Criterion C: Development

Techniques Used:

1. Object Oriented Programming
2. Conditional Logic
3. Iteration
4. Arrays/Array Lists
5. File Handling
6. Error Handling (try-catch)

Conditional Logic

```
public static void recommend(){  1 usage
    clubdatabase dab = new clubdatabase();
    Scanner input = new Scanner(System.in);

    System.out.println("Enter your shot distance: ");
    double shotdistance = input.nextDouble();

    System.out.println("Enter wind speed (mph): ");
    double windSpeed = input.nextDouble();

    System.out.println("Is the wind blowing TOWARD you or AWAY from you? (enter toward/away): ");
    String direction = input.next().toLowerCase();//this reads the input from the last question and makes sure it is lowercase
    //the goal of these conditional statements are to adjust the recommended club based on shot factors
    if (direction.equals("away")){
        shotdistance -= windSpeed;//wind blowing away makes an easier shot
    }
    else if (direction.equals("toward")){
        shotdistance += windSpeed;//wind toward makes an harder shot
    }
    else{
        System.out.println("Invalid direction");
    }

    System.out.println("Enter terrain (fairway(f)/rough(r)/deeprough(de)/bunker(b)/uphill(u)/downhill(d)): ");
    String terrain = input.next().toLowerCase();
    //adjusts the recommended club based on the terrain
    if (terrain.equals("r")){
        shotdistance = shotdistance * 1.1;//multipliers change the shot distance based on terrain
    }
    if (terrain.equals("de")){
        shotdistance = shotdistance * 1.3;
    }
    if (terrain.equals("u")){
        shotdistance = shotdistance * 1.1;
    }
    if (terrain.equals("d")){
        shotdistance = shotdistance * 0.9;
    }
    if (terrain.equals("b")){
        shotdistance = shotdistance * 1.25;//bunker shots are very difficult
    }
```

This code is a vital aspect of my club recommendation system. There are a multitude of questions

being asked of the user.

The first question covers the shot distance. Then, the user is asked to input the wind speed. Once the user has stated the wind speed, they are asked for the wind's direction. A series of conditional logic (if/else statements) is used to adjust the shot distance. If the wind is blowing away from the user, the wind speed is subtracted from the shot distance. Then an 'else if' is used to add the wind speed to the shot distance if the wind is blowing towards the user. An else statement is used at the end in case the user enters an invalid option. Then the user is asked about the terrain condition. There is a series of if statements used based on what the user indicated as the terrain condition. Once a type of terrain is determined by the if statements, multipliers are used on the shot distance to affect them accordingly.

The reason I chose if/else statements for this recommendation system is that I wanted to make sure only one condition can happen at a time. If/Else statements make sure that only one condition is applied, and they handle invalid inputs at the end with an else. Another reason I chose If/Else statements is their readability and maintainability. It is easy to add another feature to my recommendation system by just adding one more if statement. The structure of the If/Else statements is clear and logical, making it easy for me to explain how my algorithm works to peers or coworkers that I am working with.

I could have also used multiple switch statements for this algorithm, but switch statements are not the best at handling Boolean logic that my algorithm required because they evaluate one value at a time.

Iteration

```
for (club c : clubset) {
    System.out.println("Enter your average distance for " + c.getName() + ": ");//Asks the user for each of their club distances
    c.setDistance(personalDistance.nextDouble());//saves the users input to each club object by using the setter method. nextDouble() is a method from the scanner class that will convert
}
saveClubsToFile();//Saves the users club distances to the CSV file
```

```
for(club c : dab.getClubset())//loops through every club in the clubset
{
    double difference = Math.abs(c.getDistance() - shotdistance);//this calculates how far the club's distance is from the required distance
    if (difference < smallestdif) {//if this clubs is closer to target distance
        smallestdif = difference;//update smallest differece found
        closest = c;//store club as the closest matching club
    }
}
```

Both of these examples use a for-each loop. The first for-each loop goes through every club in the

user's set and asks the user to enter their average distance for that club. Then their input is stored as a Club object. The second for-each loop compares the adjusted shot distance to the user's average club distances. It calculates the difference between them to find the most fitting club. A for-each loop was used instead of a for loop since the logic in both of these iterations needed to compare values, not positions. A for-each loop is an easier and more straightforward way to do a linear search through the clubset.

