Criterion C: Development

List of Simple Techniques used for functionality purposes, not included due to the limitations of the word count
- For loops - Menu driven Switch/Case Interface - Data hiding - Constructors, Getters, Setters - Polymorphism - If statements / Else if statements - Input validation using while loops

Technique: Serialization and Deserialization
Link to success criteria: - The system must be able to store data from older sessions and keep them saved so that the system does not reset every time the application is closed - The client should be able to store the data of the sessions and the exercises on a local file.
Source: (Liguori and Liguori)

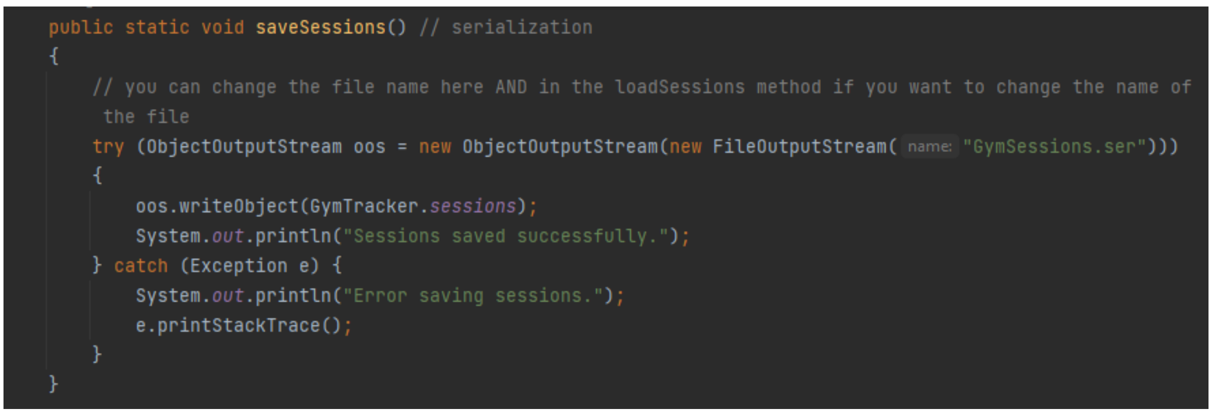
Justification:

The client needed to be able to reopen the system and not have to enter the exercises and sessions all over again in order to view the sessions or exercises. By using serialization I was able to save the data so that the data could be restored to the system after being reopened once the file was loaded into the program.

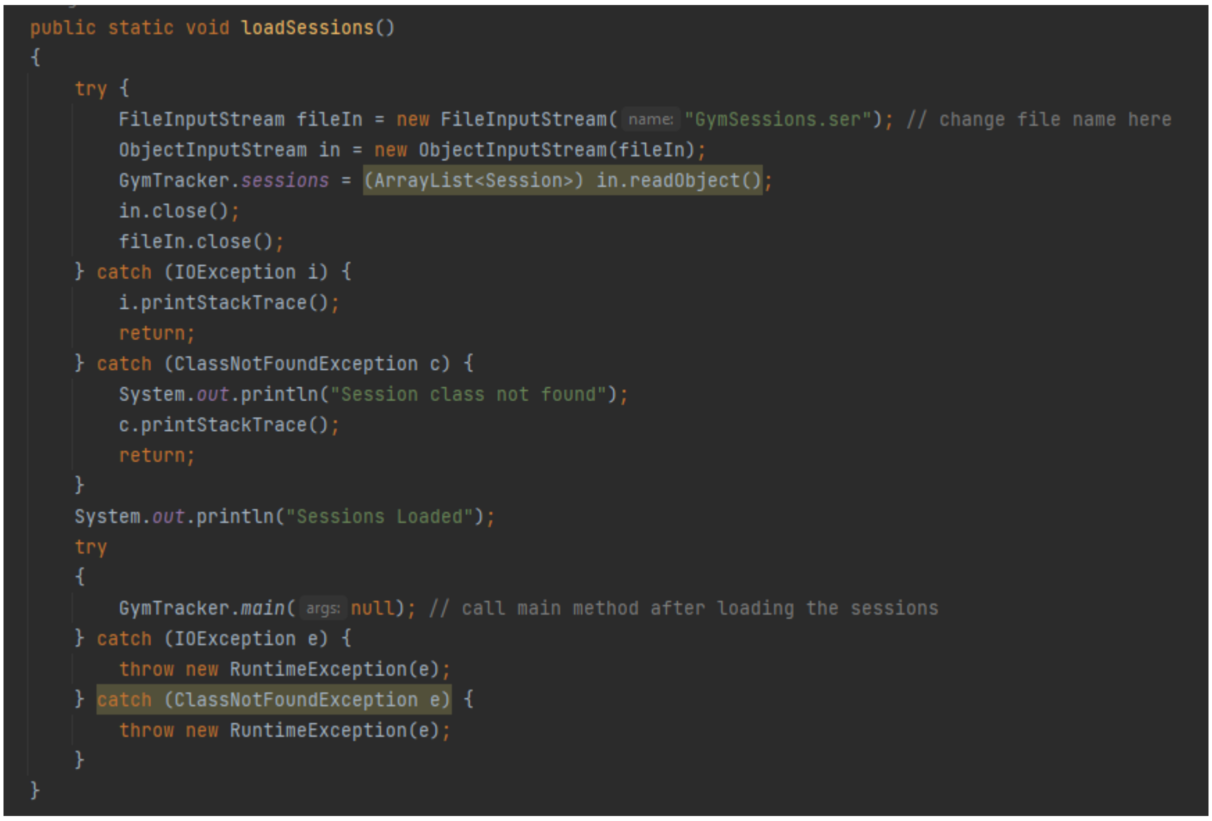
Explanation:

The Session class has 2 functions, saveSession and loadSession, which use serialization and deserialization respectively. The saveSession method saves an ArrayList of sessions to a file called "GymSessions.ser" which saves the data contained by the ArrayList of sessions, including the date, exercises, weight, muscle group, sets, and reps. Deserialization is used by the loadSession method which loads the data into the program and calls the main method again in order for the program to continue running.

