Object Oriented Programming 2 AL_KCNCM_9_1: 2024 – 25

Assignment

Submission Date: 04/04/2025

Lecturer Name: Dr Sean Kennedy

Student

Philip Herweling

A00326153

A00326153@student.tus.ie

AL_KCNCM_9

Contents

1. Introduction

This report outlines the development of a console-based Fitness Tracker application for the OOP2 module. The aim was to demonstrate a strong understanding of modern Java features by implementing a feature-rich, object-oriented program. The application allows users to log their workouts, track their fitness goals, and view progress using key Java tools like records, sealed classes, streams, and concurrency.

2. Functional and Technical User-Stories

2.1 Lambdas and Streams

As a user, I want the application to filter my workout log by workout type and calculate the total calories burned for each type. This helps me see how effective my cardio or strength workouts are.

Java Features Used:

- Collectors.groupingBy() to group workouts by type
- Collectors.summingDouble() to calculate total calories

```
//Method which returns total cals burned for each workout type
public Map<String, Double> getCaloriesByType() {  no usages
    return workoutLog.stream()
            .collect(Collectors.groupingBy(
                    Workout::type,
                    Collectors.summingDouble(Workout::calsBurned)
            ));
}

//Method which gets a users total calories burned
public double getTotalCaloriesBurned() {  3 usages
    return workoutLog.stream() Stream<Workout>
            .mapToDouble(Workout::calsBurned) DoubleStream
            .sum();
}
```

2.2 Switch Expressions

As a user, I want to receive progress updates based on my personal fitness goal. Using a switch expression lets the system provide specific feedback based on my goal type.

Java Features Used:

- Enhanced switch expressions
- Pattern matching for goal types

```
return switch (goal) {
    case WeightLossGoal weightloss -> {
        double lost = weightloss.getStartWeight() -weight;
        double toLose = weightloss.getStartWeight() - weightloss.getTargetWeight();
        double progress = (lost /toLose) * 100;
        yield String.format("You have lost %.1f kg (%.1f%% of your goal)",lost, progress);
    }//end of weight loss case
```

2.3 Sealed Classes

As a developer, I want to restrict goal types to predefined categories, ensuring the system remains maintainable and secure. Sealed classes allow me to control the hierarchy.

Java Features Used:

- Sealed interfaces and permitted subclasses for FitnessGoal
- Strong control over subclassing

```
public sealed interface FitnessGoal permits WeightLossGoal, StrengthGoal{  5 usages  2 implementations

    String getGoalDescription();  1 usage  2 implementations

}
```

2.4 Date/Time API

As a user, I want to see how many days have passed since my last workout so I can track my consistency.

Java Features Used:

- LocalDateTime to store workout times
- Duration.between() to calculate days since last activity

```java
//Method to work out the days between a users last workout
public long daysSinceLastWorkout() {  1 usage
    if (workoutLog.isEmpty()) return -1;

    Workout lastWorkout = workoutLog.get(workoutLog.size() - 1);
    return Duration.between(lastWorkout.workoutDate(), LocalDateTime.now()).toDays();
}
```

2.5 Generics and Comparator

As a user, I want to see a leaderboard of top-performing users sorted by total calories burned, so I can compare my progress with others.

Java Features Used:

- Generics with List<User>
- Comparator.comparingDouble() for sorting

```java
public List<User> getTopPerformers() {  1 usage
    return users.stream()
            .sorted(Comparator.comparingDouble(User::getTotalCaloriesBurned).reversed())
            .collect(Collectors.toList());
}
```

2.6 Concurrency

As an admin, I want to generate fitness reports for multiple users at the same time to improve performance and efficiency.

