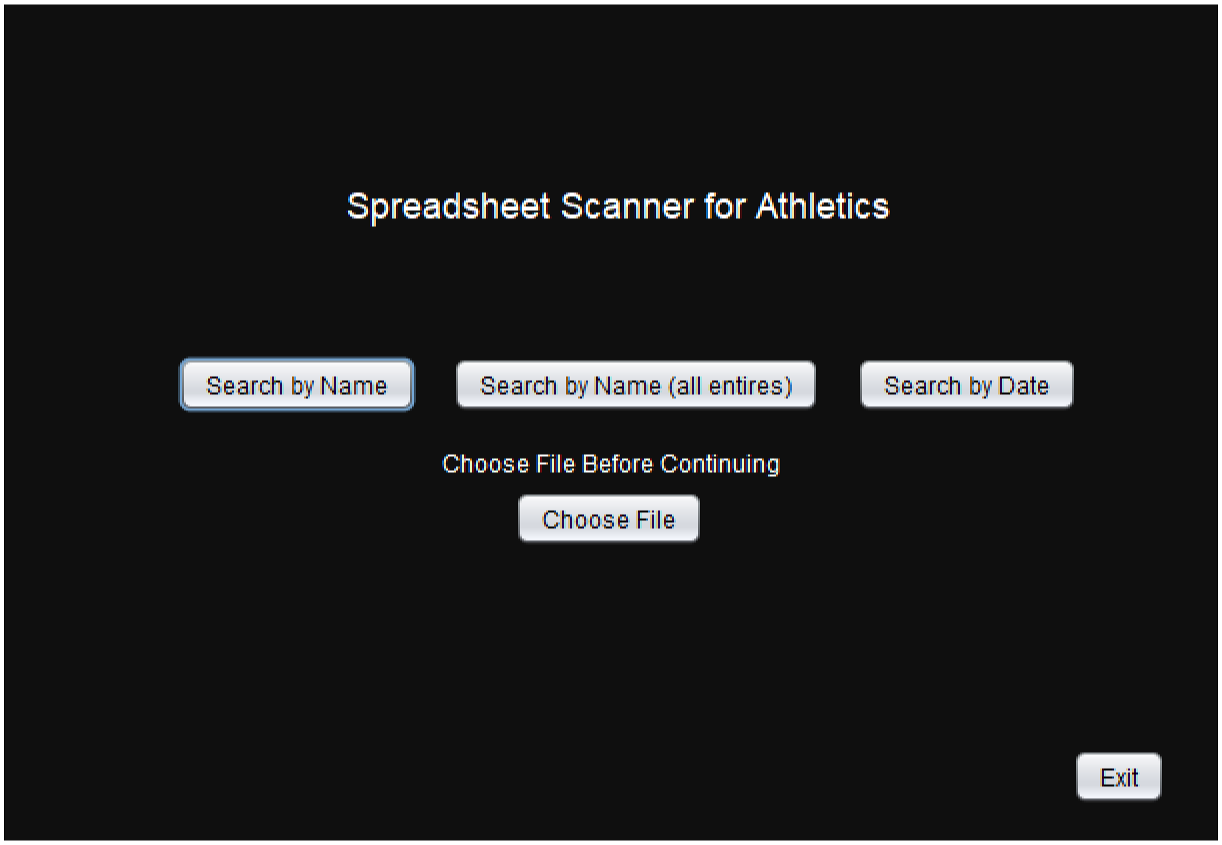
Criterion C: Development

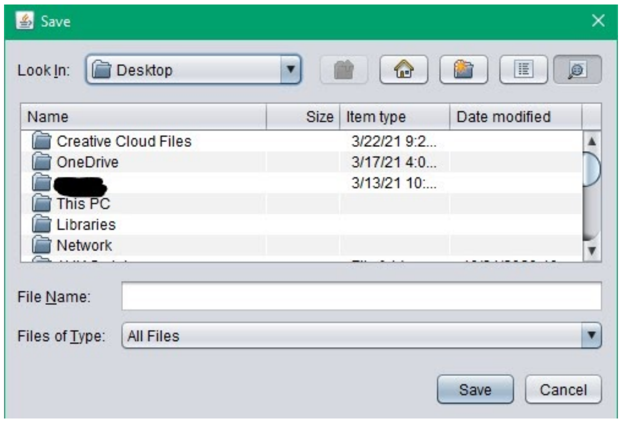
The Athletics spreadsheet scanner is a java program made for Mr. xx to scan a spreadsheet of athletics attendance information so that he can more easily find and access the information he needs.

Program structure:

Landing page GUI (Graphical user interface):


Spreadsheet Scanner for Athletics
Search by Name
Search by Name (all entires)
Search by Date
Choose File Before Continuing
Choose File
Exit

Choose File:



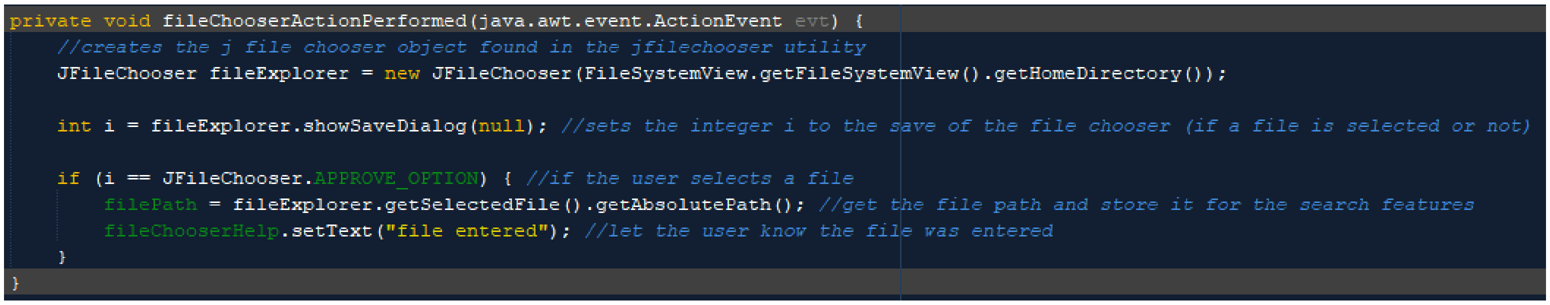
Name	Size	Item type	Date modified
Creative Cloud Files		3/22/21 9:2...	
OneDrive		3/17/21 4:0...	
[redacted]		3/13/21 10:...	
This PC			
Libraries			
Network			

File Name:

Files of Type: All Files

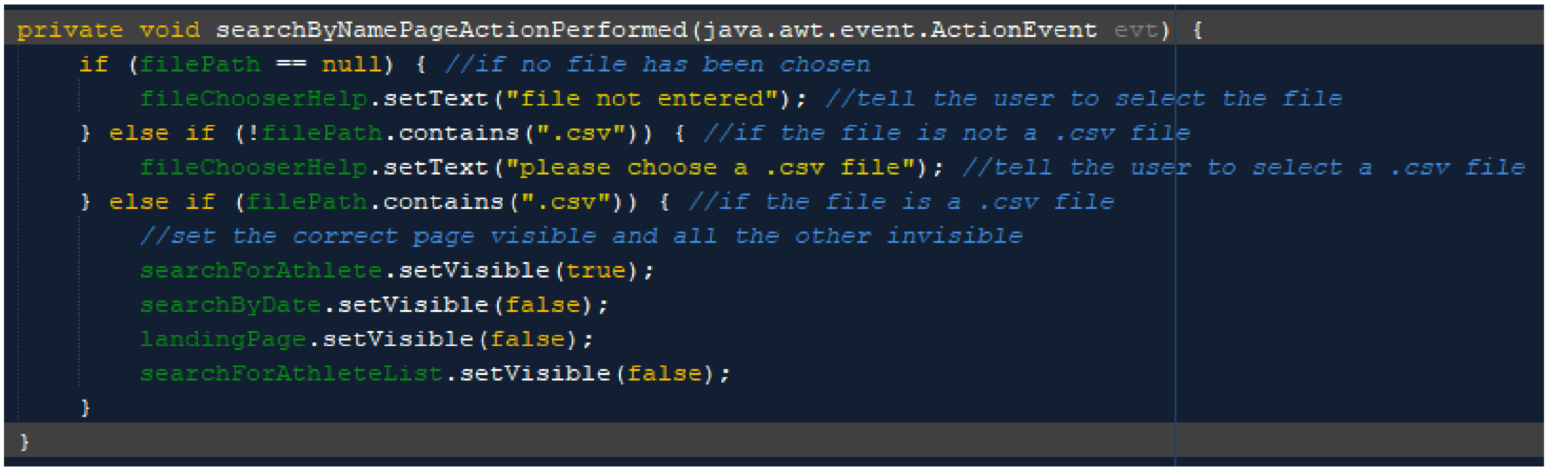
This entire GUI was coded within the Java Swing editor in Netbeans, which I chose because of its user friendly creation experience and because this is the way I learned how to create a GUI in IB CompSci SL1. This editor allowed me to easily drag and drop buttons, text fields, panels, and more into the panel of the GUI I was working on while still allowing me to access and change the code. In addition, it has a detailed editing feature that allows for precise adjustments of the way an element of the GUI appears or functions without having to look up the syntax or worry about errors. Finally, the swing editor allowed me to access it's utilities very easily, which allowed me to make the file explorer I needed with relative ease.