Example:



```
public static void saveSessions() // serialization
{
    // you can change the file name here AND in the loadSessions method if you want to change the name of
     the file
    try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream( name: "GymSessions.ser")))
    {
        oos.writeObject(GymTracker.sessions);
        System.out.println("Sessions saved successfully.");
    } catch (Exception e) {
        System.out.println("Error saving sessions.");
        e.printStackTrace();
    }
}
```



```java
public static void loadSessions()
{
    try {
        FileInputStream fileIn = new FileInputStream( name: "GymSessions.ser"); // change file name here
        ObjectInputStream in = new ObjectInputStream(fileIn);
        GymTracker.sessions = (ArrayList<Session>) in.readObject();
        in.close();
        fileIn.close();
    } catch (IOException i) {
        i.printStackTrace();
        return;
    } catch (ClassNotFoundException c) {
        System.out.println("Session class not found");
        c.printStackTrace();
        return;
    }
    System.out.println("Sessions Loaded");
    try
    {
        GymTracker.main( args: null); // call main method after loading the sessions
    } catch (IOException e) {
        throw new RuntimeException(e);
    } catch (ClassNotFoundException e) {
        throw new RuntimeException(e);
    }
}
```

Technique: Binary Search
Link to success criteria: - The system should be able to search through the sessions for certain exercises, and organise/sort the exercises based on muscle group.
Source: (Tutorialspoint)

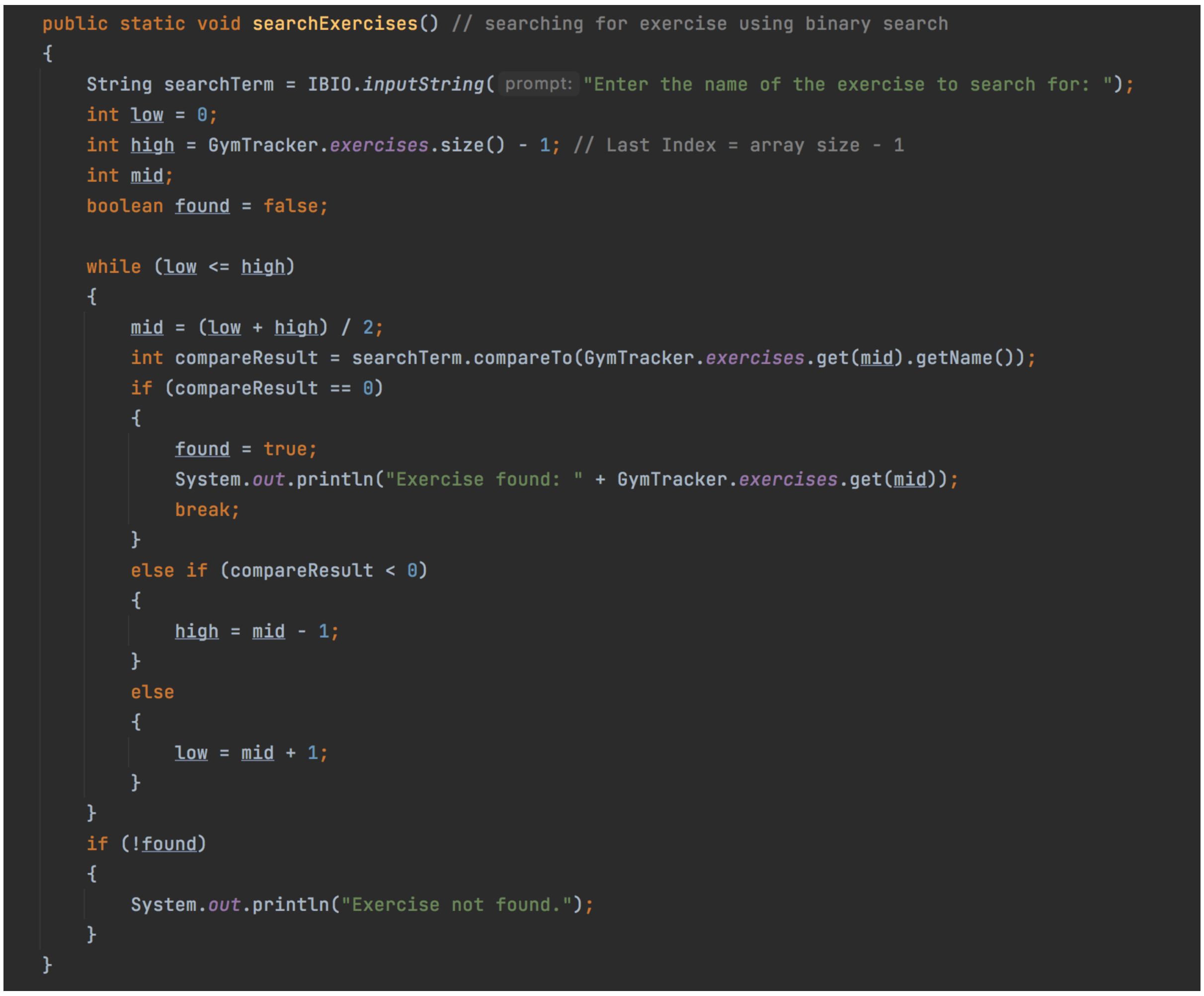
Justification:

A method to search was needed, and binary searches are very efficient with a worst case scenario run time of $O(log\ n)$.