Java Features Used:

- ExecutorService and Callable for multithreading

```
public void generateUserReports() {  1 usage
    ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4);

    List<Callable<String>> tasks = users.stream()  Stream<User>
            .map( User user -> (Callable<String>) () -> {
                return String.format(
                        "User: %s\nTotal Workouts: %d\nTotal Calories Burned: %.1f\n",
                        user.getName(),
                        user.getWorkoutLog().size(),
                        user.getTotalCaloriesBurned()
                );
            }) Stream<Callable<...>>
            .toList();

    try {
        List<Future<String>> results = executor.invokeAll(tasks);
```

2.7 Collectors

As a user, I want the system to group my workouts by type and calculate total calories per type, so I can identify the most effective ones.

Java Features Used:

- Collectors.groupingBy()
- Collectors.summingDouble()

```
//Method which returns total cals burned for each workout type
public Map<String, Double> getCaloriesByType() {  no usages
    return workoutLog.stream()
            .collect(Collectors.groupingBy(
                    Workout::type,
                    Collectors.summingDouble(Workout::calsBurned)
            ));
}
```

2.8 Localisation

As a user, I want to view my progress report in my preferred language, making it easier to understand and improving accessibility.

Java Features Used:

- Locale and ResourceBundle for multi-language support

```
    ResourceBundle bundle = ResourceBundle.getBundle( baseName: "messages", locale);

    double lost = weightLoss.getStartWeight() - weight;
    double toLose = weightLoss.getStartWeight() - weightLoss.getTargetWeight();
    double progress = (lost / toLose) * 100;

    String pattern = bundle.getString( key: "progress_message");
    return MessageFormat.format(pattern,
            String.format("%.1f", lost),
            String.format("%.1f", progress));
}
```

2.9 Records

As a developer, I want to use a lightweight, immutable structure for storing workouts, which simplifies data management.

Java Features Used:

- record for the Workout class

```
public record Workout(String type, Duration duration, double calsBurned, LocalDateTime workoutDate) {  10 usages

    //Formatter for workoutDate DD/MM/YYYY HH:mm
    private static final DateTimeFormatter FORMATTER = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");  1 usage

    //Method which returns the formatted date
    public String formattedWorkoutDate() {  1 usage
        return workoutDate.format(FORMATTER);
    }

    //Method to determine if a workout was long or not
    public boolean isLongWorkout(){  no usages
        return duration.toMinutes() > 60;
    }


}
```

2.10 NIO2

As a developer, I want to save and load workout logs to and from a file, so the user's data persists across sessions.

Java Features Used:

- Files.writeString() and Files.readAllLines()
- Path and StandardOpenOption for file handling

```
//Method to save workouts to a file
public void saveWorkoutsToFile() {  3 usages
    try {
        String content = workoutLog.stream()  Stream<Workout>
                .map( Workout w -> w.type() + "," + w.duration().toMinutes() + "," + w.calsBurned() +
                        "," + w.workoutDate().format(FORMATTER))  Stream<String>
                .collect(Collectors.joining( delimiter: "\n"));
        Files.writeString(FILE_PATH, content, StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
    } catch (IOException e) {
        System.out.println("Error saving workout log to file: " + e.getMessage());
    }
}
```

3. Implementation

The implementation of the Fitness Tracker application was centred around clean object-oriented principles and showcasing Java 21+ features. Below are detailed code snippets and explanations from key parts of the system:

3.1 Workout Record

```
public record Workout(String type, Duration duration, double calsBurned, LocalDateTime workoutDate) {  10 usages

    //Formatter for workoutDate DD/MM/YYYY HH:mm
    private static final DateTimeFormatter FORMATTER = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm");  1 usage

    //Method which returns the formatted date
    public String formattedWorkoutDate() {  1 usage
        return workoutDate.format(FORMATTER);
    }

    //Method to determine if a workout was long or not
    public boolean isLongWorkout(){  no usages
        return duration.toMinutes() > 60;
    }


}
```

