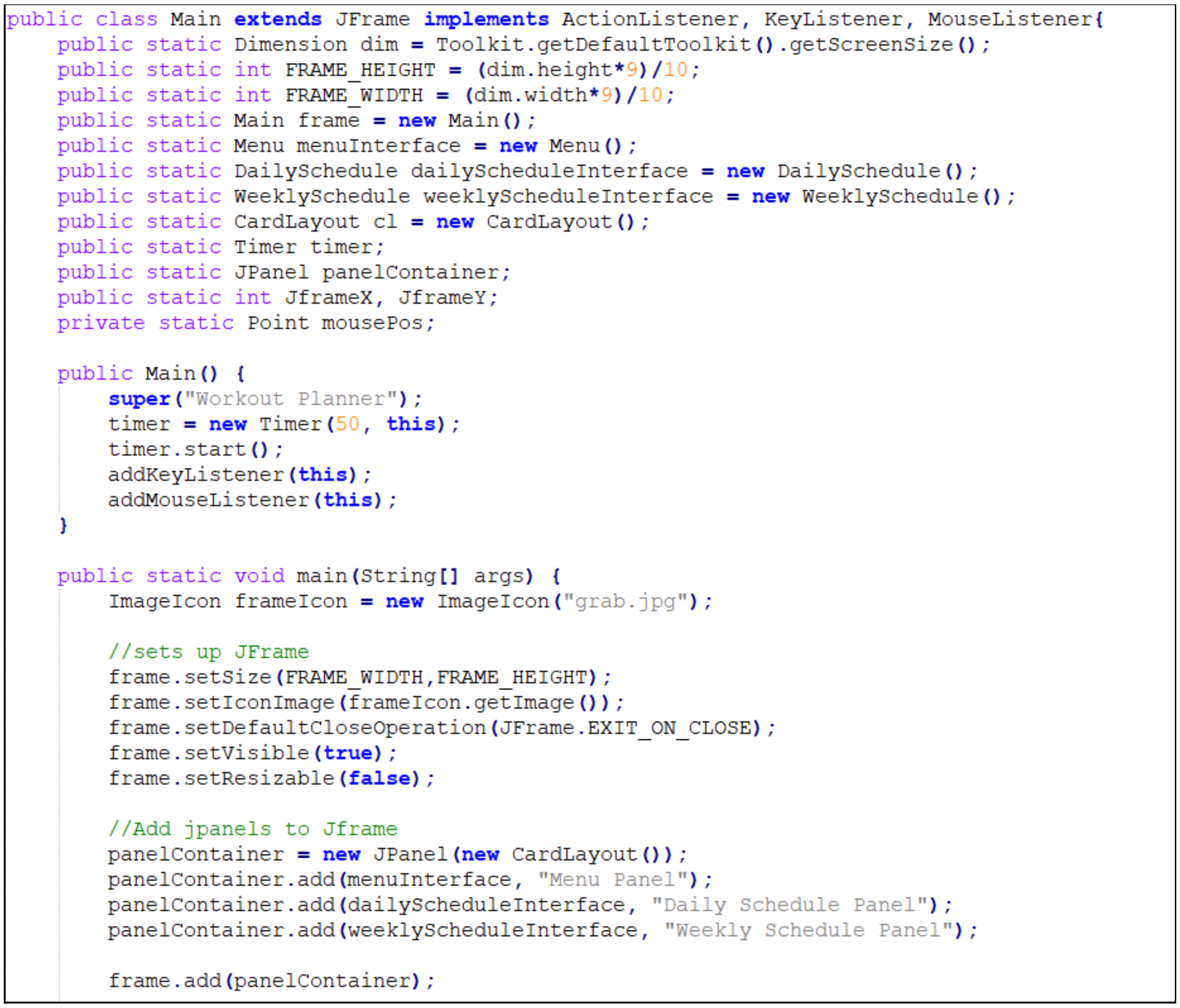
Criterion C: Development

Object Oriented Programming (OOP)

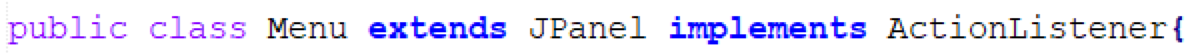
Inheritance

Inheritance refers to the process of allowing one class, referred to as the child class, to access the methods and variables of another class, referred to as the parent class. This process was implemented throughout the entire program, as all of the functionalities were presented via a graphical user interface (GUI). As such, each of the functionalities were programmed under classes which implemented various GUI parent classes, such as JFrame and JPanel.

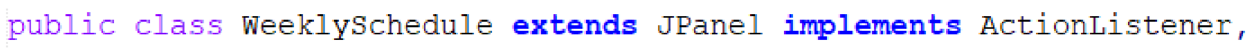


```
public class Main extends JFrame implements ActionListener, KeyListener, MouseListener{
    public static Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
    public static int FRAME_HEIGHT = (dim.height*9)/10;
    public static int FRAME_WIDTH = (dim.width*9)/10;
    public static Main frame = new Main();
    public static Menu menuInterface = new Menu();
    public static DailySchedule dailyScheduleInterface = new DailySchedule();
    public static WeeklySchedule weeklyScheduleInterface = new WeeklySchedule();
    public static CardLayout cl = new CardLayout();
    public static Timer timer;
    public static JPanel panelContainer;
    public static int JframeX, JframeY;
    private static Point mousePos;

    public Main() {
        super("Workout Planner");
        timer = new Timer(50, this);
        timer.start();
        addKeyListener(this);
        addMouseListener(this);
    }

    public static void main(String[] args) {
        ImageIcon frameIcon = new ImageIcon("grab.jpg");

        //sets up JFrame
        frame.setSize(FRAME_WIDTH,FRAME_HEIGHT);
        frame.setIconImage(frameIcon.getImage());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
        frame.setResizable(false);

        //Add jpanels to Jframe
        panelContainer = new JPanel(new CardLayout());
        panelContainer.add(menuInterface, "Menu Panel");
        panelContainer.add(dailyScheduleInterface, "Daily Schedule Panel");
        panelContainer.add(weeklyScheduleInterface, "Weekly Schedule Panel");

        frame.add(panelContainer);
```

The Main class is a child class of JFrame, and as such can access all of the methods in the JFrame class, such as .setSize(), and .add(). The **super** keyword is used to call the constructor of the parent class, creating an implicit instance of the parent class along with the child class.

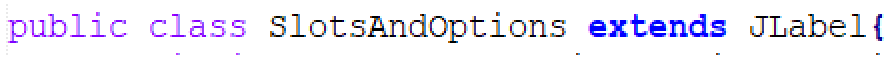


```java
public class Menu extends JPanel implements ActionListener{
```



```java
public class WeeklySchedule extends JPanel implements ActionListener,
```

My user interfaces, the home menu and schedule menu are both child classes of the JPanel class, allowing them to be added to the JFrame of the main class.

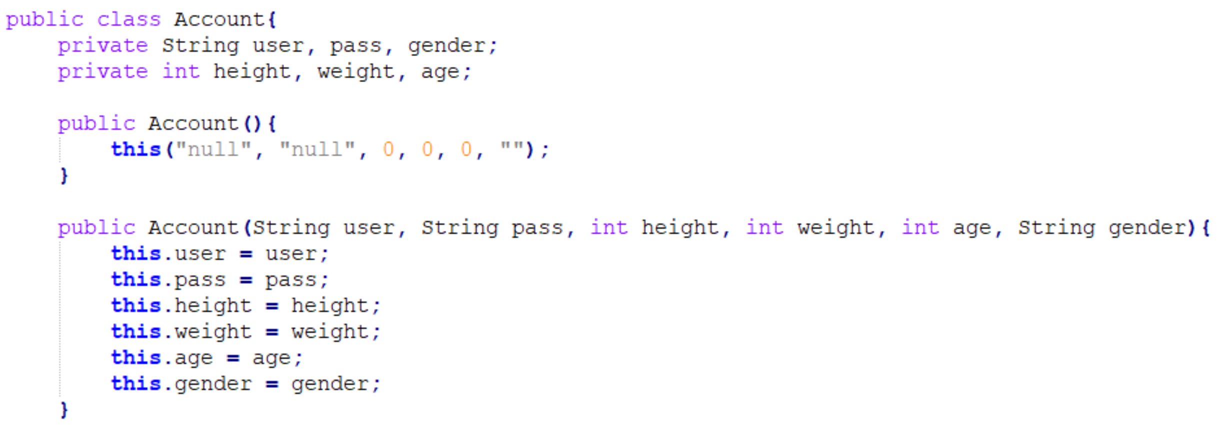


```java
public class SlotsAndOptions extends JLabel{
```