The first feature that is needed before the user is allowed to advance is the Choose File button. This button allows the user to open a file explorer GUI (using the JFileChooser and filechooser.FileSystemView swing utilities) that they can select the file they want to scan from. This is the code I used to take advantage of the utilities:

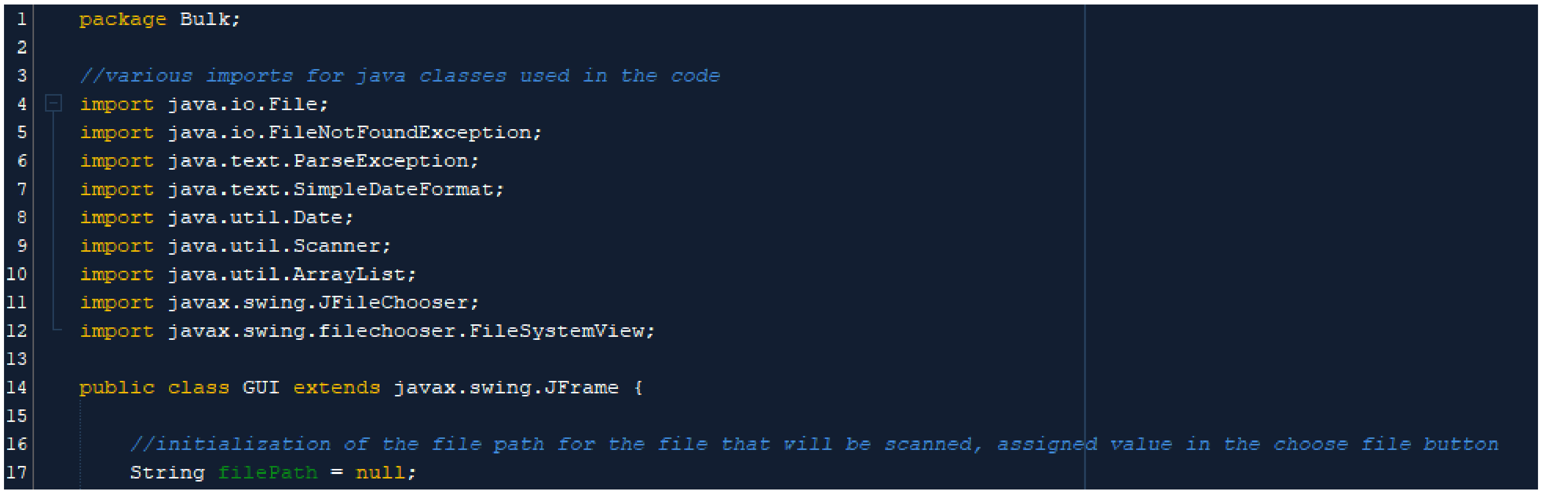


```java
private void fileChooserActionPerformed(java.awt.event.ActionEvent evt) {
    //creates the j file chooser object found in the jfilechooser utility
    JFileChooser fileExplorer = new JFileChooser(FileSystemView.getFileSystemView().getHomeDirectory());

    int i = fileExplorer.showSaveDialog(null); //sets the integer i to the save of the file chooser (if a file is selected or not)

    if (i == JFileChooser.APPROVE_OPTION) { //if the user selects a file
        filePath = fileExplorer.getSelectedFile().getAbsolutePath(); //get the file path and store it for the search features
        fileChooserHelp.setText("file entered"); //let the user know the file was entered
    }
}
```

All of this code is inside the fileChooser button's action performed, which means that when the fileChooser button is pressed, this code will be run. In addition, the filePath variable is used in every other feature so that the file chosen can be stored and doesn't have to be re-selected every time the user enters a new window. Also, to prevent the user from trying to use a feature without having selected a file each button checks to see if a file has been selected, for example the code for the Search by name button looks like this:



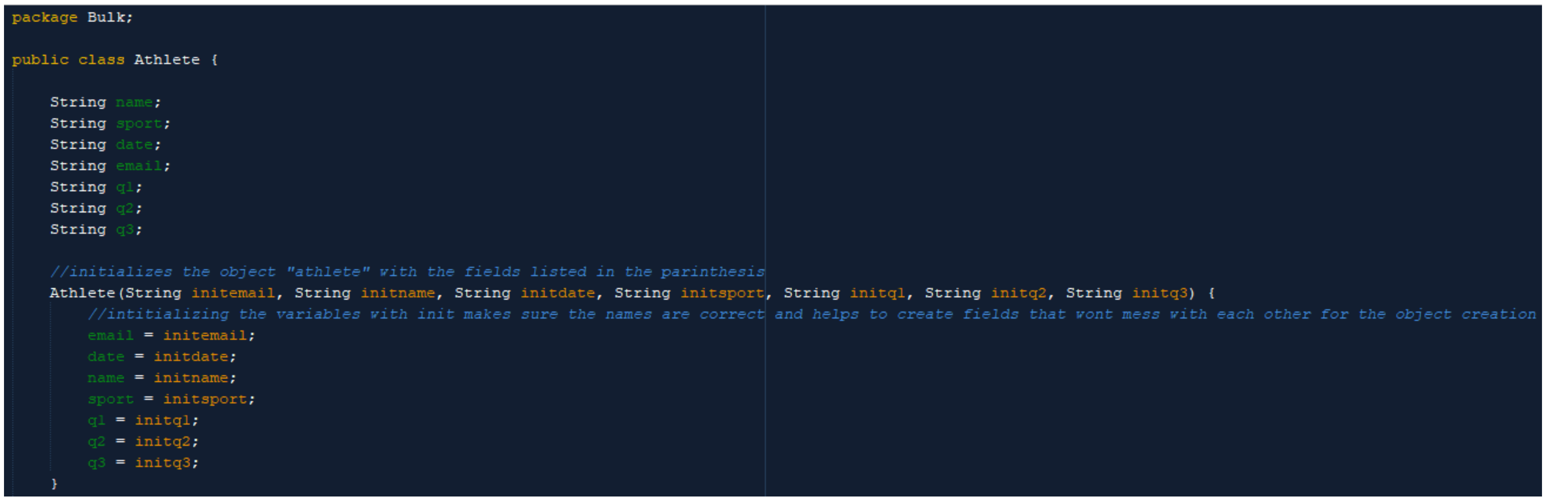
```java
private void searchByNamePageActionPerformed(java.awt.event.ActionEvent evt) {
    if (filePath == null) { //if no file has been chosen
        fileChooserHelp.setText("file not entered"); //tell the user to select the file
    } else if (!filePath.contains(".csv")) { //if the file is not a .csv file
        fileChooserHelp.setText("please choose a .csv file"); //tell the user to select a .csv file
    } else if (filePath.contains(".csv")) { //if the file is a .csv file
        //set the correct page visible and all the other invisible
        searchForAthlete.setVisible(true);
        searchByDate.setVisible(false);
        landingPage.setVisible(false);
        searchForAthleteList.setVisible(false);
    }
}
```



```java
package Bulk;

//various imports for java classes used in the code
import java.io.File;
import java.io.FileNotFoundException;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Scanner;
import java.util.ArrayList;
import javax.swing.JFileChooser;
import javax.swing.filechooser.FileSystemView;

public class GUI extends javax.swing.JFrame {

    //initialization of the file path for the file that will be scanned, assigned value in the choose file button
    String filePath = null;
```