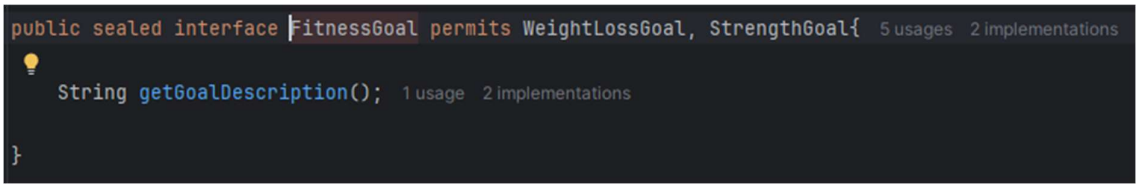
The Workout record stores the key details of a workout, As can be seen in the image above a workout is made up of a type, duration, calories burned and the date and time it was logged.

Using a record here makes the class immutable and removed the need for boilerplate code such as constructors and getters. The static formatter I used to convert the workout dates into a more readable template. It uses the DateTimeFormatter class from the Date/Time API. (Baeldung Java Records) (Baeldung: Java 8 Date and Time)

I then added a method to return a formatted date. It uses the formatter mentioned above to return the date as a string which is very useful for displaying logs and also saving to files.

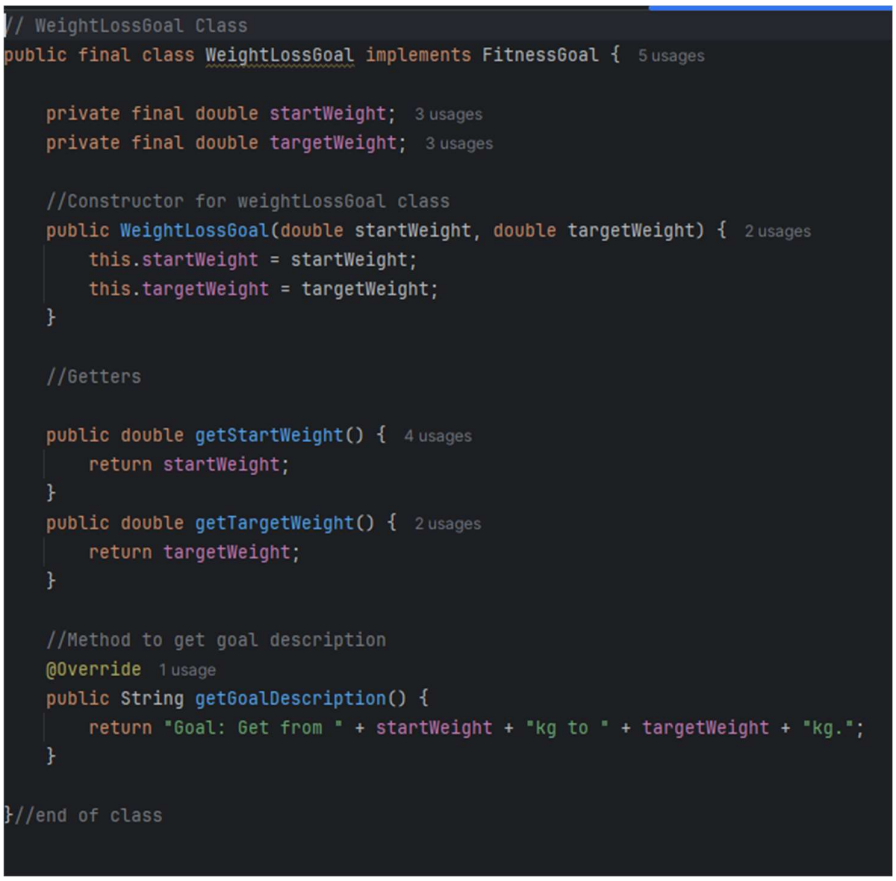
Lastly I added a small utility method which works out if a workout was a long one or not. I decided to set it so any workout over 1 hour is considered long. I thought it would be a helpful feature for highlighting more intense sessions.