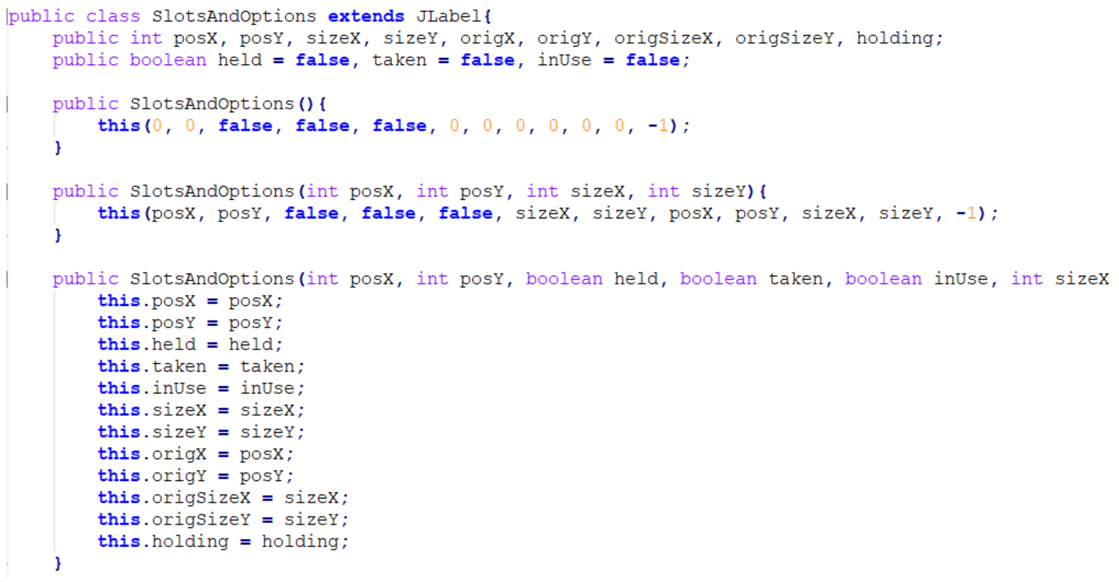
My SlotsAndOptions class, which is a class used to create time slots and exercise options, extends the parent class of JLabel. This allows me to add them onto my program in the form of a JLabel to thus freely manipulate their contents and location, while also adding more information such as its size, position, and occupancy.

Polymorphism

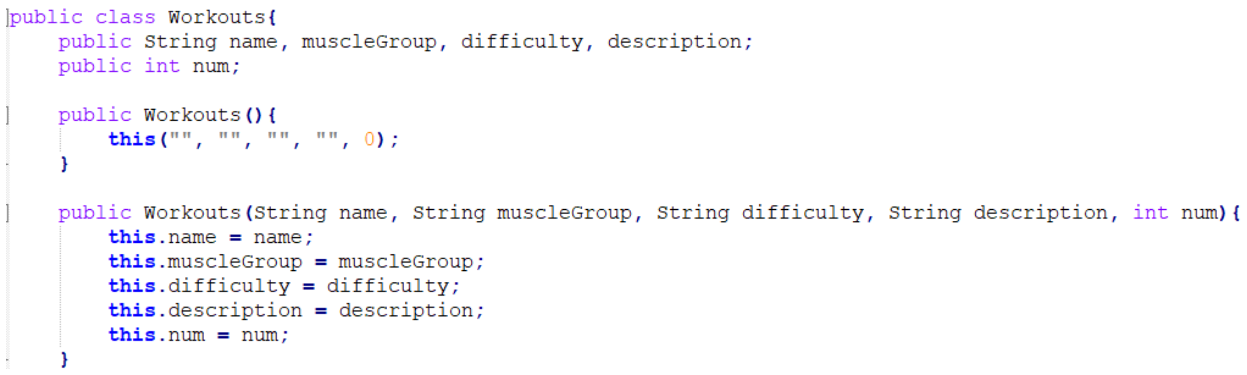
Polymorphism was incorporated in my program through constructor overloading, which is the process of creating multiple of the same constructors that each accept different arguments.



```java
public class Account{
    private String user, pass, gender;
    private int height, weight, age;

    public Account(){
        this("null", "null", 0, 0, 0, "");
    }

    public Account(String user, String pass, int height, int weight, int age, String gender){
        this.user = user;
        this.pass = pass;
        this.height = height;
        this.weight = weight;
        this.age = age;
        this.gender = gender;
    }
```



```java
public class SlotsAndOptions extends JLabel{
    public int posX, posY, sizeX, sizeY, origX, origY, origSizeX, origSizeY, holding;
    public boolean held = false, taken = false, inUse = false;

    public SlotsAndOptions(){
        this(0, 0, false, false, false, 0, 0, 0, 0, 0, 0, -1);
    }

    public SlotsAndOptions(int posX, int posY, int sizeX, int sizeY){
        this(posX, posY, false, false, false, sizeX, sizeY, posX, posY, sizeX, sizeY, -1);
    }

    public SlotsAndOptions(int posX, int posY, boolean held, boolean taken, boolean inUse, int sizeX
        this.posX = posX;
        this.posY = posY;
        this.held = held;
        this.taken = taken;
        this.inUse = inUse;
        this.sizeX = sizeX;
        this.sizeY = sizeY;
        this.origX = posX;
        this.origY = posY;
        this.origSizeX = sizeX;
        this.origSizeY = sizeY;
        this.holding = holding;
    }
```

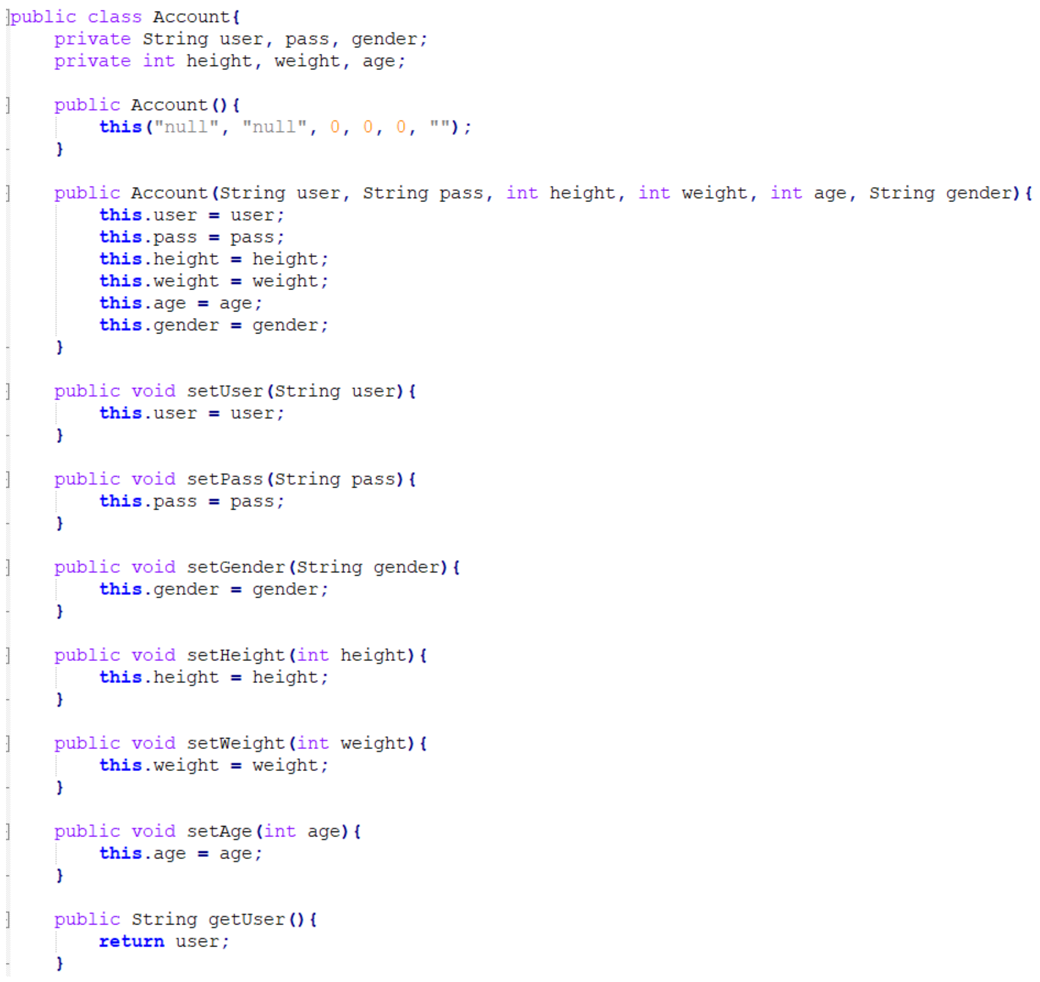


```java
public class Workouts{
    public String name, muscleGroup, difficulty, description;
    public int num;

    public Workouts(){
        this("", "", "", "", 0);
    }

    public Workouts(String name, String muscleGroup, String difficulty, String description, int num){
        this.name = name;
        this.muscleGroup = muscleGroup;
        this.difficulty = difficulty;
        this.description = description;
        this.num = num;
    }
```

In my classes Account, SlotsAndOptions, and Workouts, I have a blind constructor which creates a default object for each class with empty variables, as well as a main constructor that creates an object with specified arguments.

Encapsulation

Encapsulation is the process of packaging variables and methods under a single class to privatize the information and protect it from outside access. This means creating private variables with public methods to modify and retrieve data.