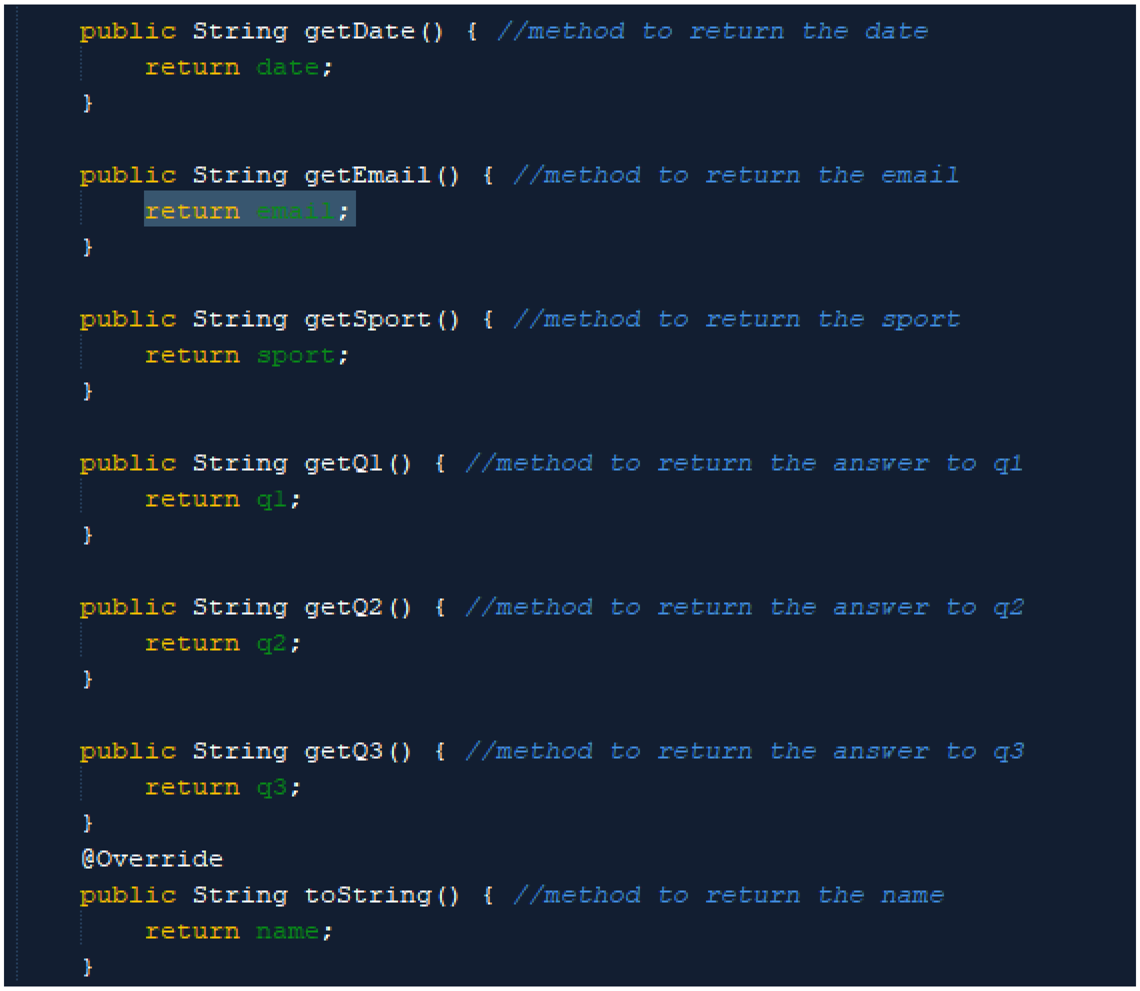
After the user has selected a file, they can choose from one of three features to continue with, one of which is the Search By Athlete feature. The code for this feature was the base code I used to make the other features, which was possible because I set it up in a modular way. The full code used can be seen in the source code appendix.

To start, I set up the athlete object in a separate class called “Athlete.java.” This creation of the object looks like this:



```java
package Bulk;

public class Athlete {

    String name;
    String sport;
    String date;
    String email;
    String q1;
    String q2;
    String q3;

    //initializes the object "athlete" with the fields listed in the parinthesis
    Athlete(String initemail, String initname, String initdate, String initsport, String initq1, String initq2, String initq3) {
        //intitializing the variables with init makes sure the names are correct and helps to create fields that wont mess with each other for the object creation
        email = initemail;
        date = initdate;
        name = initname;
        sport = initsport;
        q1 = initq1;
        q2 = initq2;
        q3 = initq3;
    }
```

<u>The object’s return methods:</u>



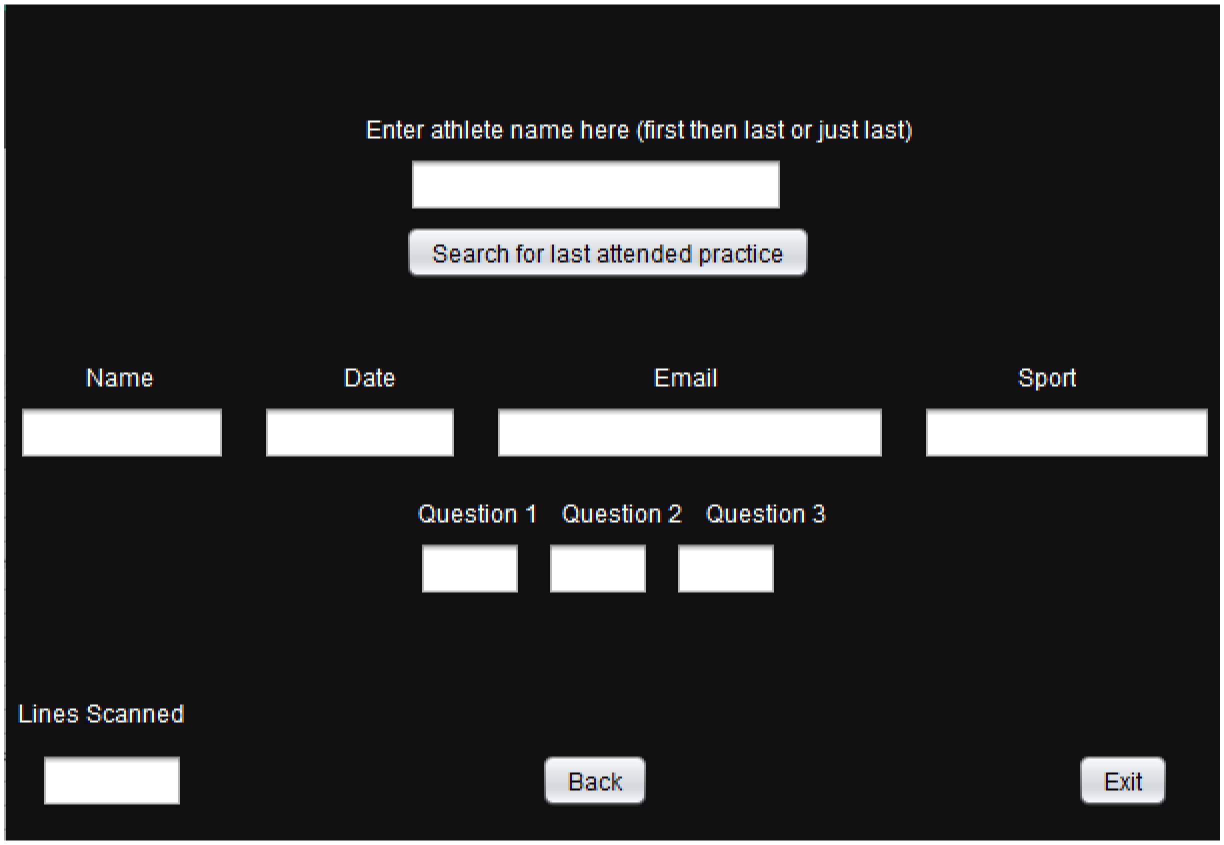
```
public String getDate() { //method to return the date
    return date;
}

public String getEmail() { //method to return the email
    return email;
}

public String getSport() { //method to return the sport
    return sport;
}

public String getQ1() { //method to return the answer to q1
    return q1;
}

public String getQ2() { //method to return the answer to q2
    return q2;
}

public String getQ3() { //method to return the answer to q3
    return q3;
}
@Override
public String toString() { //method to return the name
    return name;
}
```