Explanation:

The Exercise class has a function named searchExercise, where the middle of a sorted array is used and the searched term is compared to the median term, eliminating half of the remaining options as the search term will either be lower or higher. Hence it is a very efficient method for searching.

Example:



```java
public static void searchExercises() // searching for exercise using binary search
{
    String searchTerm = IBIO.inputString( prompt: "Enter the name of the exercise to search for: ");
    int low = 0;
    int high = GymTracker.exercises.size() - 1; // Last Index = array size - 1
    int mid;
    boolean found = false;

    while (low <= high)
    {
        mid = (low + high) / 2;
        int compareResult = searchTerm.compareTo(GymTracker.exercises.get(mid).getName());
        if (compareResult == 0)
        {
            found = true;
            System.out.println("Exercise found: " + GymTracker.exercises.get(mid));
            break;
        }
        else if (compareResult < 0)
        {
            high = mid - 1;
        }
        else
        {
            low = mid + 1;
        }
    }
    if (!found)
    {
        System.out.println("Exercise not found.");
    }
}
```

Technique: Selection Sort
Link to success criteria: - The system should be able to search through the sessions for certain exercises, and organise/sort the exercises based on muscle group.
Source: (Drien Vargas)

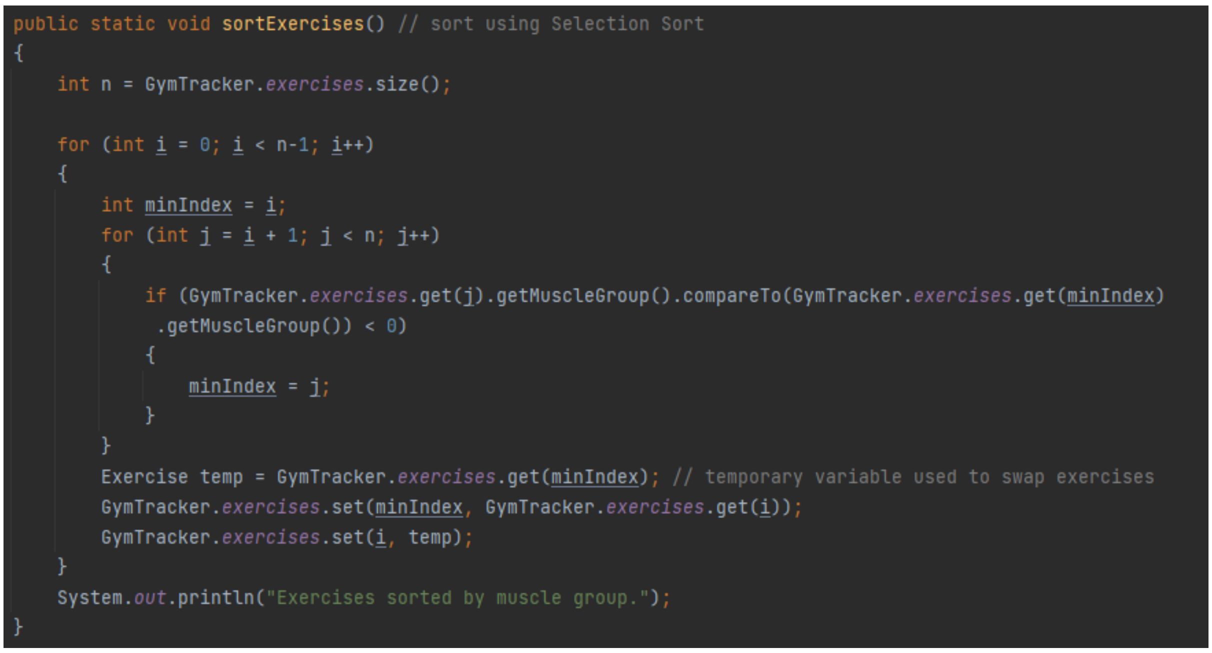
Justification:

In order for the binary search to work the Array needs to be sorted, hence by using a selection sort I am able to sort the array through multiple traversals.

Explanation:

The Exercise class has a function named sortExercise, where the Exercises are sorted by muscleGroup using selection sort in order for the user to see how many exercises they are doing per muscle group, and can then use this information to adapt their sessions.

Example:



```java
public static void sortExercises() // sort using Selection Sort
{
    int n = GymTracker.exercises.size();

    for (int i = 0; i < n-1; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < n; j++)
        {
            if (GymTracker.exercises.get(j).getMuscleGroup().compareTo(GymTracker.exercises.get(minIndex)
             .getMuscleGroup()) < 0)
            {
                minIndex = j;
            }
        }
        Exercise temp = GymTracker.exercises.get(minIndex); // temporary variable used to swap exercises
        GymTracker.exercises.set(minIndex, GymTracker.exercises.get(i));
        GymTracker.exercises.set(i, temp);
    }
    System.out.println("Exercises sorted by muscle group.");
}
```

Technique: Aggregation
Link to success criteria: - The client must be able to input the name of the exercise, weight/resistance used, number of sets, and number of repetitions. Which should all be stored into different sessions containing multiple exercises.
Source: (Drien Vargas)