3.2 FitnessGoal (Sealed Interface)



As you can see, I used the sealed keyword for this interface. This means only specified classes can implement it . This enforces a known set of fitness goals, improving control over the class hierarchy. The permits keyword restricts the interface to only two known subclasses, WeightLossGoal and StrengthGoal. (GeeksForGeeks)

3.3 WeightLossGoal and StrengthGoal



```java
// WeightLossGoal Class
public final class WeightLossGoal implements FitnessGoal {  5 usages

    private final double startWeight;  3 usages
    private final double targetWeight;  3 usages

    //Constructor for weightLossGoal class
    public WeightLossGoal(double startWeight, double targetWeight) {  2 usages
        this.startWeight = startWeight;
        this.targetWeight = targetWeight;
    }

    //Getters

    public double getStartWeight() {  4 usages
        return startWeight;
    }
    public double getTargetWeight() {  2 usages
        return targetWeight;
    }

    //Method to get goal description
    @Override  1 usage
    public String getGoalDescription() {
        return "Goal: Get from " + startWeight + "kg to " + targetWeight + "kg.";
    }

}//end of class
```

WeightLossGoal is one of the two permitted implementations of the FitnessGoal sealed interface mentioned above. It defines a simple weight loss goal, storing a user's starting and target weights. I also declared the class as final meaning it can't be subclassed which supports the sealed design and keeps the class behaviour predictable.

The constructor then requires both the starting and target weight for a user. These fields are final meaning that once they are set they can't be changed. This makes the object immutable.

I then added the getter methods for the two fields which provide access to the goal data. These are used in the progress calculations and report generation in the User class of the application.

Lastly then the method getGoalDescription overrides the abstract method defined in FitnessGoal. It returns a description of the weight loss goal, which is useful for displaying in the UI or in reports. (GeeksForGeeks)

```java
// StrengthGoal class
public final class StrengthGoal implements FitnessGoal {  3 usages

    private final int targetReps;  3 usages
    private final double targetWeightLifted;  3 usages

    //Constructor for StrengthGoal
    public StrengthGoal(int targetReps, double targetWeightLifted) {  1 usage
        this.targetReps = targetReps;
        this.targetWeightLifted = targetWeightLifted;
    }

    //Getters
    public int getTargetReps() {  no usages
        return targetReps;
    }

    public double getTargetWeightLifted() {  no usages
        return targetWeightLifted;
    }

    @Override  1 usage
    public String getGoalDescription() {
        return "Goal: Lift " + targetWeightLifted + "kg for " + targetReps + " reps.";
    }

}//end of class
```

The StrengthGoal class is implemented in the same way as WeightLossGoal, following the sealed interface structure. It stores a target number of reps and a target weight to lift and returns a description using the getGoalDescription() method. The structure and purpose are nearly identical, so it wasn't necessary to break this down in detail again.

3.4 Switch expression or Progress Evaluation

```java
return switch (goal) {
    case WeightLossGoal weightloss -> {
        double lost = weightloss.getStartWeight() -weight;
        double toLose = weightloss.getStartWeight() - weightloss.getTargetWeight();
        double progress = (lost /toLose) * 100;
        yield String.format("You have lost %.1f kg (%.1f%% of your goal)",lost, progress);
    }//end of weight loss case
```

This modern switch expression handles progress calculation based on the user's goal type. It uses pattern matching to both check the type of the goal and cast it in a single line. It then calculates how much weight the user has lost and what percentage of there goal they have achieved. Lastly it uses string.format to format the result with 1 decimal place. (Oracle)

3.5 Grouping and summing workouts

```
//Method which returns total cals burned for each workout type
public Map<String, Double> getCaloriesByType() {  no usages
    return workoutLog.stream()
            .collect(Collectors.groupingBy(
                    Workout::type,
                    Collectors.summingDouble(Workout::calsBurned)
            ));
}
```