As can be seen in the comment where the variables of the object are defined, init is used in front of the name of each variable as to avoid issues where the initial variables defined above the object would interfere with the object's variables. This object allows for all of the fields in the file to be inputted into it, creating a virtual version of an actual athlete and their information. The reason why this is necessary is because it makes storage of the data extremely easy and it allows for a lot of modularity. The ability to store the object in an arraylist after putting the values in it allows for my client to have an extremely large amount of data stored and none of it will be lost. This athlete object is used in all three features of the project, as can be seen below in the search and sort functions.

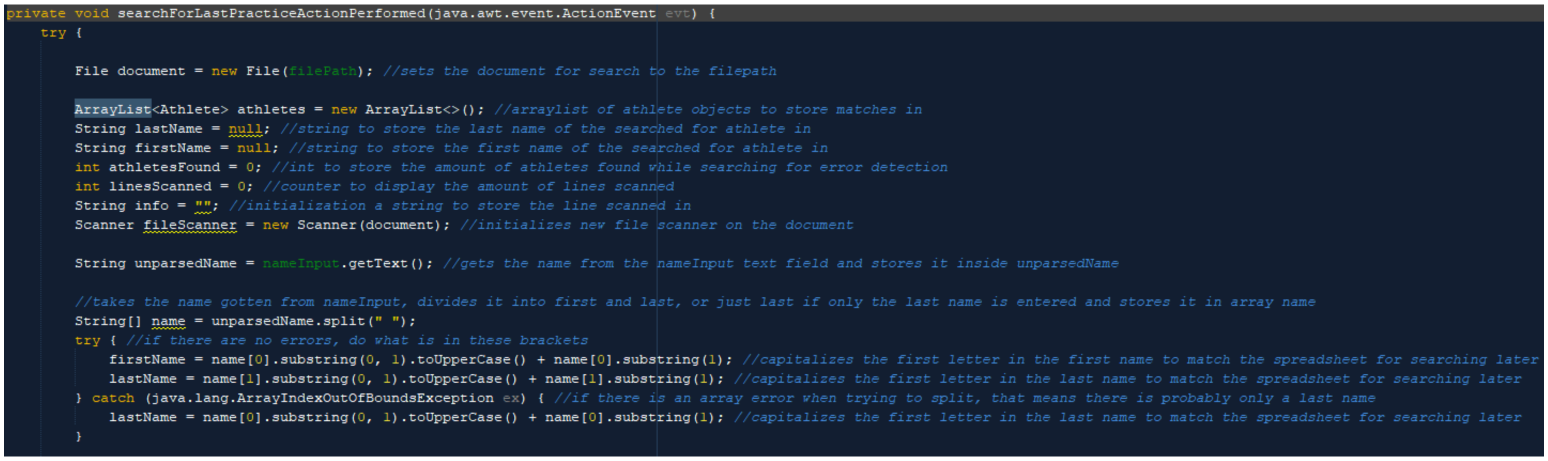
Because the code for the search by name feature is used in the other two features, I will go over individual parts of it and then when the other two features are discussed, if I do not mention a specific part in the other features it is because they are exactly the same.

Search by name

GUI:


Enter athlete name here (first then last or just last)
Search for last attended practice
Name
Date
Email
Sport
Question 1
Question 2
Question 3
Lines Scanned
Back
Exit

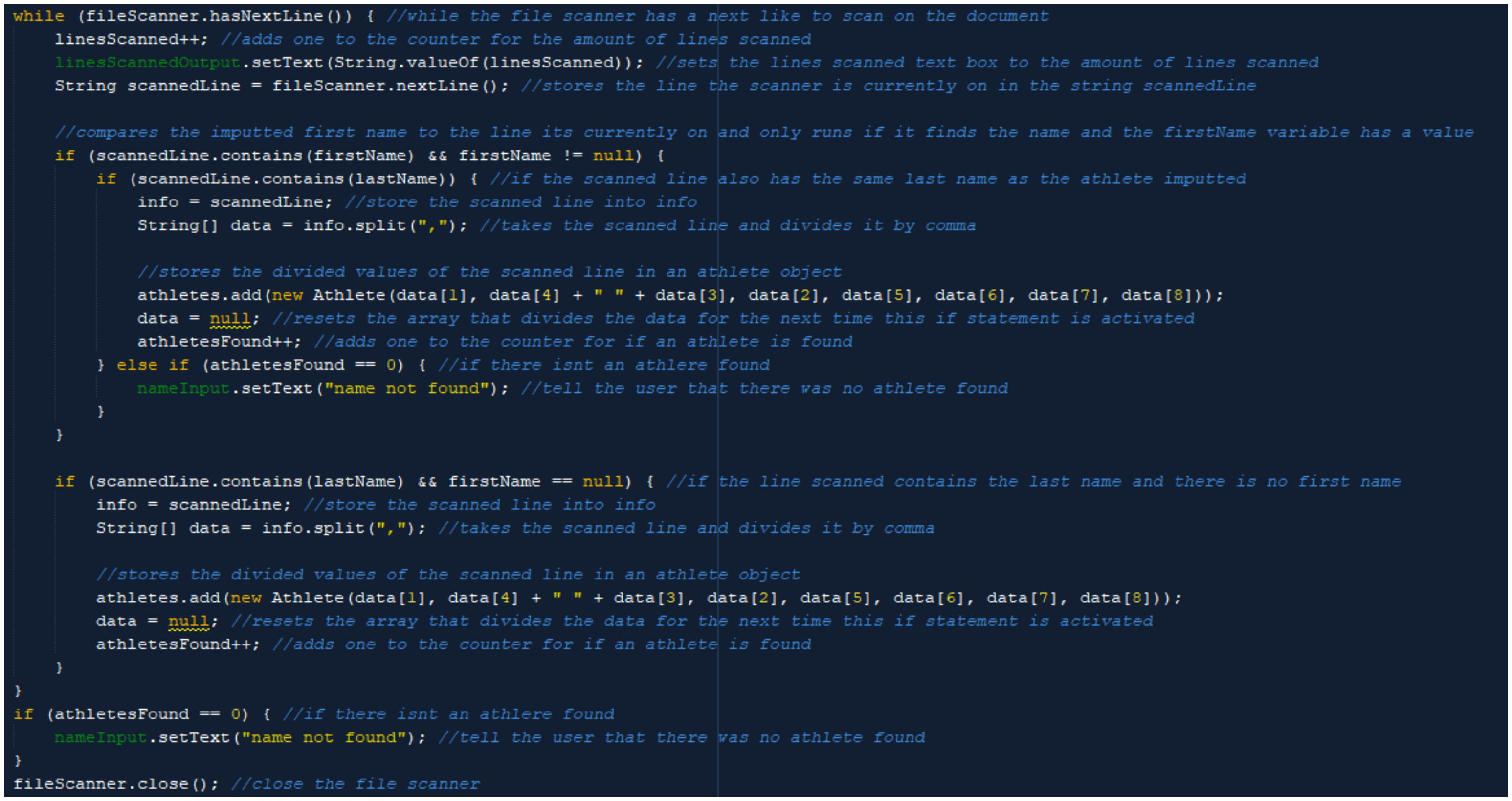
Initial setup of variables:



```java
private void searchForLastPracticeActionPerformed(java.awt.event.ActionEvent evt) {
    try {

        File document = new File(filePath); //sets the document for search to the filepath

        ArrayList<Athlete> athletes = new ArrayList<>(); //arraylist of athlete objects to store matches in
        String lastName = null; //string to store the last name of the searched for athlete in
        String firstName = null; //string to store the first name of the searched for athlete in
        int athletesFound = 0; //int to store the amount of athletes found while searching for error detection
        int linesScanned = 0; //counter to display the amount of lines scanned
        String info = ""; //initialization a string to store the line scanned in
        Scanner fileScanner = new Scanner(document); //initializes new file scanner on the document

        String unparsedName = nameInput.getText(); //gets the name from the nameInput text field and stores it inside unparsedName

        //takes the name gotten from nameInput, divides it into first and last, or just last if only the last name is entered and stores it in array name
        String[] name = unparsedName.split(" ");
        try { //if there are no errors, do what is in these brackets
            firstName = name[0].substring(0, 1).toUpperCase() + name[0].substring(1); //capitalizes the first letter in the first name to match the spreadsheet for searching later
            lastName = name[1].substring(0, 1).toUpperCase() + name[1].substring(1); //capitalizes the first letter in the last name to match the spreadsheet for searching later
        } catch (java.lang.ArrayIndexOutOfBoundsException ex) { //if there is an array error when trying to split, that means there is probably only a last name
            lastName = name[0].substring(0, 1).toUpperCase() + name[0].substring(1); //capitalizes the first letter in the last name to match the spreadsheet for searching later
        }
```