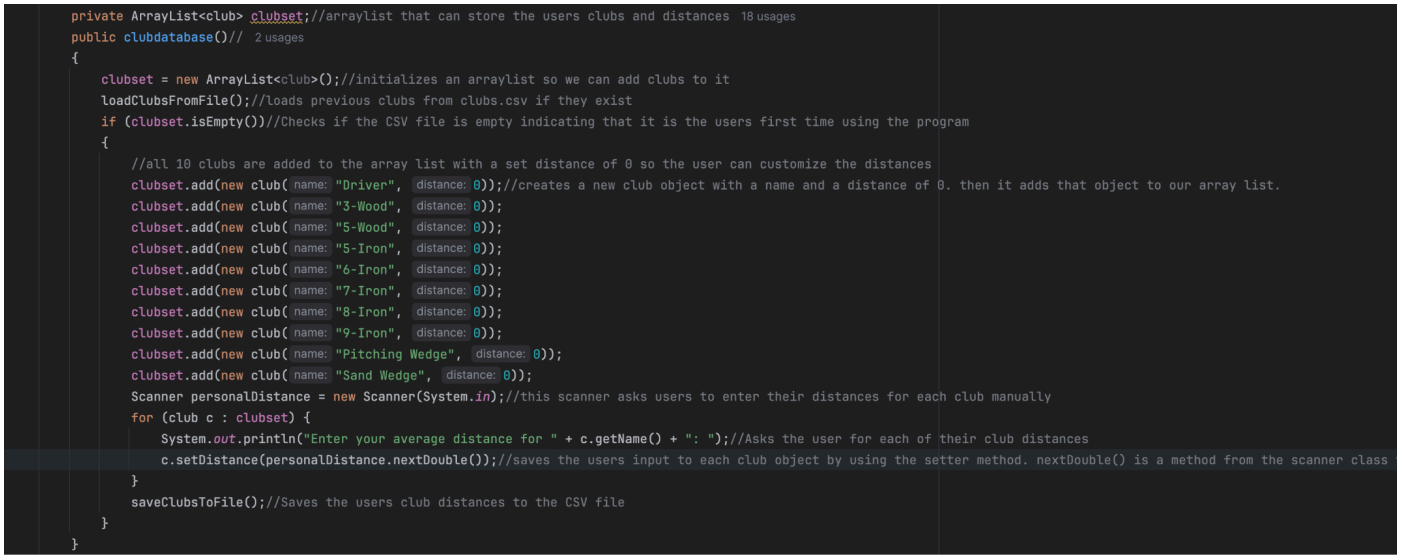
```
while(true){//infinite loop so menu keeps showing up until user exits
    System.out.println("\n~~~ Teal Bend Golf Caddie Menu ~~~");//display menu options
    System.out.println("1. View your clubs");
    System.out.println("2. Update a club's distance");
    System.out.println("3. Recommend a club");
    System.out.println("4. Add a note");
    System.out.println("5. View notes");
    System.out.println("6. Start a new round");
    System.out.println("7. View historical scores");
    System.out.println("8. Exit");
    System.out.print("Choose what you want to do:");
    int choice = input.nextInt();//tells user to make a selection
    input.nextLine();//reads the users choice
```

In this while loop, the home screen main menu is displayed. I used a while loop because we can't assume how many times the menu will need to reappear for the user. By setting a while loop to true, I can keep the home screen reappearing until the user wants to exit.

```
for(int i=1;i<=18;i++) {
    notes.viewNotes(i);//displays notes based on current hole
    System.out.println("Enter your score for hole #" + i);//tells user to enter score
    int score = input.nextInt();//reads score
    totalScore += score;//adds it to total score
```