This stream operation processes the workout log by converting it into a stream and grouping workouts by their type (e.g., Cardio, Football). For each workout type, it calculates the total calories burned using Collectors.summingDouble(). The result is returned as a Map<String, Double>, where the key represents the workout type and the value is the total calories burned for that type

3.6 Total Calories Burned

```
//Method which gets a users total calories burned
public double getTotalCaloriesBurned() {  3 usages
    return workoutLog.stream()  Stream<Workout>
            .mapToDouble(Workout::calsBurned)  DoubleStream
            .sum();
}
```

This method uses the Stream API and the .mapToDouble and .sum to go through all of a user's workouts and return the total sum of calories they have burned.

3.7 Concurrency with ExecutureService

```
public void generateUserReports() {  1 usage
    ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4);

    List<Callable<String>> tasks = users.stream() Stream<User>
            .map( User user -> (Callable<String>) () -> {
                return String.format(
                        "User: %s\nTotal Workouts: %d\nTotal Calories Burned: %.1f\n",
                        user.getName(),
                        user.getWorkoutLog().size(),
                        user.getTotalCaloriesBurned()
                );
            }) Stream<Callable<...>>
            .toList();

    try {
        List<Future<String>> results = executor.invokeAll(tasks);

        System.out.println("\n User Reports Generated Concurrently:");
        for (Future<String> result : results) {
            System.out.println("-------------");
            System.out.println(result.get());
        }
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    } finally {
        executor.shutdown();
    }
}
```

This method generates a summary report for each user. It creates multiple reports at the same time using multiple threads. The first line of the method creates a pool of 4 threads meaning that it can run up to 4 tasks at once. Next, I create a task for each user, each of these tasks returns a string report for one user. It uses String.format to neatly include a user's name, number of workouts, and total calories burned.

Next then in the try block I run all the tasks at once. The Future<String> holds the results of each task i.e. the report and this can then be retrieved when its done.

The third part of this method is then printing out the reports. So I loop through the results first (for each result in results) I then print out the report using .get().

I then shutdown the executor when it's finished to clear up system resources. (Java Code Geeks)

3.8 File Saving and Loading (NIO2)

```
//Method to save workouts to a file
public void saveWorkoutsToFile() {  3 usages
    try {
        String content = workoutLog.stream()  Stream<Workout>
                .map( Workout w -> w.type() + "," + w.duration().toMinutes() + "," + w.calsBurned() +
                        "," + w.workoutDate().format(FORMATTER))  Stream<String>
                .collect(Collectors.joining( delimiter: "\n"));
        Files.writeString(FILE_PATH, content, StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
    } catch (IOException e) {
        System.out.println("Error saving workout log to file: " + e.getMessage());
    }
}
```

The saveWorkoutToFile method converts each workout into a formatted string, joins them with a new line character and then writes each result into a file.

How it works, is firstly I turn workouts into a stream. I then mapped each workout into a comma-separated String. I then join all the workouts into a single string called content separated by new lines.

Lastly, I write the string to a file located at FILE_PATH. The CREATE is there in case the File doesn't exist and if that is the case it creates the file. TRUNCATE_EXISTING overwrites existing content in a file. (Baeldung: Java NIO)

```
//Method to load workouts from file
public void loadWorkoutsFromFile() {  4 usages
    if (!Files.exists(FILE_PATH)) return;

    try {
        List<String> lines = Files.readAllLines(FILE_PATH);
        for (String line : lines) {
            String[] parts = line.split( regex: ",");
            if (parts.length == 4) {
                String type = parts[0];
                Duration duration = Duration.ofMinutes(Long.parseLong(parts[1]));
                double calories = Double.parseDouble(parts[2]);
                LocalDateTime date = LocalDateTime.parse(parts[3], FORMATTER);
                workoutLog.add(new Workout(type, duration, calories, date));
            }
        }
    } catch (IOException e) {
        System.out.println("Error loading workouts: " + e.getMessage());
    }
}
```

The loadWorkoutsFromFIle method checks firstly if the file exists, If it does exist I reads the file line by line parsing each line to rebuild a workout and add it to the list.