When I delivered the prototype to my client, I was presented with a feature he wanted me to add, the ability to search by first and last name and by just last name. My solution to this problem was to have the input from the user divided into an array using the .split() method and catch if there was an array out of bounds error. This was very effective because my later code catches if there is no match found or not, so this allows for the client to enter either format and still get an athlete returned. My later sorting code made this very easy to implement, all I needed to do was add another search algorithm for if just the last name was inputted.

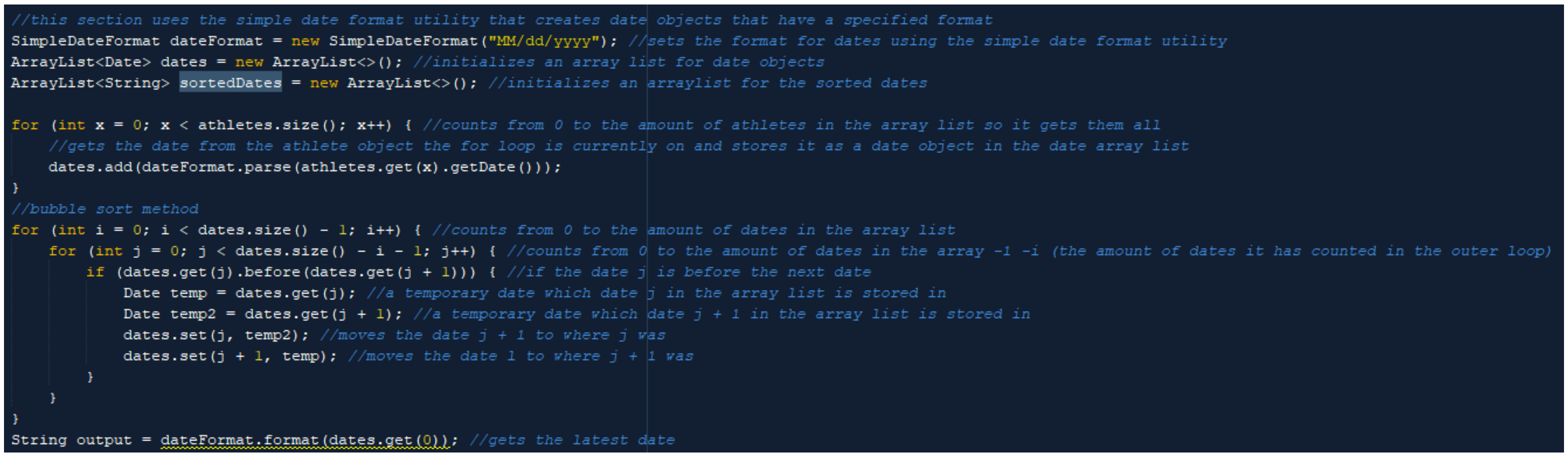
Search algorithm:



```
while (fileScanner.hasNextLine()) { //while the file scanner has a next like to scan on the document
    linesScanned++; //adds one to the counter for the amount of lines scanned
    linesScannedOutput.setText(String.valueOf(linesScanned)); //sets the lines scanned text box to the amount of lines scanned
    String scannedLine = fileScanner.nextLine(); //stores the line the scanner is currently on in the string scannedLine

    //compares the imputted first name to the line its currently on and only runs if it finds the name and the firstName variable has a value
    if (scannedLine.contains(firstName) && firstName != null) {
        if (scannedLine.contains(lastName)) { //if the scanned line also has the same last name as the athlete imputted
            info = scannedLine; //store the scanned line into info
            String[] data = info.split(","); //takes the scanned line and divides it by comma

            //stores the divided values of the scanned line in an athlete object
            athletes.add(new Athlete(data[1], data[4] + " " + data[3], data[2], data[5], data[6], data[7], data[8]));
            data = null; //resets the array that divides the data for the next time this if statement is activated
            athletesFound++; //adds one to the counter for if an athlete is found
        } else if (athletesFound == 0) { //if there isnt an athlere found
            nameInput.setText("name not found"); //tell the user that there was no athlete found
        }
    }

    if (scannedLine.contains(lastName) && firstName == null) { //if the line scanned contains the last name and there is no first name
        info = scannedLine; //store the scanned line into info
        String[] data = info.split(","); //takes the scanned line and divides it by comma

        //stores the divided values of the scanned line in an athlete object
        athletes.add(new Athlete(data[1], data[4] + " " + data[3], data[2], data[5], data[6], data[7], data[8]));
        data = null; //resets the array that divides the data for the next time this if statement is activated
        athletesFound++; //adds one to the counter for if an athlete is found
    }
}
if (athletesFound == 0) { //if there isnt an athlere found
    nameInput.setText("name not found"); //tell the user that there was no athlete found
}
fileScanner.close(); //close the file scanner
```

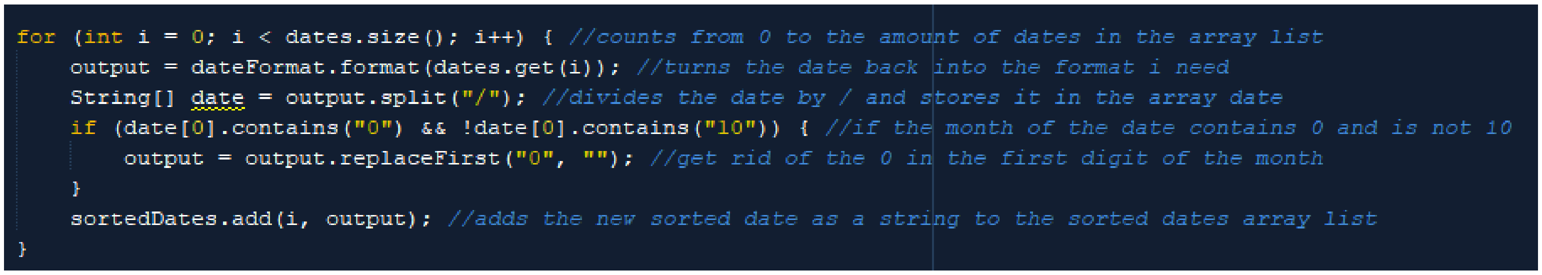
This algorithm works by comparing the name/names inputted to the line that it is scanning and if it matches, it divides the data, stores each individual part in the correct field of the athlete object, and stores that object in the athletes array list. It also increments a counter, "athletesFound," which I use as an error handling system to check if an athlete was found at all.