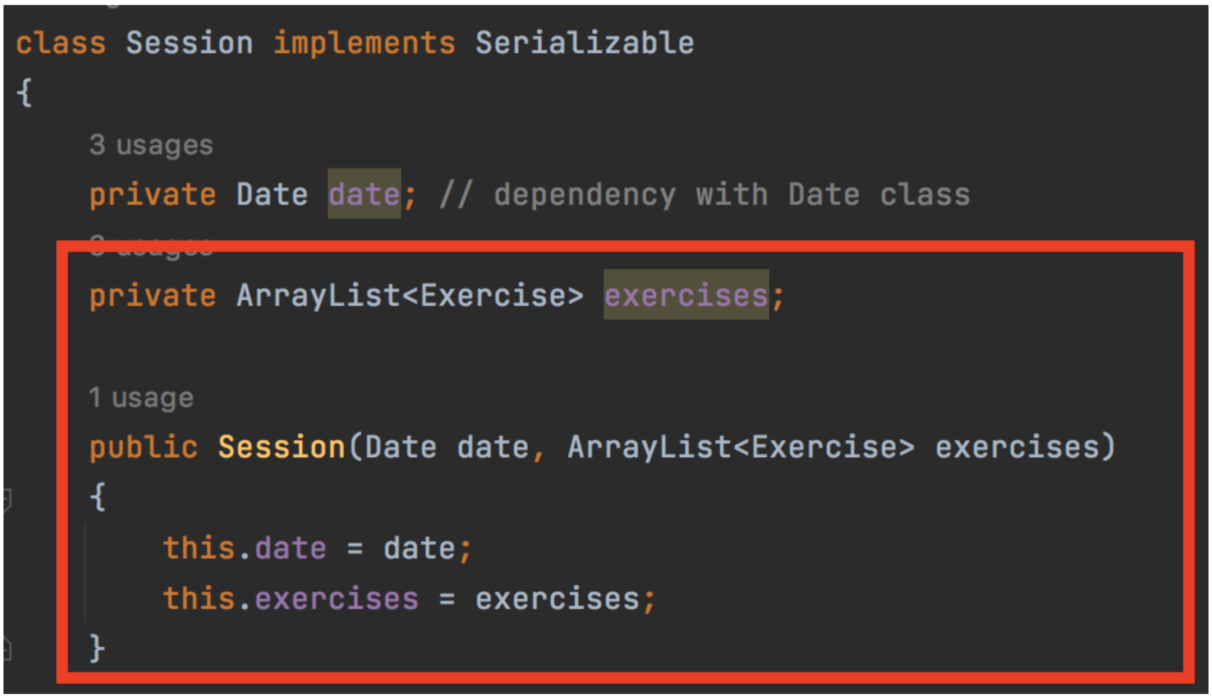
Justification:

Aggregation allows one class to contain another, in which every session contains multiple exercises. Meaning there is a has-a relationship which uses aggregation, this allows for a more efficient and connected system by adding different exercises to different sessions.

Explanation:

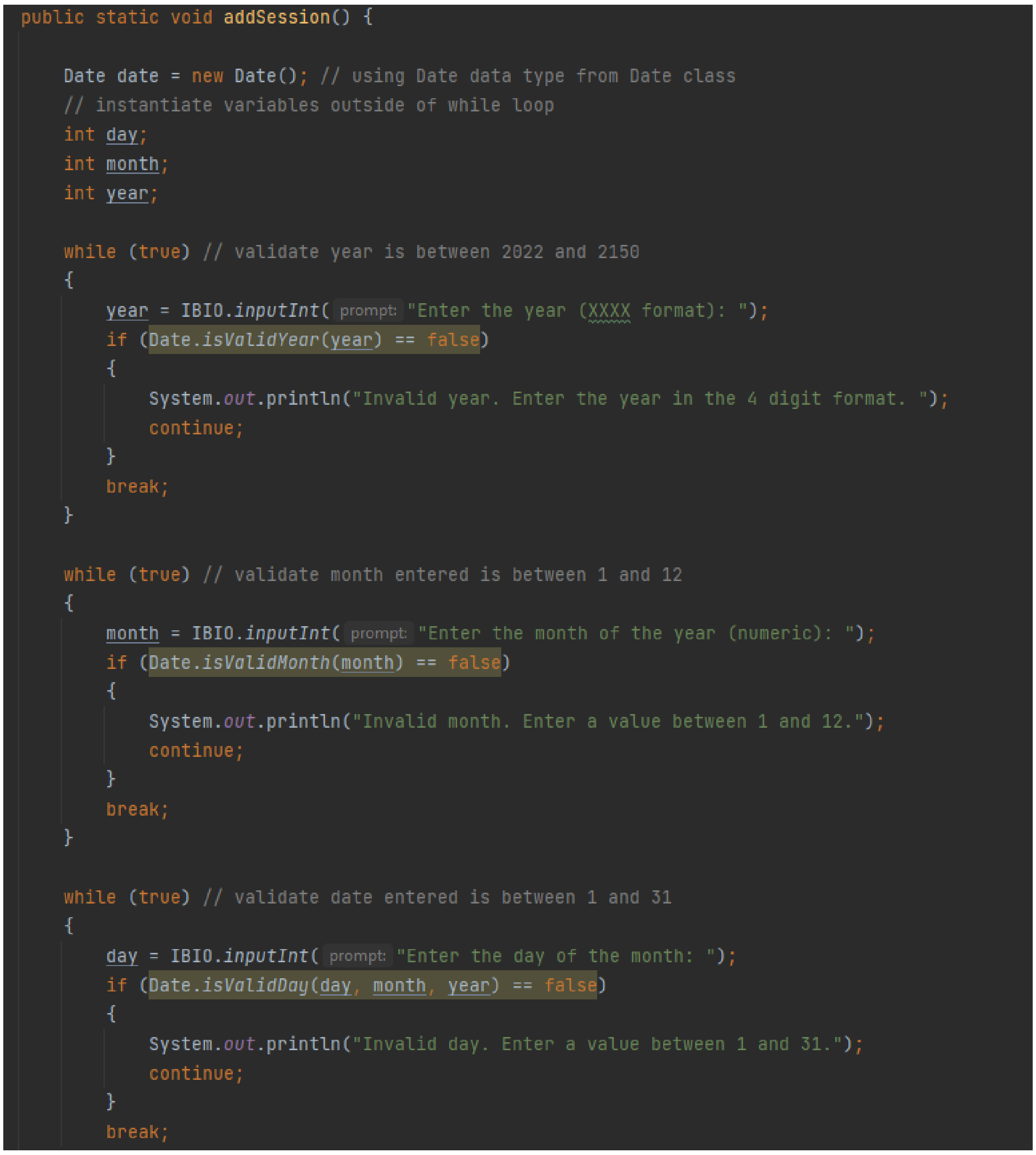
The Session class contains an ArrayList that aggregates all the classes that the Exercise class contains. In which the ArrayList of sessions is made using an arraylist of exercises in order for every session to be able to contain multiple exercises, so there is a 1 - n relationship between sessions and exercises.