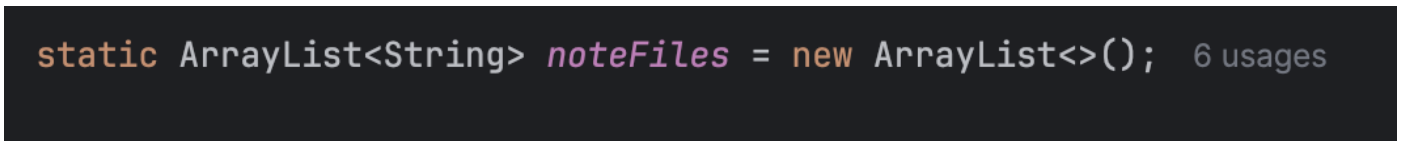
In this for loop, it displays notes associated with the current hole that the user is on. The for loop iterates over all 18 holes for a standard golf round. Since a standard golf round always has 18 holes, a regular for loop made the most sense since I could automatically set it to 18 iterations.

Data Structures



```
private ArrayList<club> clubset;//arraylist that can store the users clubs and distances  18 usages
public clubdatabase()//  2 usages
{
    clubset = new ArrayList<club>();//initializes an arraylist so we can add clubs to it
    loadClubsFromFile();//Loads previous clubs from clubs.csv if they exist
    if (clubset.isEmpty())//Checks if the CSV file is empty indicating that it is the users first time using the program
    {
        //all 10 clubs are added to the array list with a set distance of 0 so the user can customize the distances
        clubset.add(new club( name: "Driver",  distance: 0));//creates a new club object with a name and a distance of 0. then it adds that object to our array list.
        clubset.add(new club( name: "3-Wood",  distance: 0));
        clubset.add(new club( name: "5-Wood",  distance: 0));
        clubset.add(new club( name: "5-Iron",  distance: 0));
        clubset.add(new club( name: "6-Iron",  distance: 0));
        clubset.add(new club( name: "7-Iron",  distance: 0));
        clubset.add(new club( name: "8-Iron",  distance: 0));
        clubset.add(new club( name: "9-Iron",  distance: 0));
        clubset.add(new club( name: "Pitching Wedge",  distance: 0));
        clubset.add(new club( name: "Sand Wedge",  distance: 0));
        Scanner personalDistance = new Scanner(System.in);//this scanner asks users to enter their distances for each club manually
        for (club c : clubset) {
            System.out.println("Enter your average distance for " + c.getName() + ": ");//Asks the user for each of their club distances
            c.setDistance(personalDistance.nextDouble());//saves the users input to each club object by using the setter method. nextDouble() is a method from the scanner class
        }
        saveClubsToFile();//Saves the users club distances to the CSV file
    }
}
```

This ArrayList called ‘club’ stores all of the user’s golf clubs as Club objects. Each Club object contains a club name and an average distance. 10 clubs are added to the array list with a name and an average distance of 0. Then the user is asked to input each distance based on their information. An ArrayList was used because there isn’t a fixed number of clubs. Since ArrayLists can grow and shrink compared to other data structures (arrays, linked lists, stacks, queues), it was the perfect choice, as some people have more clubs than others.



```
static ArrayList<String> noteFiles = new ArrayList<>();  6 usages
```

This line of code creates a static ArrayList that stores the names of note files as strings. Each element in this ArrayList is a saved notes file, including specific hole notes and general notes. The reason it is static is so that the note files can be shared across my entire program (they are used in more than one part of it). I chose to use an ArrayList (compared to Arrays that are static) because I didn't know how many note files I would eventually need, so I wanted a dynamic data structure in case I needed to add or remove note files.

File Handling

```
try {
    FileWriter writer = new FileWriter( fileName: "rounds.csv", append: true); //creates FileWriter object
    writer.write( str: date + ", " + totalScore + "\n");//writes date and total score as new line in CSV
    writer.close();//closes FileWriter
    System.out.println("Round saved to rounds.csv");//Tells user the round was saved
}
catch (IOException e) {
    System.out.println("Error saving round to CSV.");//Handels any input errors
}
```

I created a FileWriter object that allows for the user's input to be added to a CSV file so that it can be permanently saved. Then the file is closed for good practice. I decided to use a CSV format to handle my files because it doesn't require external libraries like BufferedWriter, and stores my data in a clear structure. It is extremely simple to read, and I am able to manually edit it too. Also, many golfers would want to export their data onto a spreadsheet; my CSV format is compatible with spreadsheets like Excel and Sheets.