Sorting algorithm:



```
//this section uses the simple date format utility that creates date objects that have a specified format
SimpleDateFormat dateFormat = new SimpleDateFormat("MM/dd/yyyy"); //sets the format for dates using the simple date format utility
ArrayList<Date> dates = new ArrayList<>(); //initializes an array list for date objects
ArrayList<String> sortedDates = new ArrayList<>(); //initializes an arraylist for the sorted dates

for (int x = 0; x < athletes.size(); x++) { //counts from 0 to the amount of athletes in the array list so it gets them all
    //gets the date from the athlete object the for loop is currently on and stores it as a date object in the date array list
    dates.add(dateFormat.parse(athletes.get(x).getDate()));
}
//bubble sort method
for (int i = 0; i < dates.size() - 1; i++) { //counts from 0 to the amount of dates in the array list
    for (int j = 0; j < dates.size() - i - 1; j++) { //counts from 0 to the amount of dates in the array -1 -i (the amount of dates it has counted in the outer loop)
        if (dates.get(j).before(dates.get(j + 1))) { //if the date j is before the next date
            Date temp = dates.get(j); //a temporary date which date j in the array list is stored in
            Date temp2 = dates.get(j + 1); //a temporary date which date j + 1 in the array list is stored in
            dates.set(j, temp2); //moves the date j + 1 to where j was
            dates.set(j + 1, temp); //moves the date 1 to where j + 1 was
        }
    }
}
String output = dateFormat.format(dates.get(0)); //gets the latest date
```

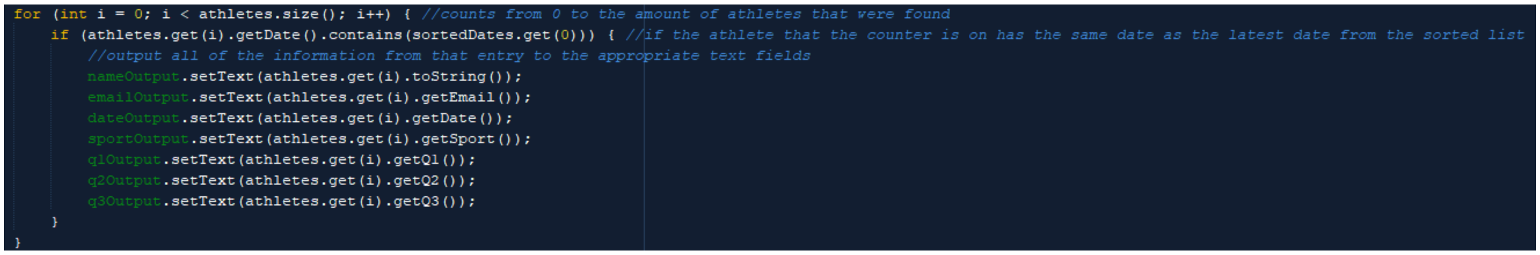
Here, I initialize all of the variables I will need and take the dates from the athletes stored in the athletes array list above and convert them into a SimpleDateFormat so that I can use the .before() method in the sorting. This allows me to implement a bubble sort method very easily because I can directly compare the dates without comparing their integers. The utility was extremely helpful and at the end of the sorting process, I convert all of the dates back into the form I need them in here:



```java
for (int i = 0; i < dates.size(); i++) { //counts from 0 to the amount of dates in the array list
    output = dateFormat.format(dates.get(i)); //turns the date back into the format i need
    String[] date = output.split("/"); //divides the date by / and stores it in the array date
    if (date[0].contains("0") && !date[0].contains("10")) { //if the month of the date contains 0 and is not 10
        output = output.replaceFirst("0", ""); //get rid of the 0 in the first digit of the month
    }
    sortedDates.add(i, output); //adds the new sorted date as a string to the sorted dates array list
}
```

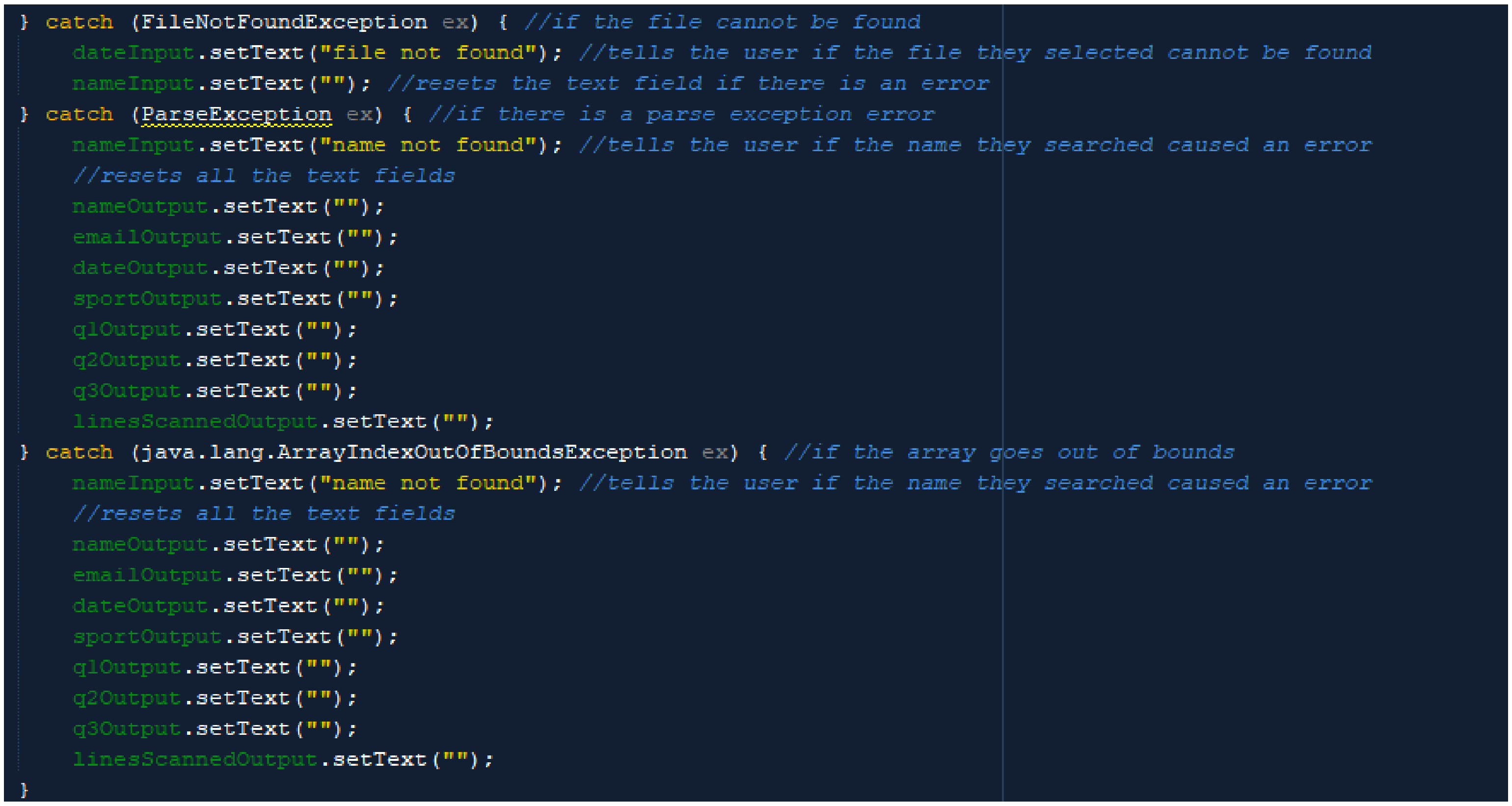
I also encountered errors where removing the 0 in the date to match the format of the google form for something like “10/1/2021” where there is a 0 in the date, but it needs to stay there. This is why I set the condition to get rid of the 0 only to be if it contains a 0 and is not 10.