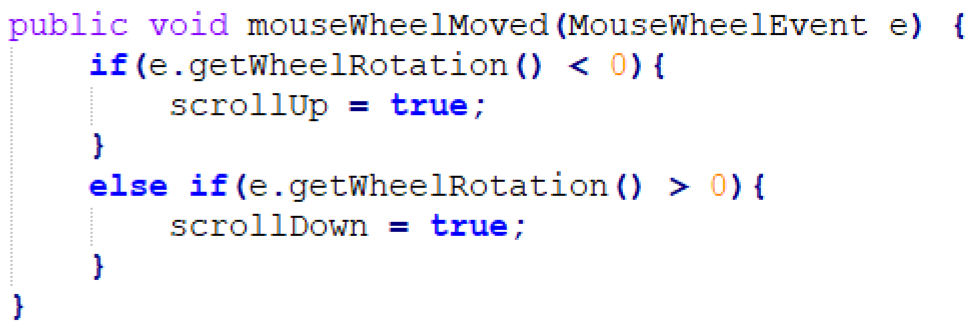


```java
public class Account{
    private String user, pass, gender;
    private int height, weight, age;

    public Account(){
        this("null", "null", 0, 0, 0, "");
    }

    public Account(String user, String pass, int height, int weight, int age, String gender){
        this.user = user;
        this.pass = pass;
        this.height = height;
        this.weight = weight;
        this.age = age;
        this.gender = gender;
    }

    public void setUser(String user){
        this.user = user;
    }

    public void setPass(String pass){
        this.pass = pass;
    }

    public void setGender(String gender){
        this.gender = gender;
    }

    public void setHeight(int height){
        this.height = height;
    }

    public void setWeight(int weight){
        this.weight = weight;
    }

    public void setAge(int age){
        this.age = age;
    }

    public String getUser(){
        return user;
    }
```

In my Account class, all the instance variables are private, however there are many methods to set and get the data for this class.

Abstraction

Abstraction helps hide unnecessary complicated code and only reveals what is necessary. In my program, I implemented many interfaces with overriding methods. This is a form of abstraction because I am able to use the functionalities provided in the methods of the interface, and all of the complicated code behind the methods are hidden away.



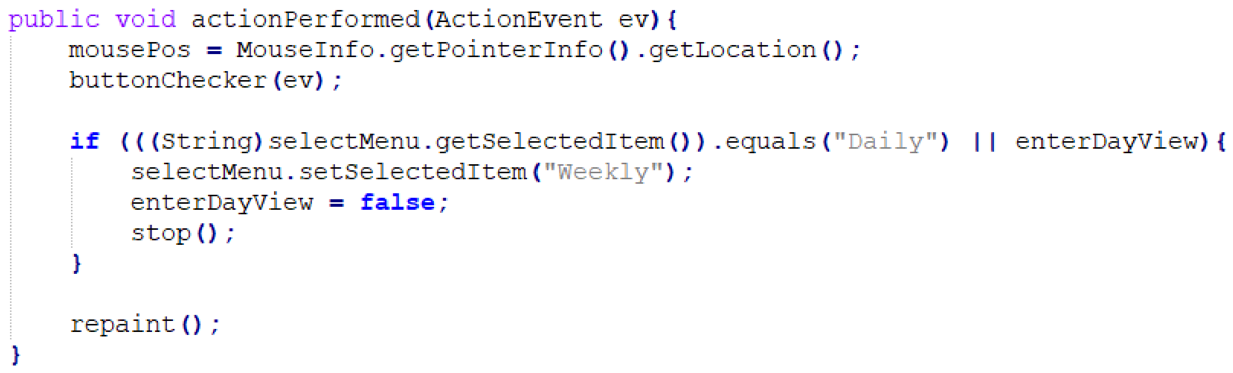
```java
public void mouseWheelMoved(MouseWheelEvent e) {
    if(e.getWheelRotation() < 0){
        scrollUp = true;
    }
    else if(e.getWheelRotation() > 0){
        scrollDown = true;
    }
}
```

For example, I implemented the interface MouseWheelListener, and thus I was able to use and override their methods, without having to interact with the underlying complicated code.

Input Detection

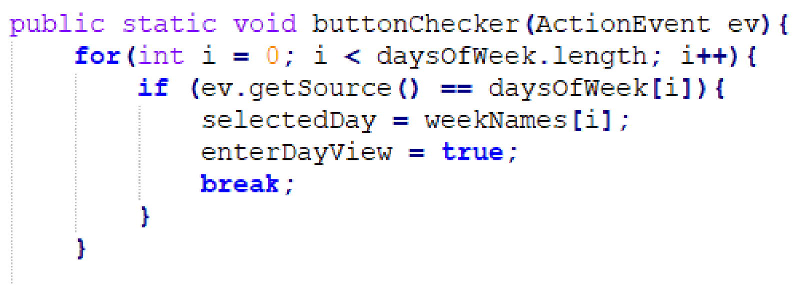
ActionListener

ActionListener is an interface implemented on all interfaces to detect user input.



```java
public void actionPerformed(ActionEvent ev){
    mousePos = MouseInfo.getPointerInfo().getLocation();
    buttonChecker(ev);

    if (((String)selectMenu.getSelectedItem()).equals("Daily") || enterDayView){
        selectMenu.setSelectedItem("Weekly");
        enterDayView = false;
        stop();
    }

    repaint();
}
```

In my schedule class, my actionPerformed method overrides the method in ActionListener, and calls my buttonChecker method, which determines which button, if any, has been pressed. The if statement checks the status of my JComboBox to determine whether the user has decided to switch schedules.

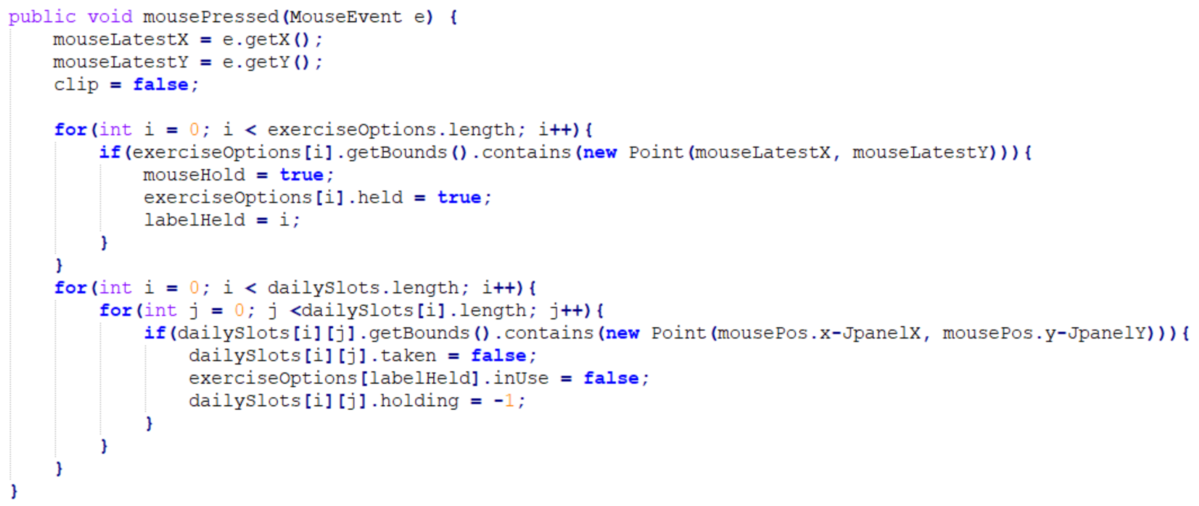


```java
public static void buttonChecker(ActionEvent ev){
    for(int i = 0; i < daysOfWeek.length; i++){
        if (ev.getSource() == daysOfWeek[i]){
            selectedDay = weekNames[i];
            enterDayView = true;
            break;
        }
    }
```

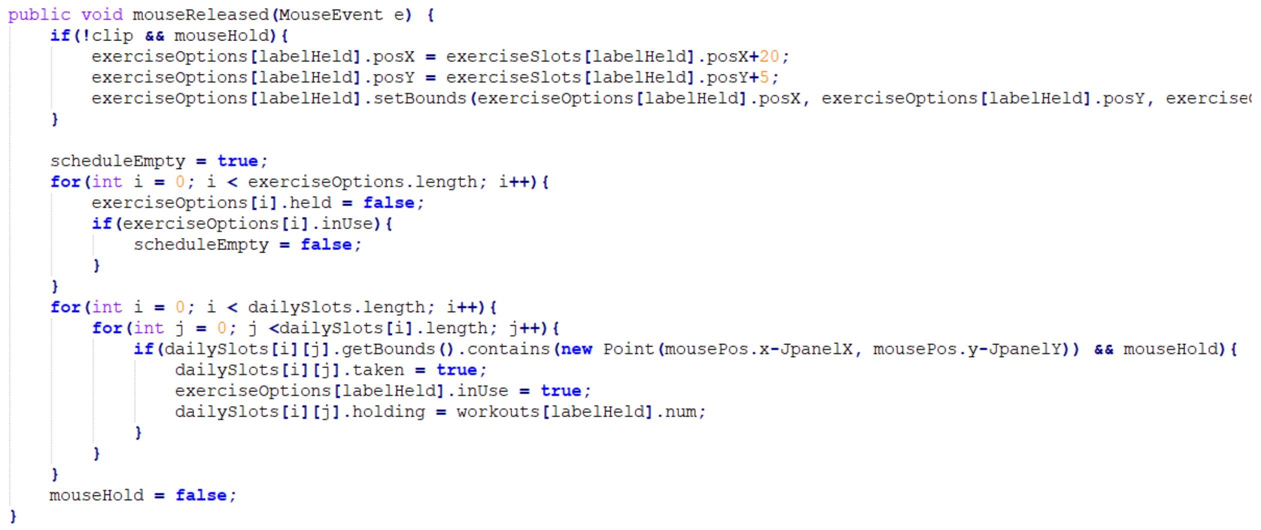
The first for loop in my buttonChecker method checks to see if the user decided to enter the daily schedule, as well as which day of the week they selected.