Example:



```
class Session implements Serializable
{
    3 usages
    private Date date; // dependency with Date class
    private ArrayList<Exercise> exercises;

    1 usage
    public Session(Date date, ArrayList<Exercise> exercises)
    {
        this.date = date;
        this.exercises = exercises;
    }
```



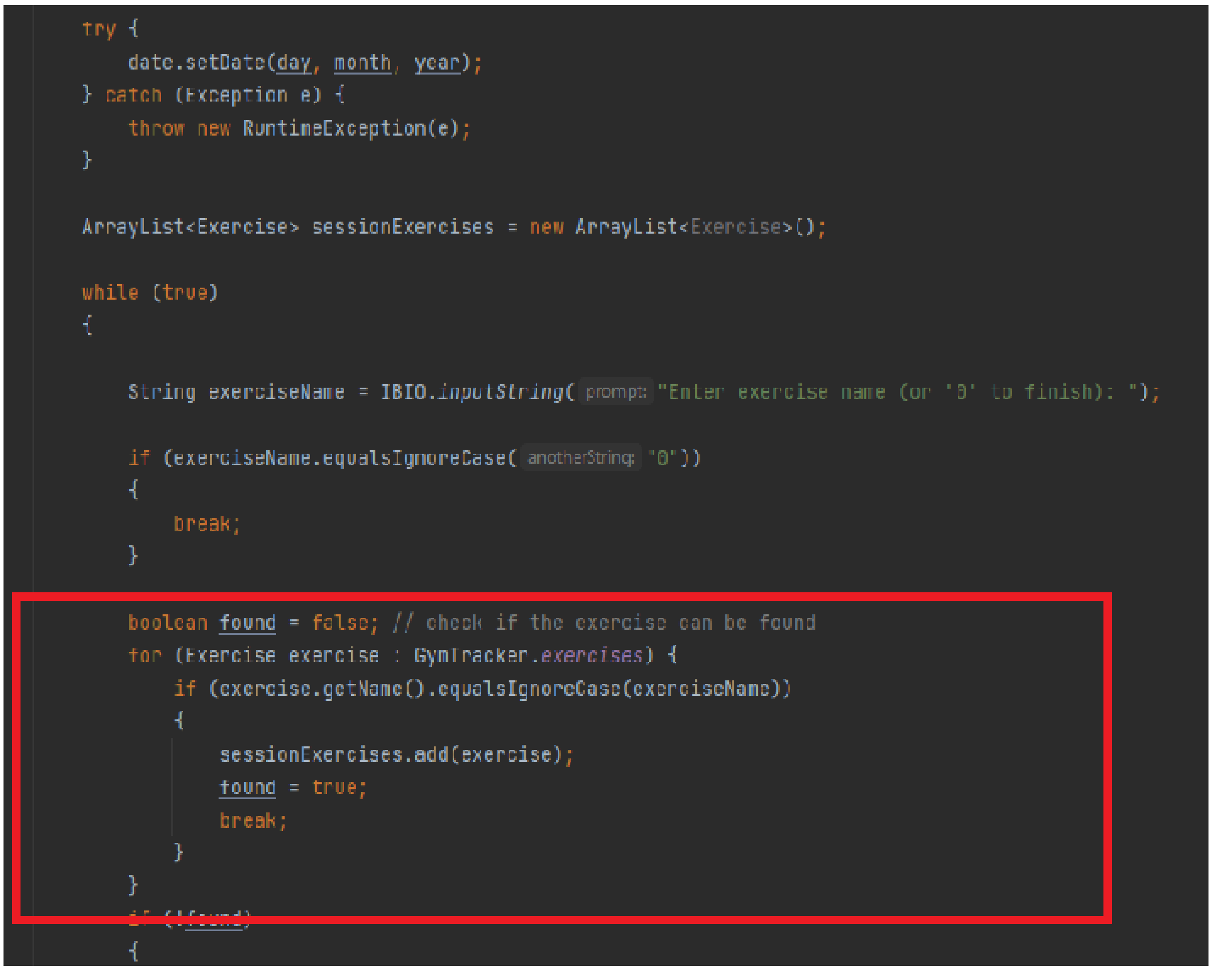
```
public static void addSession() {

    Date date = new Date(); // using Date data type from Date class
    // instantiate variables outside of while loop
    int day;
    int month;
    int year;

    while (true) // validate year is between 2022 and 2150
    {
        year = IBIO.inputInt( prompt: "Enter the year (XXXX format): ");
        if (Date.isValidYear(year) == false)
        {
            System.out.println("Invalid year. Enter the year in the 4 digit format. ");
            continue;
        }
        break;
    }

    while (true) // validate month entered is between 1 and 12
    {
        month = IBIO.inputInt( prompt: "Enter the month of the year (numeric): ");
        if (Date.isValidMonth(month) == false)
        {
            System.out.println("Invalid month. Enter a value between 1 and 12.");
            continue;
        }
        break;
    }

    while (true) // validate date entered is between 1 and 31
    {
        day = IBIO.inputInt( prompt: "Enter the day of the month: ");
        if (Date.isValidDay(day, month, year) == false)
        {
            System.out.println("Invalid day. Enter a value between 1 and 31.");
            continue;
        }
        break;
```



```
try {
    date.setDate(day, month, year);
} catch (Exception e) {
    throw new RuntimeException(e);
}

ArrayList<Exercise> sessionExercises = new ArrayList<Exercise>();

while (true)
{

    String exerciseName = IBIO.inputString( prompt: "Enter exercise name (or '0' to finish): ");

    if (exerciseName.equalsIgnoreCase( anotherString: "0"))
    {
        break;
    }

    boolean found = false; // check if the exercise can be found
    for (Exercise exercise : GymTracker.exercises) {
        if (exercise.getName().equalsIgnoreCase(exerciseName))
        {
            sessionExercises.add(exercise);
            found = true;
            break;
        }
    }
    if (!found)
    {
```