<u>Output:</u>



```java
for (int i = 0; i < athletes.size(); i++) { //counts from 0 to the amount of athletes that were found
    if (athletes.get(i).getDate().contains(sortedDates.get(0))) { //if the athlete that the counter is on has the same date as the latest date from the sorted list
        //output all of the information from that entry to the appropriate text fields
        nameOutput.setText(athletes.get(i).toString());
        emailOutput.setText(athletes.get(i).getEmail());
        dateOutput.setText(athletes.get(i).getDate());
        sportOutput.setText(athletes.get(i).getSport());
        q1Output.setText(athletes.get(i).getQ1());
        q2Output.setText(athletes.get(i).getQ2());
        q3Output.setText(athletes.get(i).getQ3());
    }
}
```

Error handling:

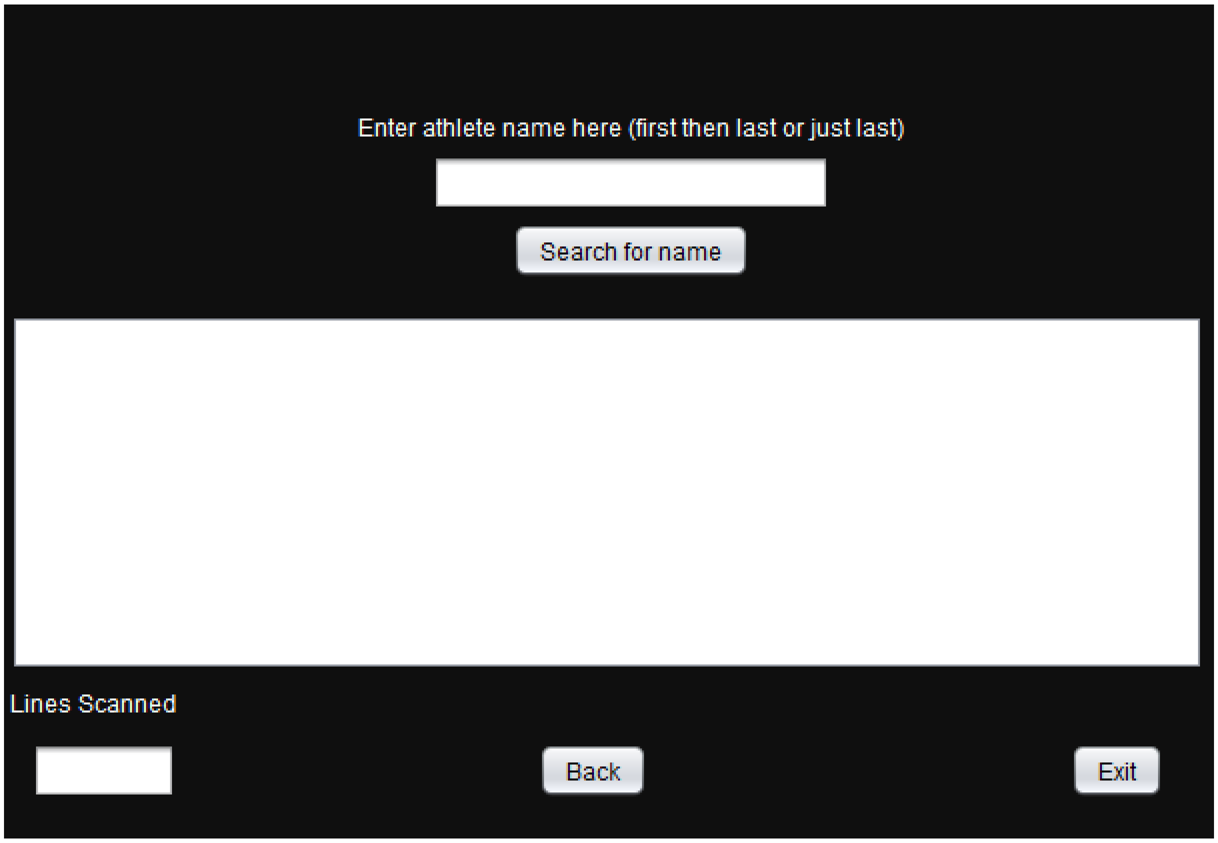


```
} catch (FileNotFoundException ex) { //if the file cannot be found
    dateInput.setText("file not found"); //tells the user if the file they selected cannot be found
    nameInput.setText(""); //resets the text field if there is an error
} catch (ParseException ex) { //if there is a parse exception error
    nameInput.setText("name not found"); //tells the user if the name they searched caused an error
    //resets all the text fields
    nameOutput.setText("");
    emailOutput.setText("");
    dateOutput.setText("");
    sportOutput.setText("");
    q1Output.setText("");
    q2Output.setText("");
    q3Output.setText("");
    linesScannedOutput.setText("");
} catch (java.lang.ArrayIndexOutOfBoundsException ex) { //if the array goes out of bounds
    nameInput.setText("name not found"); //tells the user if the name they searched caused an error
    //resets all the text fields
    nameOutput.setText("");
    emailOutput.setText("");
    dateOutput.setText("");
    sportOutput.setText("");
    q1Output.setText("");
    q2Output.setText("");
    q3Output.setText("");
    linesScannedOutput.setText("");
}
```

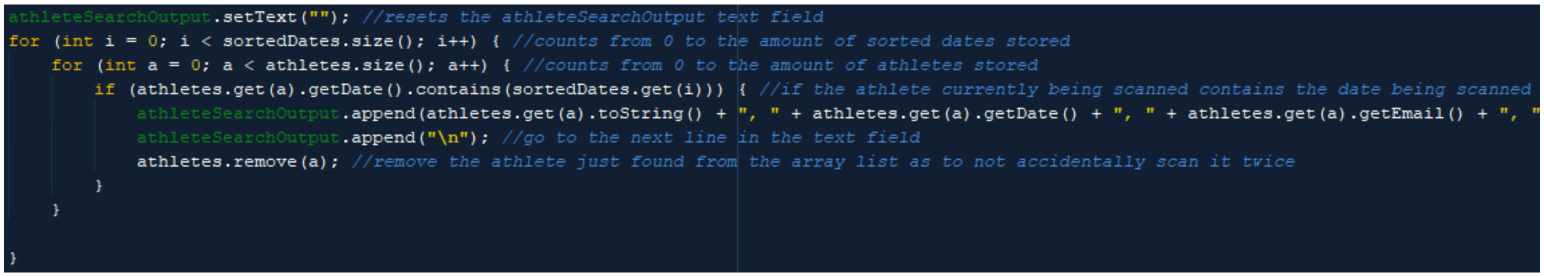
For the error handling, I have it catch general errors and reset all of the text fields to blank in addition to informing the user something went wrong.