MouseListener

The MouseListener interface was used to determine which elements in my program have been clicked, and whether or not the mouse was pressed or released.



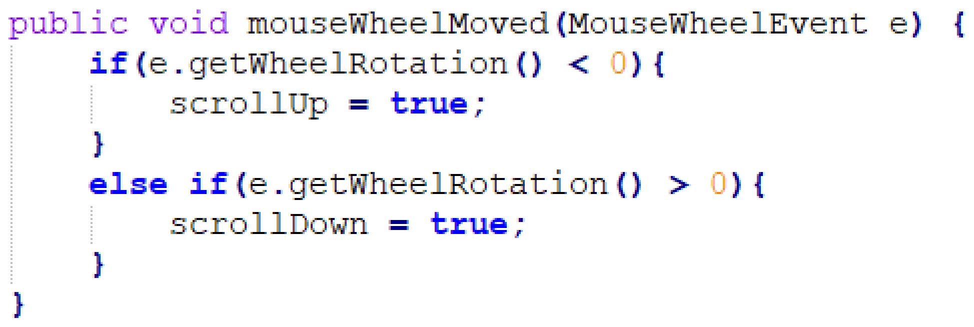
```java
public void mousePressed(MouseEvent e) {
    mouseLatestX = e.getX();
    mouseLatestY = e.getY();
    clip = false;

    for(int i = 0; i < exerciseOptions.length; i++){
        if(exerciseOptions[i].getBounds().contains(new Point(mouseLatestX, mouseLatestY))){
            mouseHold = true;
            exerciseOptions[i].held = true;
            labelHeld = i;
        }
    }
    for(int i = 0; i < dailySlots.length; i++){
        for(int j = 0; j <dailySlots[i].length; j++){
            if(dailySlots[i][j].getBounds().contains(new Point(mousePos.x-JpanelX, mousePos.y-JpanelY))){
                dailySlots[i][j].taken = false;
                exerciseOptions[labelHeld].inUse = false;
                dailySlots[i][j].holding = -1;
            }
        }
    }
}
```



```java
public void mouseReleased(MouseEvent e) {
    if(!clip && mouseHold){
        exerciseOptions[labelHeld].posX = exerciseSlots[labelHeld].posX+20;
        exerciseOptions[labelHeld].posY = exerciseSlots[labelHeld].posY+5;
        exerciseOptions[labelHeld].setBounds(exerciseOptions[labelHeld].posX, exerciseOptions[labelHeld].posY, exercise
    }

    scheduleEmpty = true;
    for(int i = 0; i < exerciseOptions.length; i++){
        exerciseOptions[i].held = false;
        if(exerciseOptions[i].inUse){
            scheduleEmpty = false;
        }
    }
    for(int i = 0; i < dailySlots.length; i++){
        for(int j = 0; j <dailySlots[i].length; j++){
            if(dailySlots[i][j].getBounds().contains(new Point(mousePos.x-JpanelX, mousePos.y-JpanelY)) && mouseHold){
                dailySlots[i][j].taken = true;
                exerciseOptions[labelHeld].inUse = true;
                dailySlots[i][j].holding = workouts[labelHeld].num;
            }
        }
    }
    mouseHold = false;
}
```

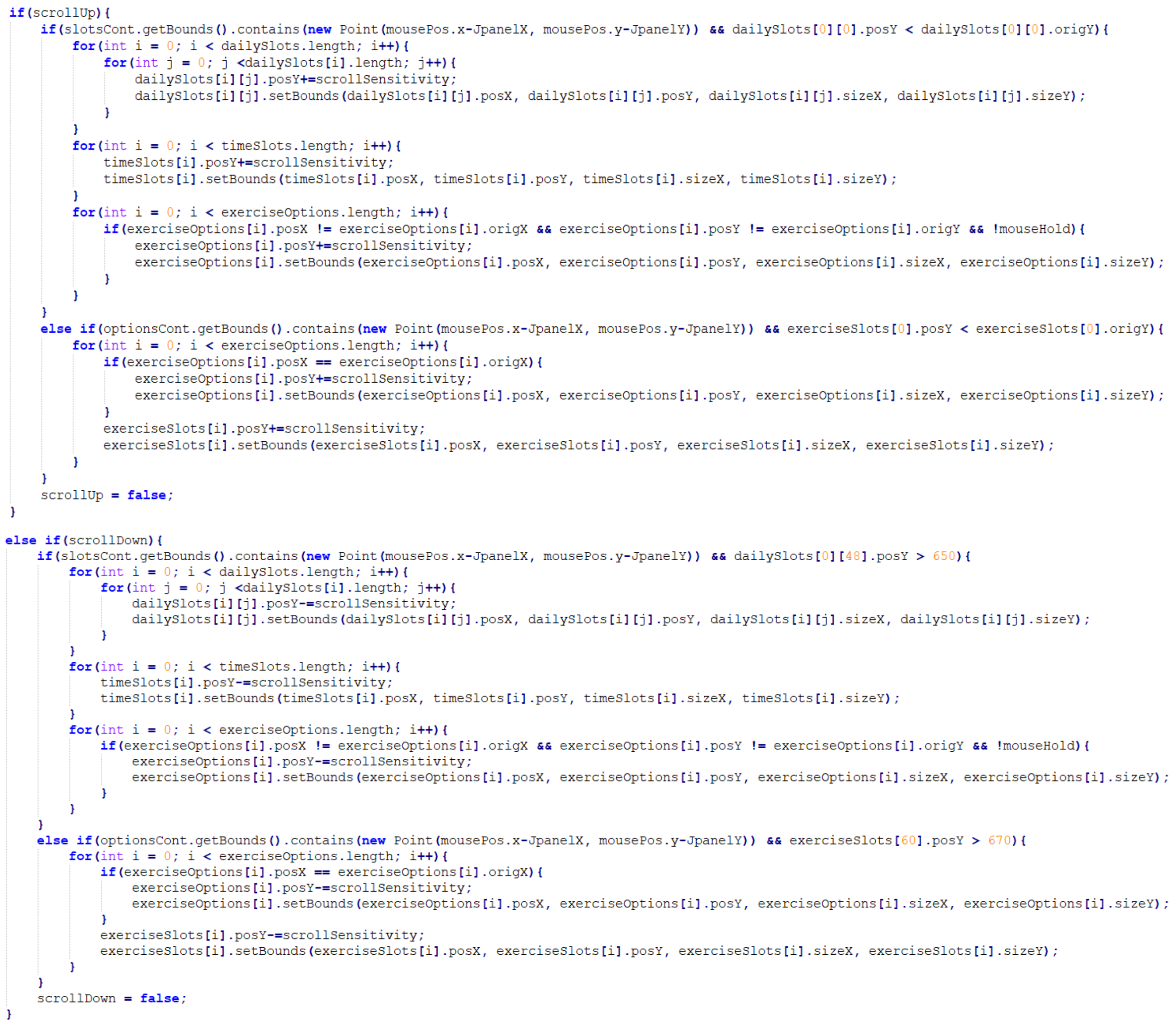
These overriding methods were used to program the drag and drop features of the schedule.

MouseWheelListener



```
public void mouseWheelMoved(MouseWheelEvent e) {
    if(e.getWheelRotation() < 0){
        scrollUp = true;
    }
    else if(e.getWheelRotation() > 0){
        scrollDown = true;
    }
}
```

MouseWheelListener was implemented to detect the scrolling direction of the mouse.