Technique: ArrayLists
Link to success criteria: - The client must be able to input the name of the exercise, weight/resistance used, number of sets, and number of repetitions. Which should all be stored into different sessions containing multiple exercises.
Source: (Geeksforgeeks)

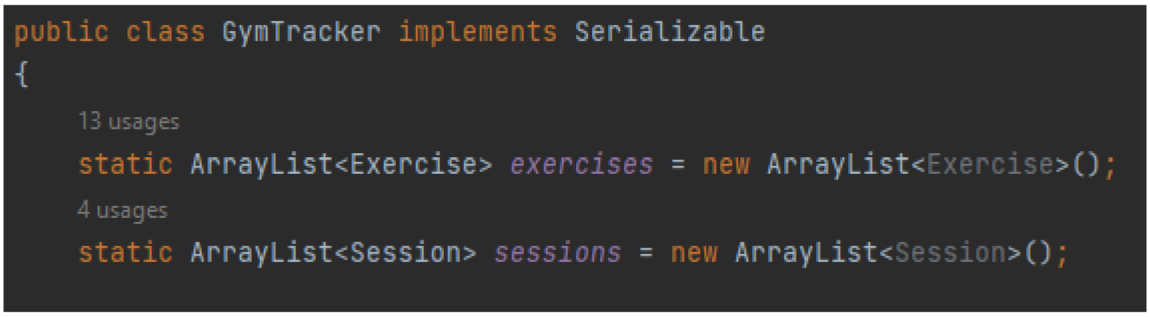
Justification:

In order to store all the required variables and data where they would all be linked, meaning each exercise would have data such as reps, weight, sets linked to it, and each session would have multiple exercises linked to each session. Upon conducting research the best way to do this was with either ArrayLists or HashMaps, however ArrayLists stores elements only as values, which results in ArrayLists taking up less memory.

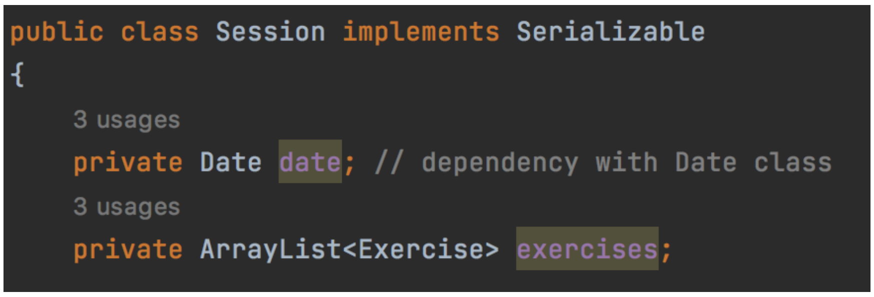
Explanation:

By using an ArrayList for each exercise, as well as an ArrayList for each session I was able to link each session to multiple exercises, while having each individual exercise have individual values for its specific variables (name, muscleGroup, sets etc.)

Example:



```java
public class GymTracker implements Serializable
{
    13 usages
    static ArrayList<Exercise> exercises = new ArrayList<Exercise>();
    4 usages
    static ArrayList<Session> sessions = new ArrayList<Session>();
```



```java
public class Session implements Serializable
{
    3 usages
    private Date date; // dependency with Date class
    3 usages
    private ArrayList<Exercise> exercises;
```

Technique: Try-Catch blocks for Exceptions
Link to success criteria: - The system should be able to prevent the input of invalid data, incorrectly formatted data, or wrong data; and any data that does not match the requirements should be correctly managed
Source: (W3Schools)