So the first line in this method checks if the file exists if it doesn't it exist it skips loading.

The first line inside the try block reads all the lines from the file into a list.

Then I have a for which loops through each line in the now stored in the list. Next each line is split commas. The parts are then converted to the right data types and finally a new workout object is created and added to the workout log. (Baeldung: Java NIO)

3.9 Localisation with ResourceBundle

```
//Method to show localisation
public String getLocalizedProgress(Locale locale) {  1 usage
    if (!(goal instanceof WeightLossGoal weightLoss)) {
        return "Progress reporting not supported for this goal.";
    }

    ResourceBundle bundle = ResourceBundle.getBundle( baseName: "messages", locale);

    double lost = weightLoss.getStartWeight() - weight;
    double toLose = weightLoss.getStartWeight() - weightLoss.getTargetWeight();
    double progress = (lost / toLose) * 100;

    String pattern = bundle.getString( key: "progress_message");
    return MessageFormat.format(pattern,
            String.format("%.1f", lost),
            String.format("%.1f", progress));
}
```

getLocalizedProgress method generates a progress message for the user in there preferred language, In this case I choose to implement just English and German in future works more languages would be added. It supports localisation using .properties files and javas built in internationalisation tools.

So firstly the if statement at the start of the method. This checks if a users goal is a weightLossGoal. If it's not a weight loss goal then it returns a fallback message. The if statement uses pattern matching with instance of to check and cast in the one line.

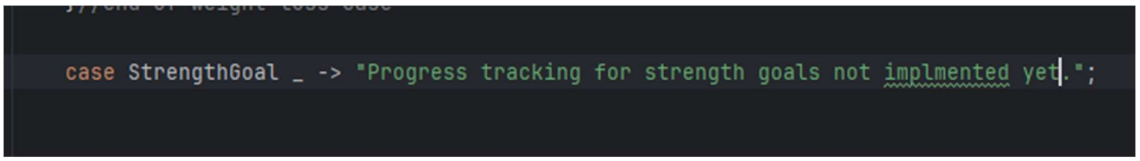
Next in the method a resource bundle for the selected locale is loaded. So either Locale.English or Locale.German. this then access the file message_en.properties or message_de.properties. These files contain loaclised templates like this:

```
progress_message=You have lost {0} kg. ({1}% of your goal)
```

Next I then calculate how much weight the user has lost, Then I work out how far they are towards reaching there set goal, as a percentage.

After that I then I get the message template from the properties file for the selected language. Lastly then I replace the {0} and {1} in the message with their actual values i.e. how much weight they lost and their progress in percent. I also format these values to 1 decimal point, and this then returns a clean, localised String. (JetBrains Academy)

3.10 Java 22 unnamed pattern:



I also used a Java 22 feature, unnamed pattern variables by including _ in a switch expression to handle unused pattern matches. This helped simplify the code and avoid unnecessary variable names. (OpenJDK JEP 443)

4. Evaluation

4.1 Code Design:

The overall design of the application followed clean object-oriented principles. Using a record for Workout made the data structure lightweight and simple to work with. The separation of goal types into a sealed interface with specific implementations helped enforce structure and reduced the chance of logic errors. The class responsibilities were well-defined, making the application easy to maintain.

4.2 Java Features Used:

The project allowed me to apply a wide range of Java features from the brief. I found sealed interfaces and switch expressions with pattern matching particularly useful, as they made goal handling more intuitive and safer. Using streams simplified many of the data processing tasks, such as grouping workouts and calculating totals. Localisation was a new feature for me, and I learned how Java handles different language files dynamically.