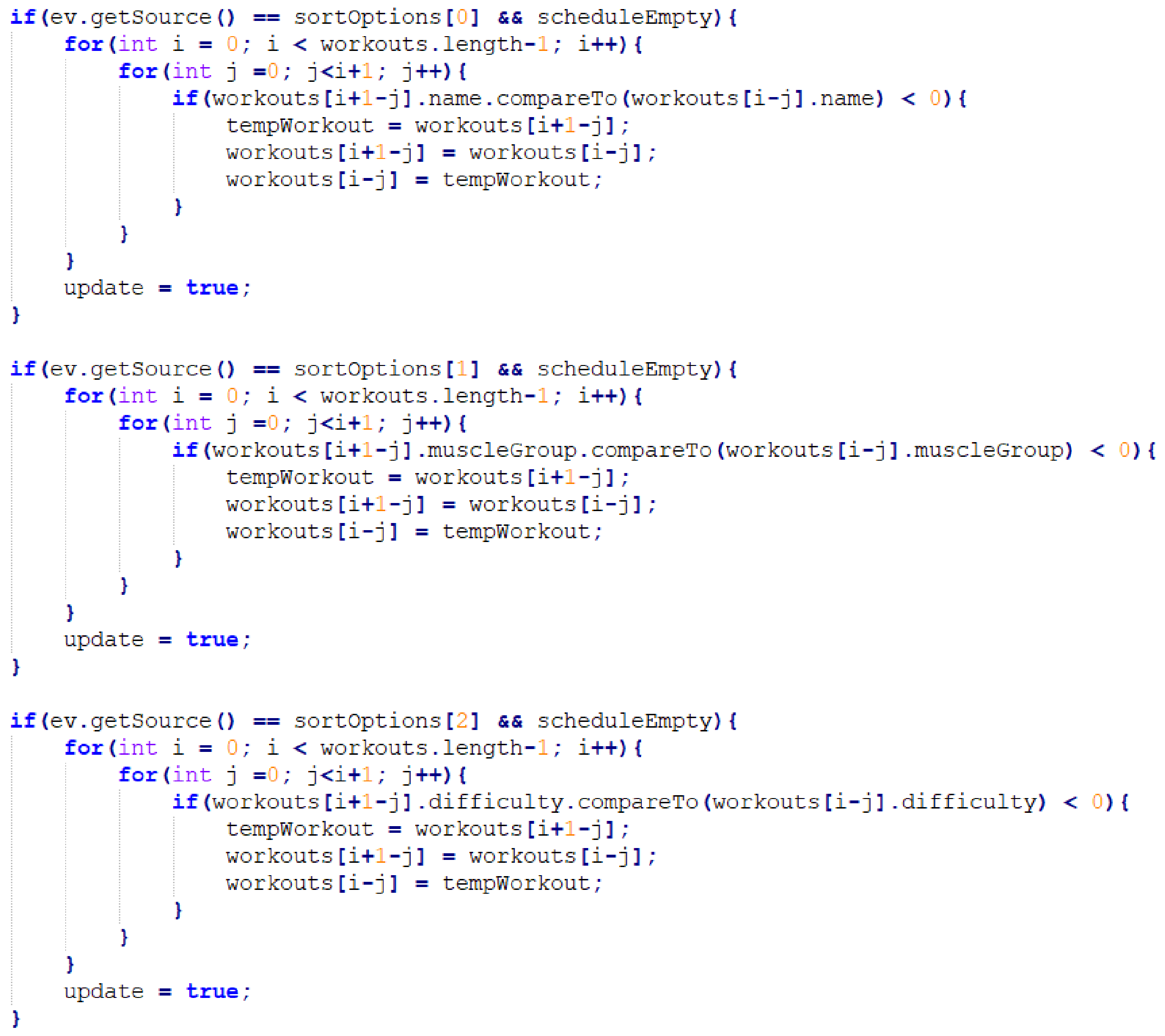
```
if(scrollUp){
    if(slotsCont.getBounds().contains(new Point(mousePos.x-JpanelX, mousePos.y-JpanelY)) && dailySlots[0][0].posY < dailySlots[0][0].origY){
        for(int i = 0; i < dailySlots.length; i++){
            for(int j = 0; j <dailySlots[i].length; j++){
                dailySlots[i][j].posY+=scrollSensitivity;
                dailySlots[i][j].setBounds(dailySlots[i][j].posX, dailySlots[i][j].posY, dailySlots[i][j].sizeX, dailySlots[i][j].sizeY);
            }
        }
        for(int i = 0; i < timeSlots.length; i++){
            timeSlots[i].posY+=scrollSensitivity;
            timeSlots[i].setBounds(timeSlots[i].posX, timeSlots[i].posY, timeSlots[i].sizeX, timeSlots[i].sizeY);
        }
        for(int i = 0; i < exerciseOptions.length; i++){
            if(exerciseOptions[i].posX != exerciseOptions[i].origX && exerciseOptions[i].posY != exerciseOptions[i].origY && !mouseHold){
                exerciseOptions[i].posY+=scrollSensitivity;
                exerciseOptions[i].setBounds(exerciseOptions[i].posX, exerciseOptions[i].posY, exerciseOptions[i].sizeX, exerciseOptions[i].sizeY);
            }
        }
    }
    else if(optionsCont.getBounds().contains(new Point(mousePos.x-JpanelX, mousePos.y-JpanelY)) && exerciseSlots[0].posY < exerciseSlots[0].origY){
        for(int i = 0; i < exerciseOptions.length; i++){
            if(exerciseOptions[i].posX == exerciseOptions[i].origX){
                exerciseOptions[i].posY+=scrollSensitivity;
                exerciseOptions[i].setBounds(exerciseOptions[i].posX, exerciseOptions[i].posY, exerciseOptions[i].sizeX, exerciseOptions[i].sizeY);
            }
            exerciseSlots[i].posY+=scrollSensitivity;
            exerciseSlots[i].setBounds(exerciseSlots[i].posX, exerciseSlots[i].posY, exerciseSlots[i].sizeX, exerciseSlots[i].sizeY);
        }
    }
    scrollUp = false;
}

else if(scrollDown){
    if(slotsCont.getBounds().contains(new Point(mousePos.x-JpanelX, mousePos.y-JpanelY)) && dailySlots[0][48].posY > 650){
        for(int i = 0; i < dailySlots.length; i++){
            for(int j = 0; j <dailySlots[i].length; j++){
                dailySlots[i][j].posY-=scrollSensitivity;
                dailySlots[i][j].setBounds(dailySlots[i][j].posX, dailySlots[i][j].posY, dailySlots[i][j].sizeX, dailySlots[i][j].sizeY);
            }
        }
        for(int i = 0; i < timeSlots.length; i++){
            timeSlots[i].posY-=scrollSensitivity;
            timeSlots[i].setBounds(timeSlots[i].posX, timeSlots[i].posY, timeSlots[i].sizeX, timeSlots[i].sizeY);
        }
        for(int i = 0; i < exerciseOptions.length; i++){
            if(exerciseOptions[i].posX != exerciseOptions[i].origX && exerciseOptions[i].posY != exerciseOptions[i].origY && !mouseHold){
                exerciseOptions[i].posY-=scrollSensitivity;
                exerciseOptions[i].setBounds(exerciseOptions[i].posX, exerciseOptions[i].posY, exerciseOptions[i].sizeX, exerciseOptions[i].sizeY);
            }
        }
    }
    else if(optionsCont.getBounds().contains(new Point(mousePos.x-JpanelX, mousePos.y-JpanelY)) && exerciseSlots[60].posY > 670){
        for(int i = 0; i < exerciseOptions.length; i++){
            if(exerciseOptions[i].posX == exerciseOptions[i].origX){
                exerciseOptions[i].posY-=scrollSensitivity;
                exerciseOptions[i].setBounds(exerciseOptions[i].posX, exerciseOptions[i].posY, exerciseOptions[i].sizeX, exerciseOptions[i].sizeY);
            }
            exerciseSlots[i].posY-=scrollSensitivity;
            exerciseSlots[i].setBounds(exerciseSlots[i].posX, exerciseSlots[i].posY, exerciseSlots[i].sizeX, exerciseSlots[i].sizeY);
        }
    }
    scrollDown = false;
}
```

The positions of all elements on the interface were then recalculated based on the scroll wheel input.