Search by name list

<u>GUI:</u>



<u>Output:</u>

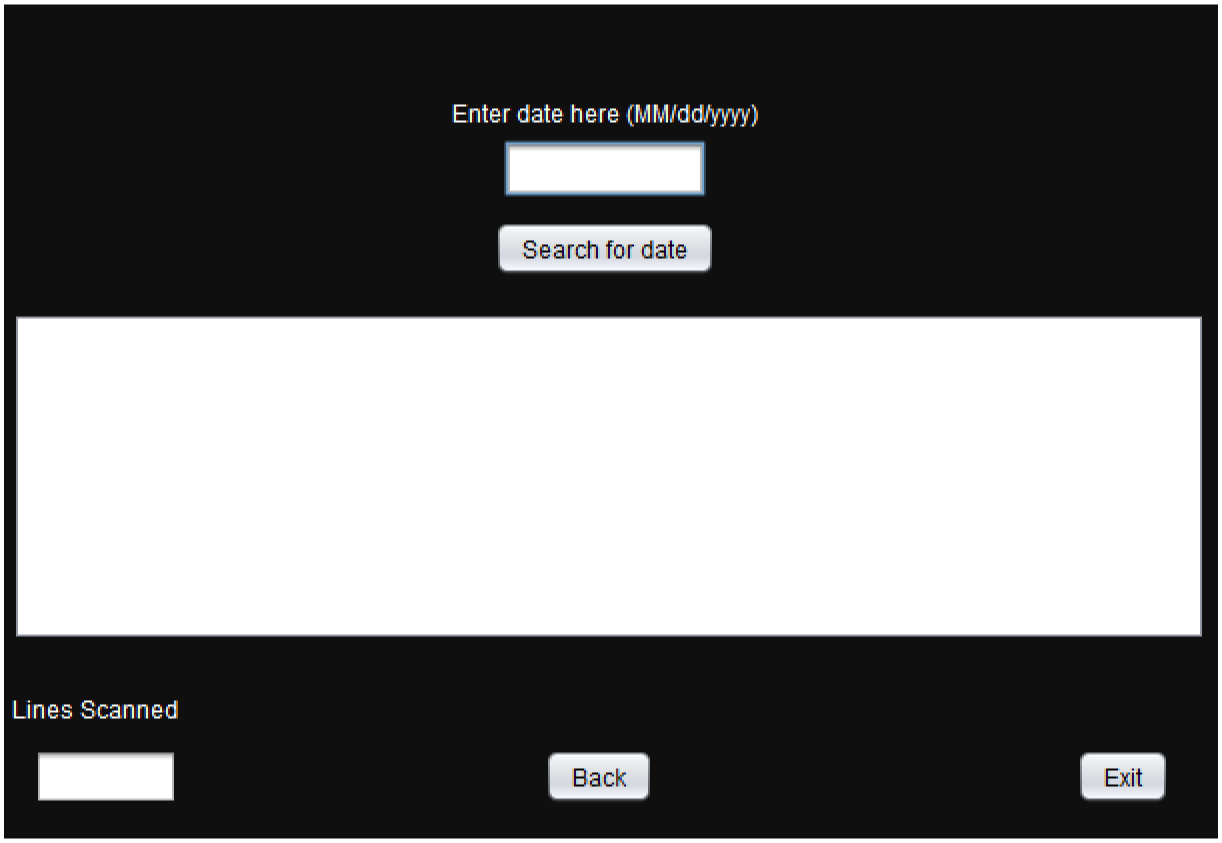


```
athleteSearchOutput.setText(""); //resets the athleteSearchOutput text field
for (int i = 0; i < sortedDates.size(); i++) { //counts from 0 to the amount of sorted dates stored
    for (int a = 0; a < athletes.size(); a++) { //counts from 0 to the amount of athletes stored
        if (athletes.get(a).getDate().contains(sortedDates.get(i))) { //if the athlete currently being scanned contains the date being scanned
            athleteSearchOutput.append(athletes.get(a).toString() + ", " + athletes.get(a).getDate() + ", " + athletes.get(a).getEmail() + ", "
            athleteSearchOutput.append("\n"); //go to the next line in the text field
            athletes.remove(a); //remove the athlete just found from the array list as to not accidentally scan it twice
        }
    }

}
```

Because I need to return all of the athletes in order, this return method compares a date to every athlete then removes the athlete from the array when it finds one to avoid errors. The output is also different from the search by name feature because it calls all the methods of the object to output on the same line.

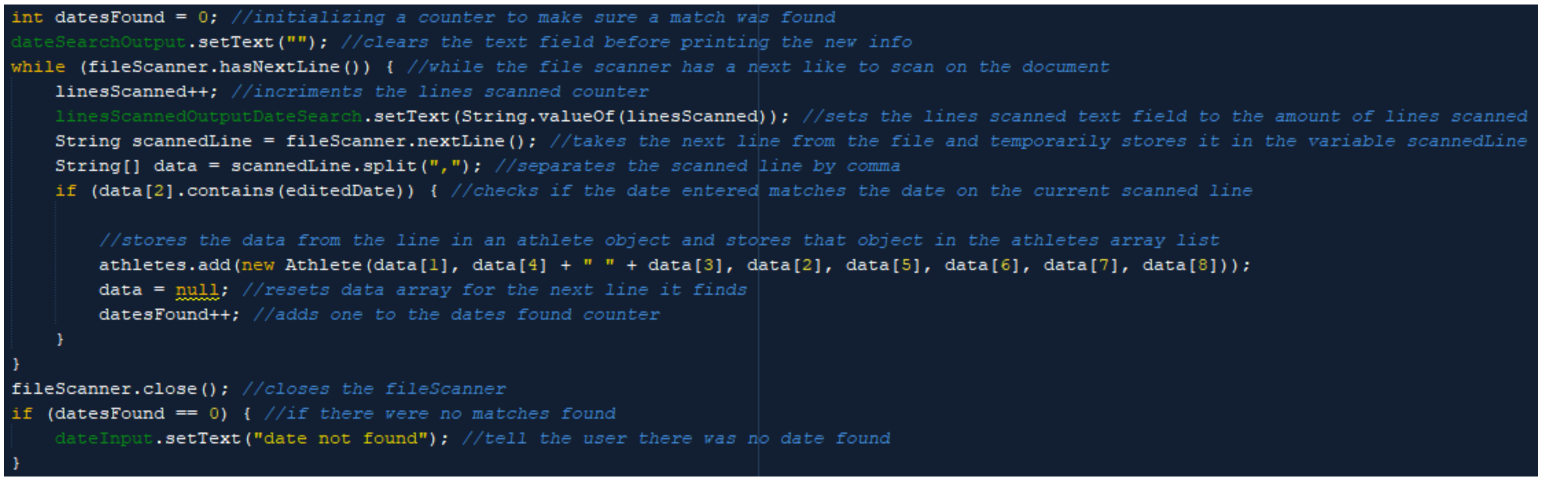
Search by date

GUI:



This feature had many more variations than the search by name and name list features. Most of the concepts involved doing things by date instead of name, which is not that difficult, but because all of the return values are the same date, I was able to remove the sorting algorithm.

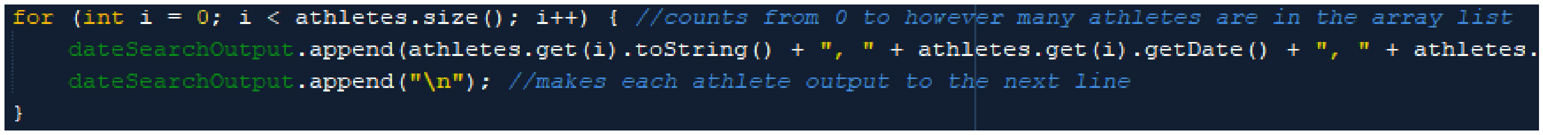
Search algorithm:



```
int datesFound = 0; //initializing a counter to make sure a match was found
dateSearchOutput.setText(""); //clears the text field before printing the new info
while (fileScanner.hasNextLine()) { //while the file scanner has a next like to scan on the document
    linesScanned++; //incriments the lines scanned counter
    linesScannedOutputDateSearch.setText(String.valueOf(linesScanned)); //sets the lines scanned text field to the amount of lines scanned
    String scannedLine = fileScanner.nextLine(); //takes the next line from the file and temporarily stores it in the variable scannedLine
    String[] data = scannedLine.split(","); //separates the scanned line by comma
    if (data[2].contains(editedDate)) { //checks if the date entered matches the date on the current scanned line

        //stores the data from the line in an athlete object and stores that object in the athletes array list
        athletes.add(new Athlete(data[1], data[4] + " " + data[3], data[2], data[5], data[6], data[7], data[8]));
        data = null; //resets data array for the next line it finds
        datesFound++; //adds one to the dates found counter
    }
}
fileScanner.close(); //closes the fileScanner
if (datesFound == 0) { //if there were no matches found
    dateInput.setText("date not found"); //tell the user there was no date found
}
```

As can be seen in the code, I search for the date instead of the name, but everything else is the exact same.

<u>Output:</u>



```
for (int i = 0; i < athletes.size(); i++) { //counts from 0 to however many athletes are in the array list
    dateSearchOutput.append(athletes.get(i).toString() + ", " + athletes.get(i).getDate() + ", " + athletes.
    dateSearchOutput.append("\n"); //makes each athlete output to the next line
}
```

Word count: 1132