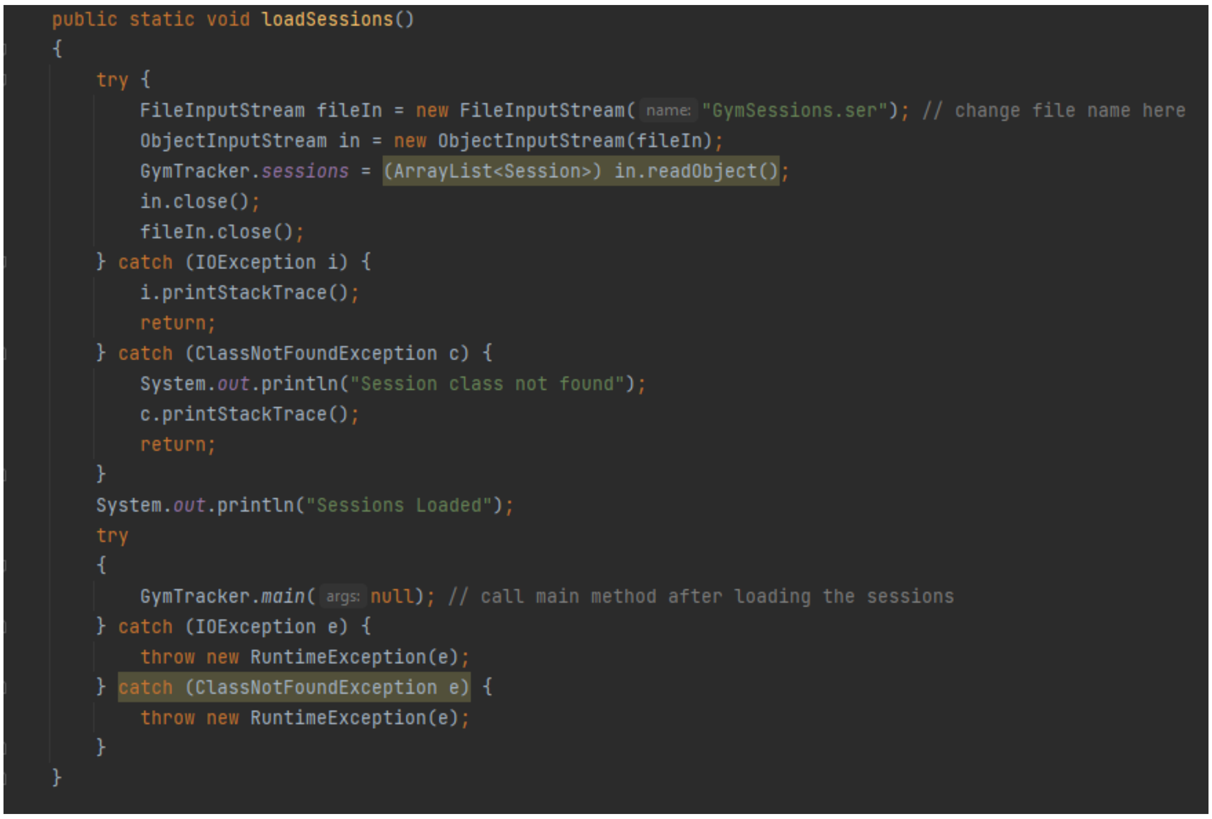
Justification:

Although the user inputs are validated using while loops, when exceptions occur the try catch will allow me to deal with certain errors easily.

Explanation:

By using a Try-Catch block I can act upon certain exceptions as by catching errors in areas such as the loadSessions or saveSessions methods in the Session class I am able to use a Try-Catch block in order to catch multiple exceptions specifically in the loadSessions method.

Example:


```java
public static void loadSessions()
{
    try {
        FileInputStream fileIn = new FileInputStream( name: "GymSessions.ser"); // change file name here
        ObjectInputStream in = new ObjectInputStream(fileIn);
        GymTracker.sessions = (ArrayList<Session>) in.readObject();
        in.close();
        fileIn.close();
    } catch (IOException i) {
        i.printStackTrace();
        return;
    } catch (ClassNotFoundException c) {
        System.out.println("Session class not found");
        c.printStackTrace();
        return;
    }
    System.out.println("Sessions Loaded");
    try
    {
        GymTracker.main( args: null); // call main method after loading the sessions
    } catch (IOException e) {
        throw new RuntimeException(e);
    } catch (ClassNotFoundException e) {
        throw new RuntimeException(e);
    }
}
```

Technique: Dependency
Link to success criteria: - The client must be able to input the name of the exercise, weight/resistance used, number of sets, and number of repetitions. Which should all be stored into different sessions containing multiple exercises.
Source: (Drien Vargas)

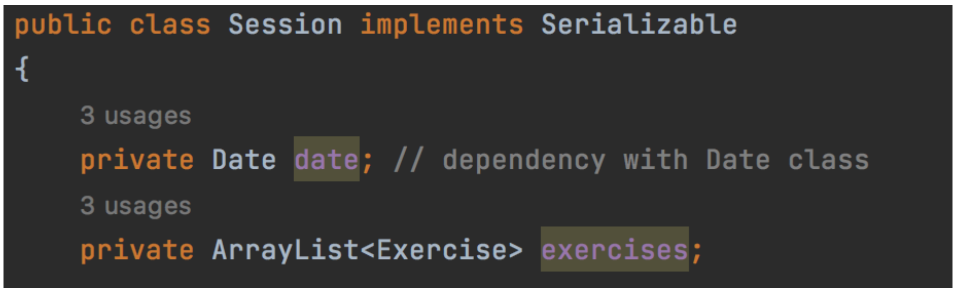
Justification:

By using dependency I am able to link all the classes together so that the code is able to be more efficient, specifically with the Date class and the Session class and the relationship between the 2 classes.

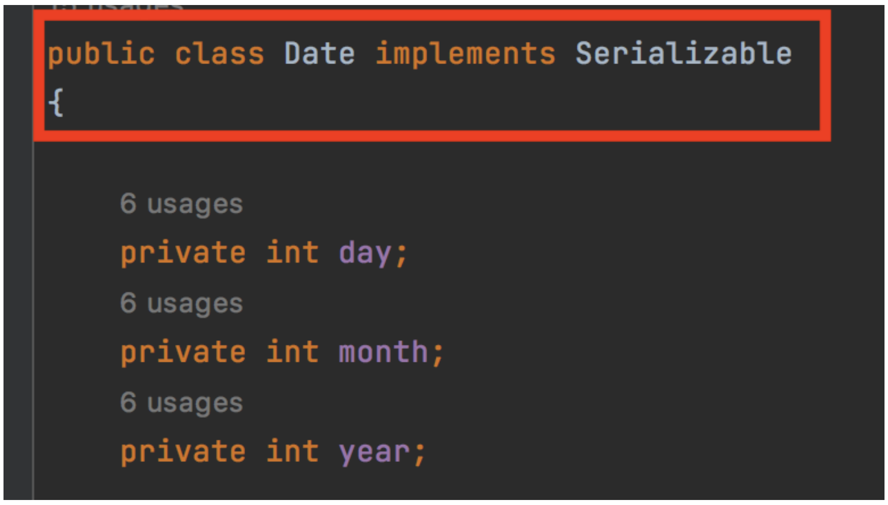
Explanation:

The Sessions class has a dependency class relationship with the Date class, as the controller class uses the Date class in order to create a new data type which is used to differentiate different sessions apart, similar to that of an ID for each session, as one would not typically undergo 2 sessions in a single day.