Searching and Sorting Algorithms

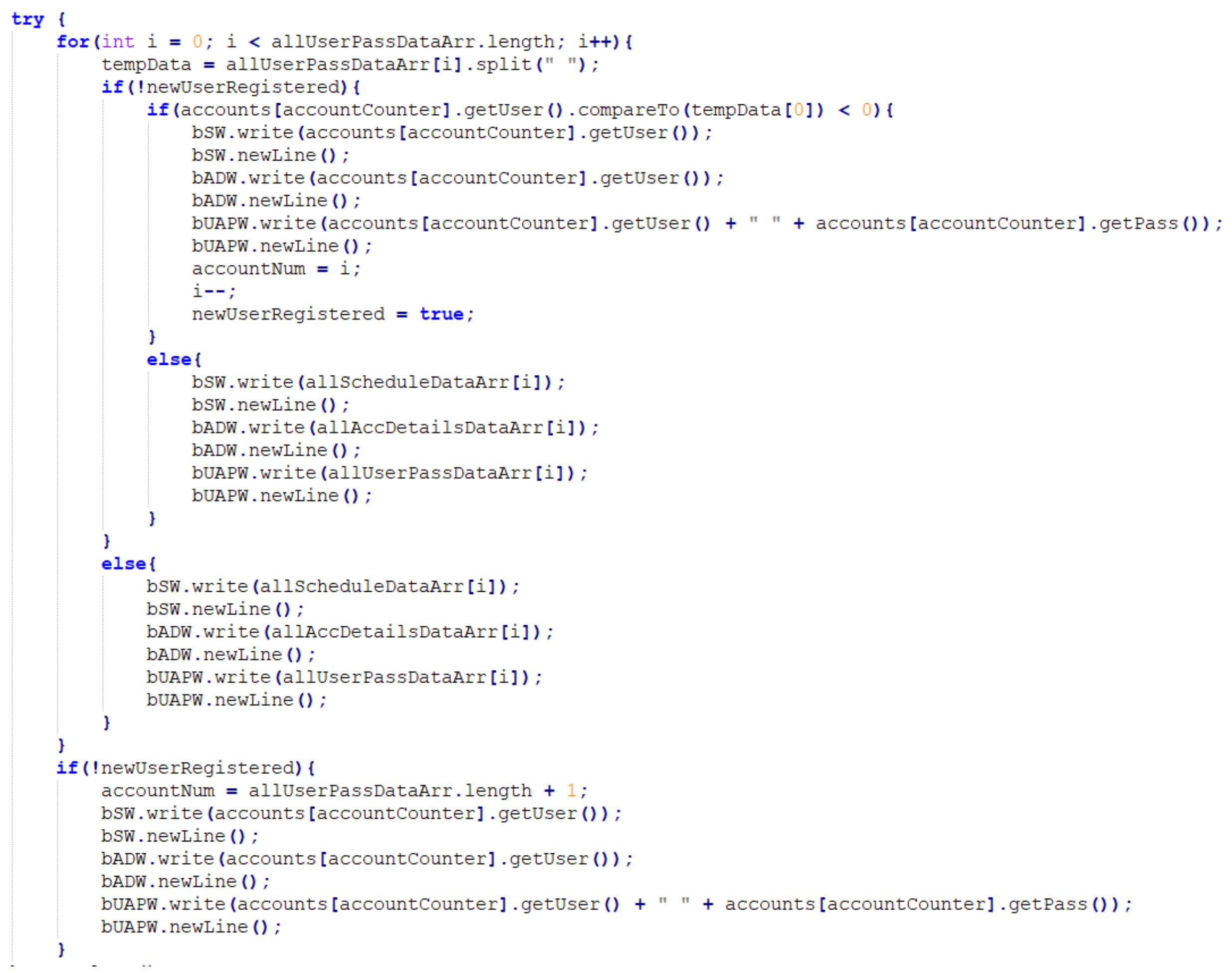
Insertion Sort

Insertion sort was used to sort account data as well as the list of exercises, as it tested to be very efficient with large data sets.



```
if(ev.getSource() == sortOptions[0] && scheduleEmpty){
    for(int i = 0; i < workouts.length-1; i++){
        for(int j =0; j<i+1; j++){
            if(workouts[i+1-j].name.compareTo(workouts[i-j].name) < 0){
                tempWorkout = workouts[i+1-j];
                workouts[i+1-j] = workouts[i-j];
                workouts[i-j] = tempWorkout;
            }
        }
    }
    update = true;
}

if(ev.getSource() == sortOptions[1] && scheduleEmpty){
    for(int i = 0; i < workouts.length-1; i++){
        for(int j =0; j<i+1; j++){
            if(workouts[i+1-j].muscleGroup.compareTo(workouts[i-j].muscleGroup) < 0){
                tempWorkout = workouts[i+1-j];
                workouts[i+1-j] = workouts[i-j];
                workouts[i-j] = tempWorkout;
            }
        }
    }
    update = true;
}

if(ev.getSource() == sortOptions[2] && scheduleEmpty){
    for(int i = 0; i < workouts.length-1; i++){
        for(int j =0; j<i+1; j++){
            if(workouts[i+1-j].difficulty.compareTo(workouts[i-j].difficulty) < 0){
                tempWorkout = workouts[i+1-j];
                workouts[i+1-j] = workouts[i-j];
                workouts[i-j] = tempWorkout;
            }
        }
    }
    update = true;
}
```

In my buttonChecker class, there are 3 buttons corresponding to 3 sorting options. The list of exercises can be sorted alphabetically, by muscle group, and by difficulty, all of which are done so via insertion sort.

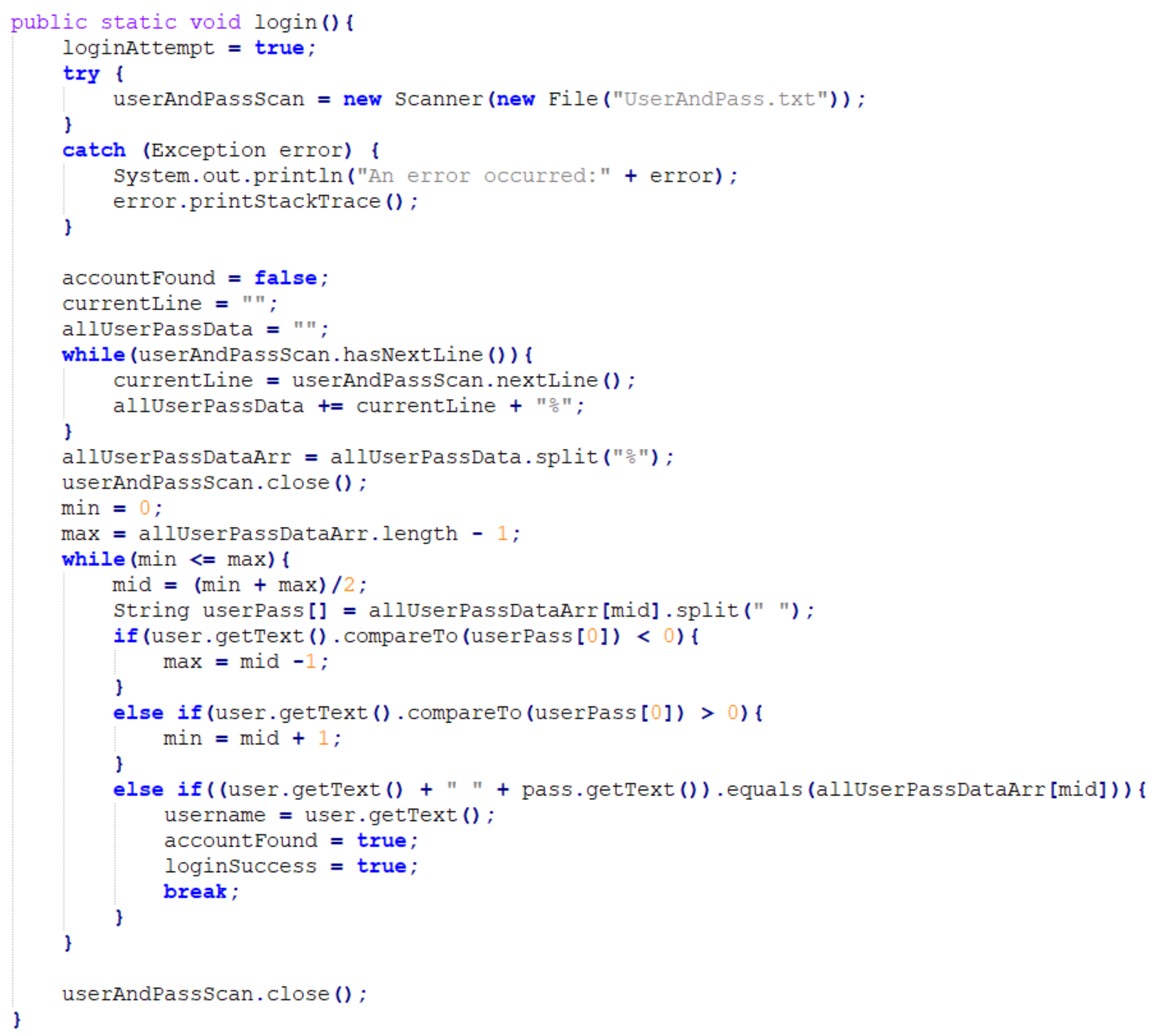


```
try {
    for(int i = 0; i < allUserPassDataArr.length; i++){
        tempData = allUserPassDataArr[i].split(" ");
        if(!newUserRegistered){
            if(accounts[accountCounter].getUser().compareTo(tempData[0]) < 0){
                bSW.write(accounts[accountCounter].getUser());
                bSW.newLine();
                bADW.write(accounts[accountCounter].getUser());
                bADW.newLine();
                bUAPW.write(accounts[accountCounter].getUser() + " " + accounts[accountCounter].getPass());
                bUAPW.newLine();
                accountNum = i;
                i--;
                newUserRegistered = true;
            }
            else{
                bSW.write(allScheduleDataArr[i]);
                bSW.newLine();
                bADW.write(allAccDetailsDataArr[i]);
                bADW.newLine();
                bUAPW.write(allUserPassDataArr[i]);
                bUAPW.newLine();
            }
        }
        else{
            bSW.write(allScheduleDataArr[i]);
            bSW.newLine();
            bADW.write(allAccDetailsDataArr[i]);
            bADW.newLine();
            bUAPW.write(allUserPassDataArr[i]);
            bUAPW.newLine();
        }
    }
    if(!newUserRegistered){
        accountNum = allUserPassDataArr.length + 1;
        bSW.write(accounts[accountCounter].getUser());
        bSW.newLine();
        bADW.write(accounts[accountCounter].getUser());
        bADW.newLine();
        bUAPW.write(accounts[accountCounter].getUser() + " " + accounts[accountCounter].getPass());
        bUAPW.newLine();
    }
```

Account data is also sorted in a similar manner, however since all user data is stored in a database, the data only needs to be sorted once everytime a new account is registered.