Error Handling (try-catch)

```
public void loadClubsFromFile()  1 usage
//loads the users saves clubs from clubs.csv
{
    try {
        Scanner fileIn = new Scanner(new File( pathname: "clubs.csv"));//tries to open the clubs.csv file
        while (fileIn.hasNext()) {//reads each line in the clubs.csv file
            String lineIn = fileIn.nextLine();//reads the whole line as text
            String[] parts = lineIn.split( regex: ",");//splits the line up into two pieces that are seperated by a comma
            if (parts.length == 2) {//only add the club if the line has 2 parts so it can protect the program against corrupted files
                String name = parts[0];//take the name out of the first part and remove any spaces
                double distance = Double.parseDouble(parts[1]);//convert the second part into a double
                clubset.add(new club(name, distance));//create a new club object with this name and distance and add it to the club list
            }
        }
        fileIn.close();//done reading the file
        System.out.println("Loaded clubs from clubs.csv");//confirming that the clubs were loaded
    } catch (IOException e) {
        // File not found is normal on first run
        System.out.println("No clubs.csv found, will create new one.");
    }
}
```

The goal of this method is to load data related to golf clubs from a CSV. Within the ‘try’ part of this method, the program opens clubs.csv and uses a loop to read every line in the file. Then it uses a comma to separate the club name and distance on every line. After loading all the data, it lets the user know it is done telling the user that it is done loading. Within the ‘catch’ part of this code, the program will catch exceptions and let the user know it ran into an error. The try-catch error handling technique was used to keep the

program from crashing if clubs.csv never existed. Also, it informs the user well if an error occurred. I didn't want to use 'throw IOException' because it would not handle the error where it occurred.

Object Oriented Programming

```
public class club { //creates the club object used throughout this program
   private double avgDistance;//one attribute of the club is the average distance the
user is able to hit it in yards
   private String name;// another attribute of the club is the name of it
   public club(String name, double distance)//initializing both private instance
variables in the constructor
   {
       this.name = name;
       avgDistance = distance;
   }
   public double getDistance()//getter method that returns the distance of a club object
   {
       return avgDistance;
   }
   public  void setDistance(double distance) {// setter method that allows a user to set
a new average distance for a club
       avgDistance = distance;
   }
   public String getName()// getter method that allows a user to access the name of their
club
   {
       return name;
   }

   public String toString() {//prints a club in a readable format with all attributes of
it included
       return name + ": " +  avgDistance;
```

```
    }
}
```

This snippet demonstrates how I used object oriented programming throughout my code. This is a club class that represents a golf club that has two features: a name and an average distance. The constructor initializes each golf club object, while the getter/setter methods allow the user to receive and modify their club features.

I decided to use object-oriented programming for two main reasons: data security and maintainability. My data is protected through the use of encapsulation; the strategic use of public and private makes sure my data is not misused. Modularity allows my code to be broken down into simpler methods so that my program is easier to interpret and maintain. I don't have to worry about affecting my entire code when trying to fix small errors thanks to modularity.

Word Count: 1075

Works Cited:

"ChatGPT." *ChatGPT*, 2025, chatgpt.com/g/g-p-68dab8bda97881918501f3a99a74b6af-computer-science-ia-aditya-parekh/project. Accessed 11 Jan. 2026.

GeeksforGeeks. "Java FileWriter Class." *GeeksforGeeks*, 14 Dec. 2020, www.geeksforgeeks.org/java/filewriter-class-in-java/.

W3Schools. "Java Exceptions (Try...Catch)." *W3Schools*, 2020, www.w3schools.com/java/java_try_catch.asp.