Example:

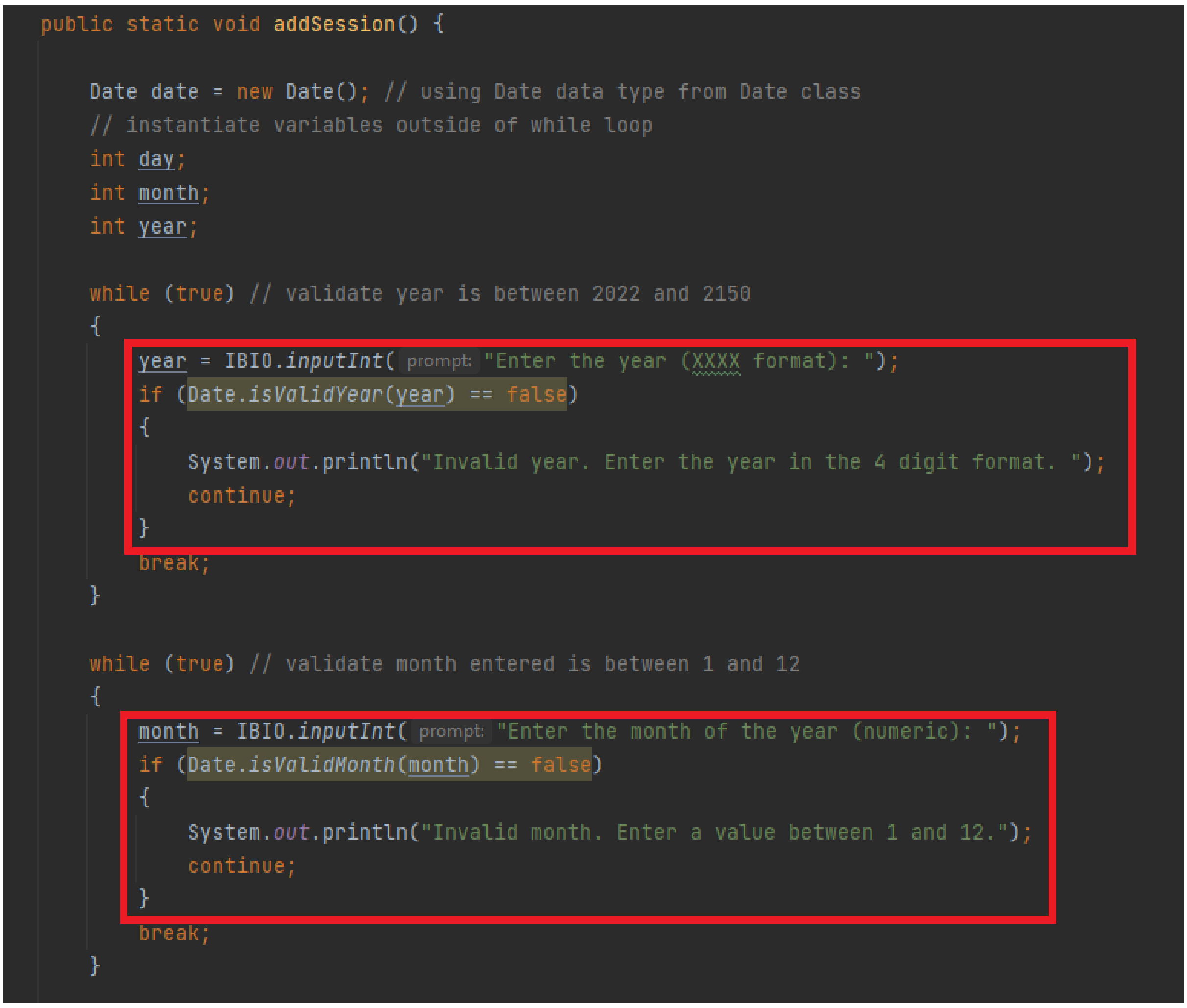


```
public class Session implements Serializable
{
    3 usages
    private Date date; // dependency with Date class
    3 usages
    private ArrayList<Exercise> exercises;
```



```
public class Date implements Serializable
{

    6 usages
    private int day;
    6 usages
    private int month;
    6 usages
    private int year;
```



```
public static void addSession() {

    Date date = new Date(); // using Date data type from Date class
    // instantiate variables outside of while loop
    int day;
    int month;
    int year;

    while (true) // validate year is between 2022 and 2150
    {
        year = IBIO.inputInt( prompt: "Enter the year (XXXX format): ");
        if (Date.isValidYear(year) == false)
        {
            System.out.println("Invalid year. Enter the year in the 4 digit format. ");
            continue;
        }
        break;
    }

    while (true) // validate month entered is between 1 and 12
    {
        month = IBIO.inputInt( prompt: "Enter the month of the year (numeric): ");
        if (Date.isValidMonth(month) == false)
        {
            System.out.println("Invalid month. Enter a value between 1 and 12.");
            continue;
        }
        break;
    }
```

[Word Count: 998]