Binary search

Binary search is an efficient searching algorithm that searches for targets by splitting the list into halves during every iteration.



```
public static void login(){
    loginAttempt = true;
    try {
        userAndPassScan = new Scanner(new File("UserAndPass.txt"));
    }
    catch (Exception error) {
        System.out.println("An error occurred:" + error);
        error.printStackTrace();
    }

    accountFound = false;
    currentLine = "";
    allUserPassData = "";
    while(userAndPassScan.hasNextLine()){
        currentLine = userAndPassScan.nextLine();
        allUserPassData += currentLine + "%";
    }
    allUserPassDataArr = allUserPassData.split("%");
    userAndPassScan.close();
    min = 0;
    max = allUserPassDataArr.length - 1;
    while(min <= max){
        mid = (min + max)/2;
        String userPass[] = allUserPassDataArr[mid].split(" ");
        if(user.getText().compareTo(userPass[0]) < 0){
            max = mid -1;
        }
        else if(user.getText().compareTo(userPass[0]) > 0){
            min = mid + 1;
        }
        else if((user.getText() + " " + pass.getText()).equals(allUserPassDataArr[mid])){
            username = user.getText();
            accountFound = true;
            loginSuccess = true;
            break;
        }
    }

    userAndPassScan.close();
}
```

This algorithm was implemented in the login method of the Menu class to efficiently determine whether a username or password inputted by the user is correct. This is most effective because the accounts are already sorted upon registration, and are not a random list.

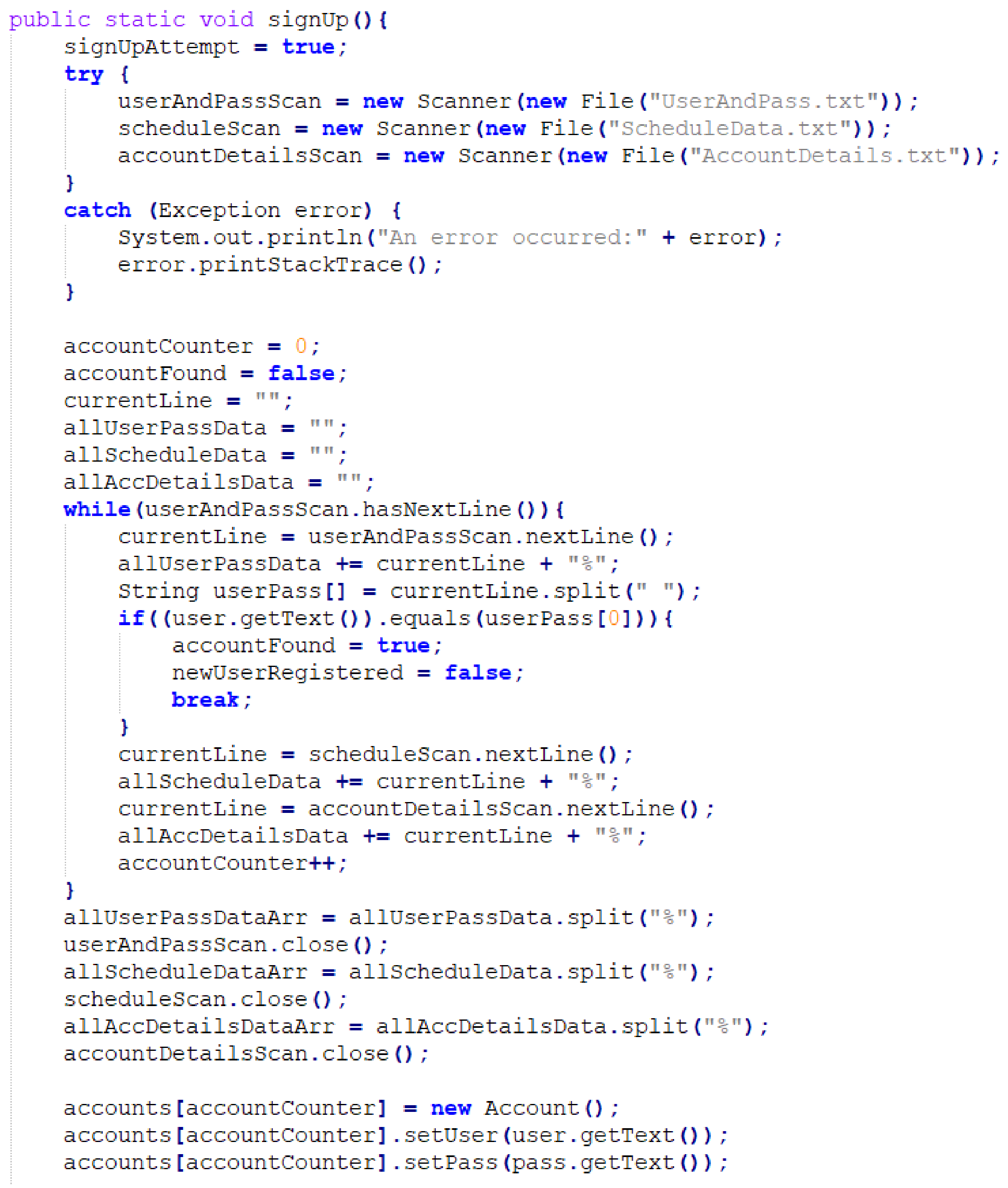
Libraries

Imported libraries used in this program include: IO, Util, Swing, AWT, and Time.

Database

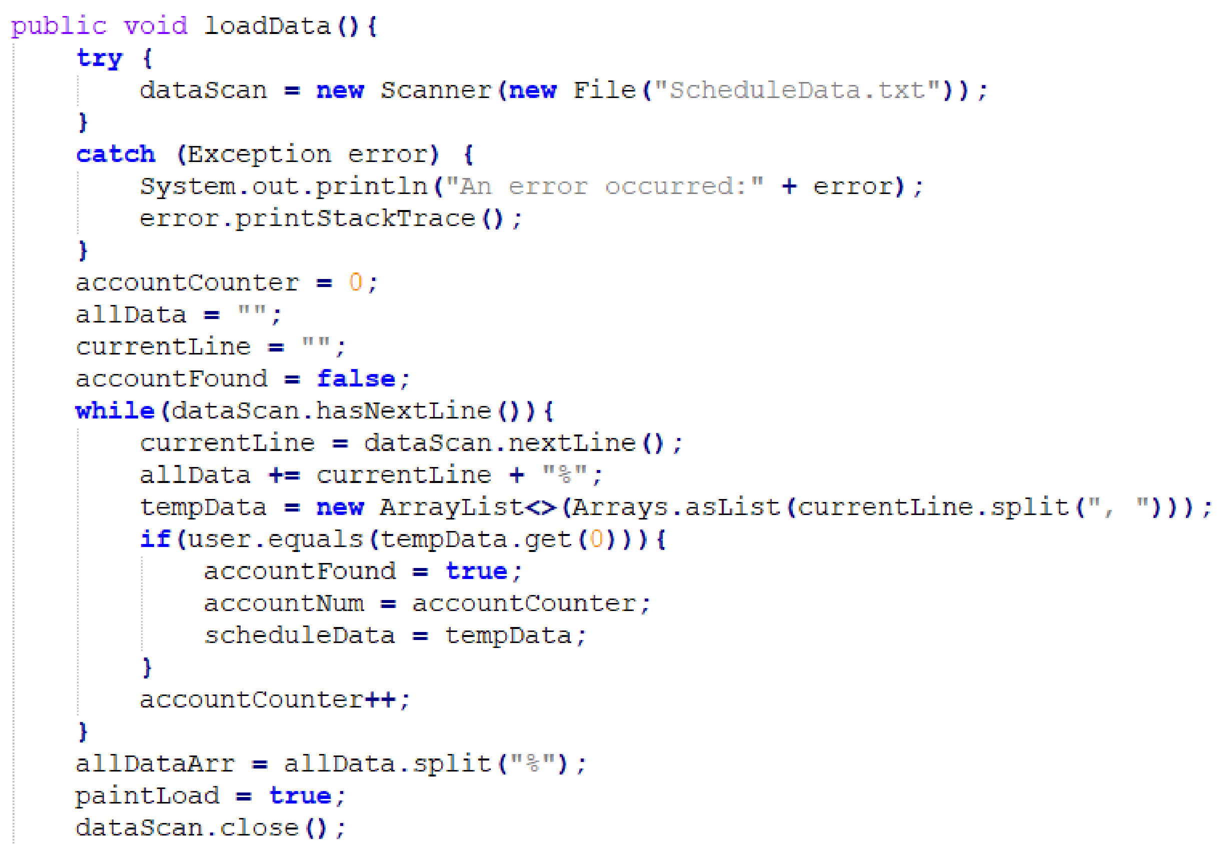
Retrieving data

Account data and schedule data is retrieved from text files using scanners.



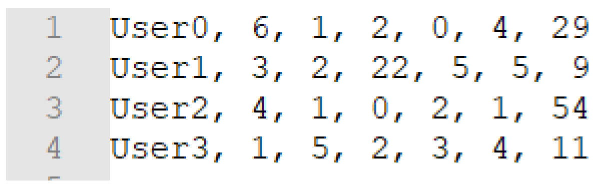
```java
public static void signUp(){
    signUpAttempt = true;
    try {
        userAndPassScan = new Scanner(new File("UserAndPass.txt"));
        scheduleScan = new Scanner(new File("ScheduleData.txt"));
        accountDetailsScan = new Scanner(new File("AccountDetails.txt"));
    }
    catch (Exception error) {
        System.out.println("An error occurred:" + error);
        error.printStackTrace();
    }

    accountCounter = 0;
    accountFound = false;
    currentLine = "";
    allUserPassData = "";
    allScheduleData = "";
    allAccDetailsData = "";
    while(userAndPassScan.hasNextLine()){
        currentLine = userAndPassScan.nextLine();
        allUserPassData += currentLine + "%";
        String userPass[] = currentLine.split(" ");
        if((user.getText()).equals(userPass[0])){
            accountFound = true;
            newUserRegistered = false;
            break;
        }
        currentLine = scheduleScan.nextLine();
        allScheduleData += currentLine + "%";
        currentLine = accountDetailsScan.nextLine();
        allAccDetailsData += currentLine + "%";
        accountCounter++;
    }
    allUserPassDataArr = allUserPassData.split("%");
    userAndPassScan.close();
    allScheduleDataArr = allScheduleData.split("%");
    scheduleScan.close();
    allAccDetailsDataArr = allAccDetailsData.split("%");
    accountDetailsScan.close();

    accounts[accountCounter] = new Account();
    accounts[accountCounter].setUser(user.getText());
    accounts[accountCounter].setPass(pass.getText());
```

For example, the usernames and passwords are retrieved from the text files by indexing each line of the file into an array, where each line includes the username and password for a specific account. Then, each of those lines are split and indexed in an array for use.



```java
public void loadData(){
    try {
        dataScan = new Scanner(new File("ScheduleData.txt"));
    }
    catch (Exception error) {
        System.out.println("An error occurred:" + error);
        error.printStackTrace();
    }
    accountCounter = 0;
    allData = "";
    currentLine = "";
    accountFound = false;
    while(dataScan.hasNextLine()){
        currentLine = dataScan.nextLine();
        allData += currentLine + "%";
        tempData = new ArrayList<>(Arrays.asList(currentLine.split(", ")));
        if(user.equals(tempData.get(0))){
            accountFound = true;
            accountNum = accountCounter;
            scheduleData = tempData;
        }
        accountCounter++;
    }
    allDataArr = allData.split("%");
    paintLoad = true;
    dataScan.close();
```

The same method is used to load data for the schedule.

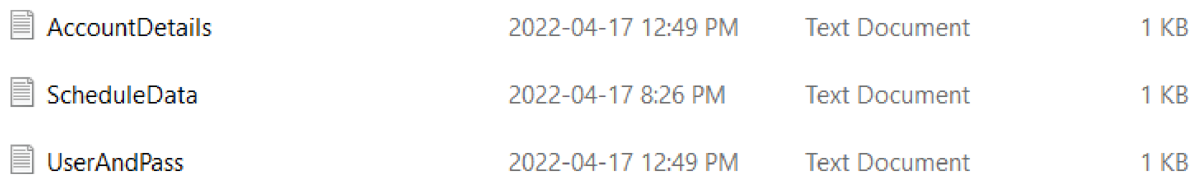


```
User0, 6, 1, 2, 0, 4, 29
User1, 3, 2, 22, 5, 5, 9
User2, 4, 1, 0, 2, 1, 54
User3, 1, 5, 2, 3, 4, 11
```

Each line in the text file contains schedule data for a specific user. It begins with their username, followed by a sequence of numbers. The first two numbers correspond to the position of a time slot that is taken in the 2D time slot array, and the third number indicates which exercise from the list of workouts is selected at that specific time slot.

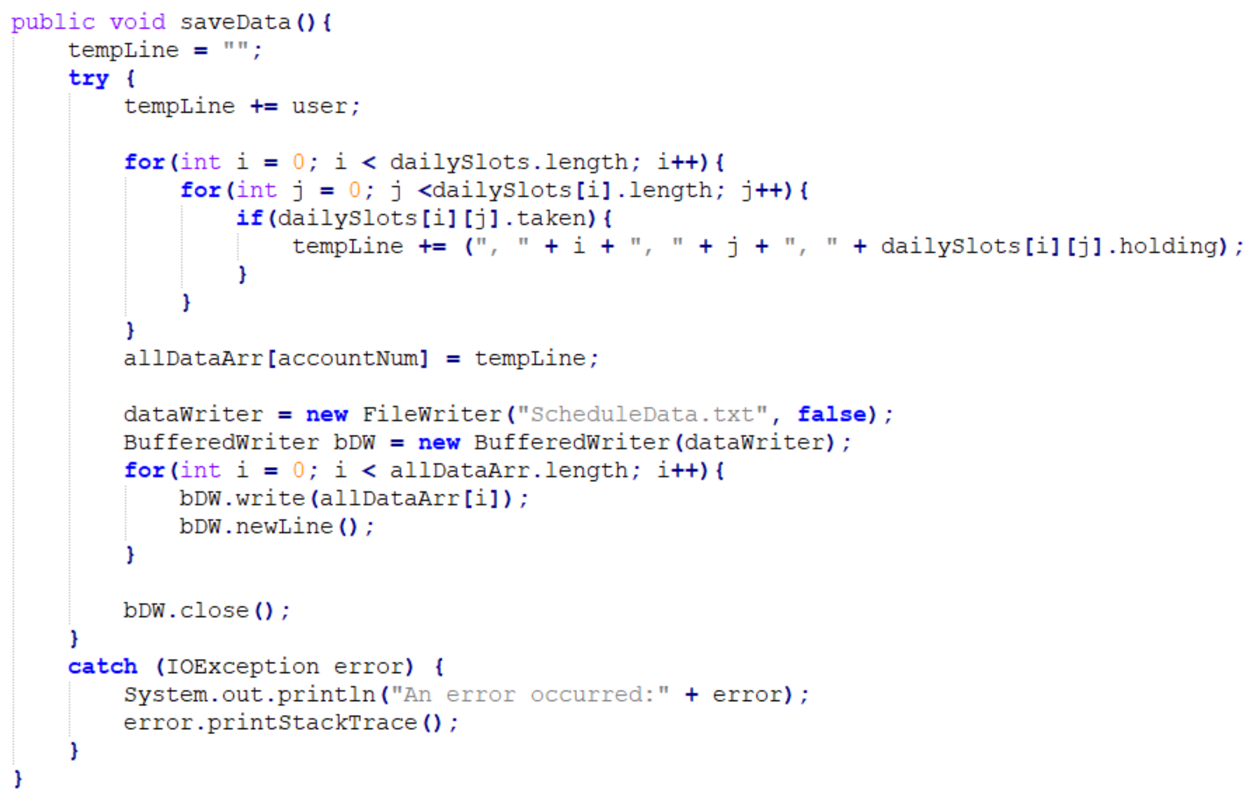
Saving Data

Account data and schedule data are stored in text files using FileWriter and BufferedWriter.



AccountDetails	2022-04-17 12:49 PM	Text Document	1 KB
ScheduleData	2022-04-17 8:26 PM	Text Document	1 KB
UserAndPass	2022-04-17 12:49 PM	Text Document	1 KB

Three files are used: AccountDetails to store user information (height, weight, age), UserAndPass to store usernames and passwords, and ScheduleData to store data in the schedule.



```
public void saveData(){
    tempLine = "";
    try {
        tempLine += user;

        for(int i = 0; i < dailySlots.length; i++){
            for(int j = 0; j <dailySlots[i].length; j++){
                if(dailySlots[i][j].taken){
                    tempLine += (", " + i + ", " + j + ", " + dailySlots[i][j].holding);
                }
            }
        }
        allDataArr[accountNum] = tempLine;

        dataWriter = new FileWriter("ScheduleData.txt", false);
        BufferedWriter bDW = new BufferedWriter(dataWriter);
        for(int i = 0; i < allDataArr.length; i++){
            bDW.write(allDataArr[i]);
            bDW.newLine();
        }

        bDW.close();
    }
    catch (IOException error) {
        System.out.println("An error occurred:" + error);
        error.printStackTrace();
    }
}
```

Since the schedule and user data are saved in lines, all the information from the file must be read and indexed in an array during the retrieval process before it is ready to be saved. This is because it is otherwise very difficult to access and edit a specific line in a text file. Thus, when changes are made and the data is ready to be saved, all previous data for other accounts are rewritten to the file, and data regarding the current in-use account is overwritten with the new changes.

Word Count: 1036