4.3 Challenges Faced:

I had a few difficulties getting file reading and writing to work properly using NIO2, especially when parsing strings into dates and durations. Setting up localisation also required careful file placement and folder configuration in IntelliJ. Understanding concurrency with ExecutorService took some trial and error, but once working, it really improved the application's performance and scalability.

4.4 Testing and Usage:

I built a console menu to manually test each feature in a clean, repeatable way. This allowed me to demonstrate the system interactively, logging workouts, generating reports, and switching languages live. Although there are no automated tests, the structure made it easy to test individual methods.

5. Conclusion

This project gave me the opportunity to build a complete object-oriented application using modern Java features. It helped reinforce many of the concepts that were taught in class, such as using records, sealed interfaces, and switch expressions to write clean and structured code. I also deepened my understanding of streams, collectors, and file handling through practical implementation. Working on features like localisation and concurrency pushed me beyond the basics and gave me a better appreciation for how Java handles more advanced functionality. Overall, I'm proud of how the application turned out. It meets all the requirements from the brief and is easy to test and demonstrate using the console menu. I feel more confident in both my Java skills and my ability to apply object-oriented design principles effectively, I also think the features and principles I have learned from doing this project will help me reach my goal of getting my first job as an software engineer.

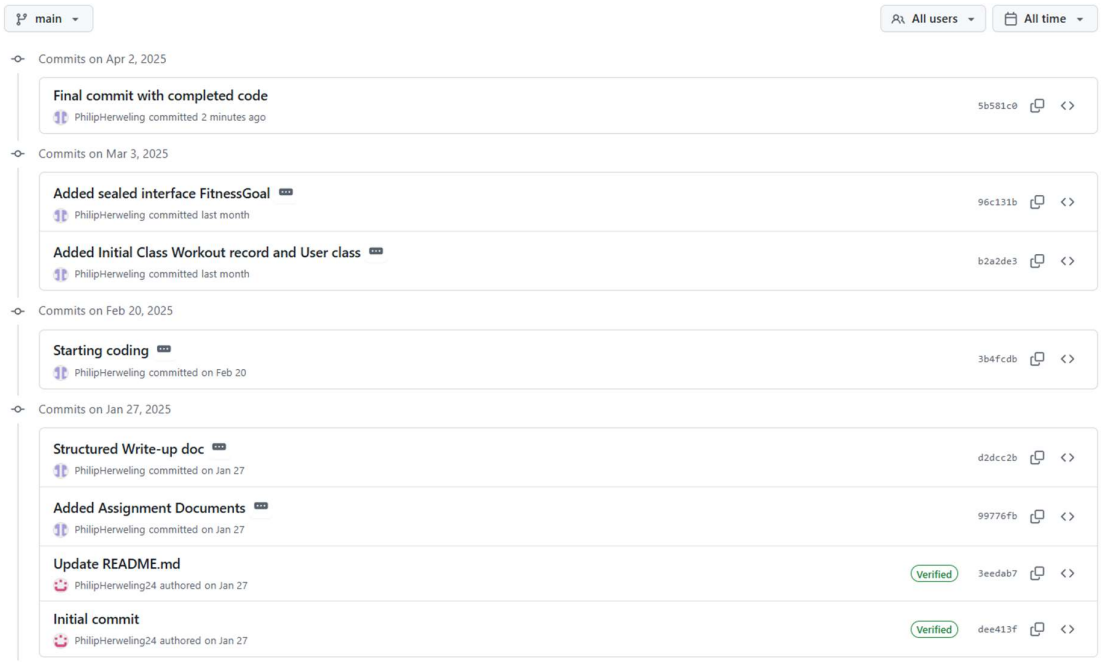
5.1 Future Improvements:

If I had more time, I would have liked to add a basic graphical user interface to improve usability. I would also replace the text file storage with a lightweight database like SQLite for better data handling. Adding unit tests would also be a good next step to improve test coverage and ensure reliability as the application grows.

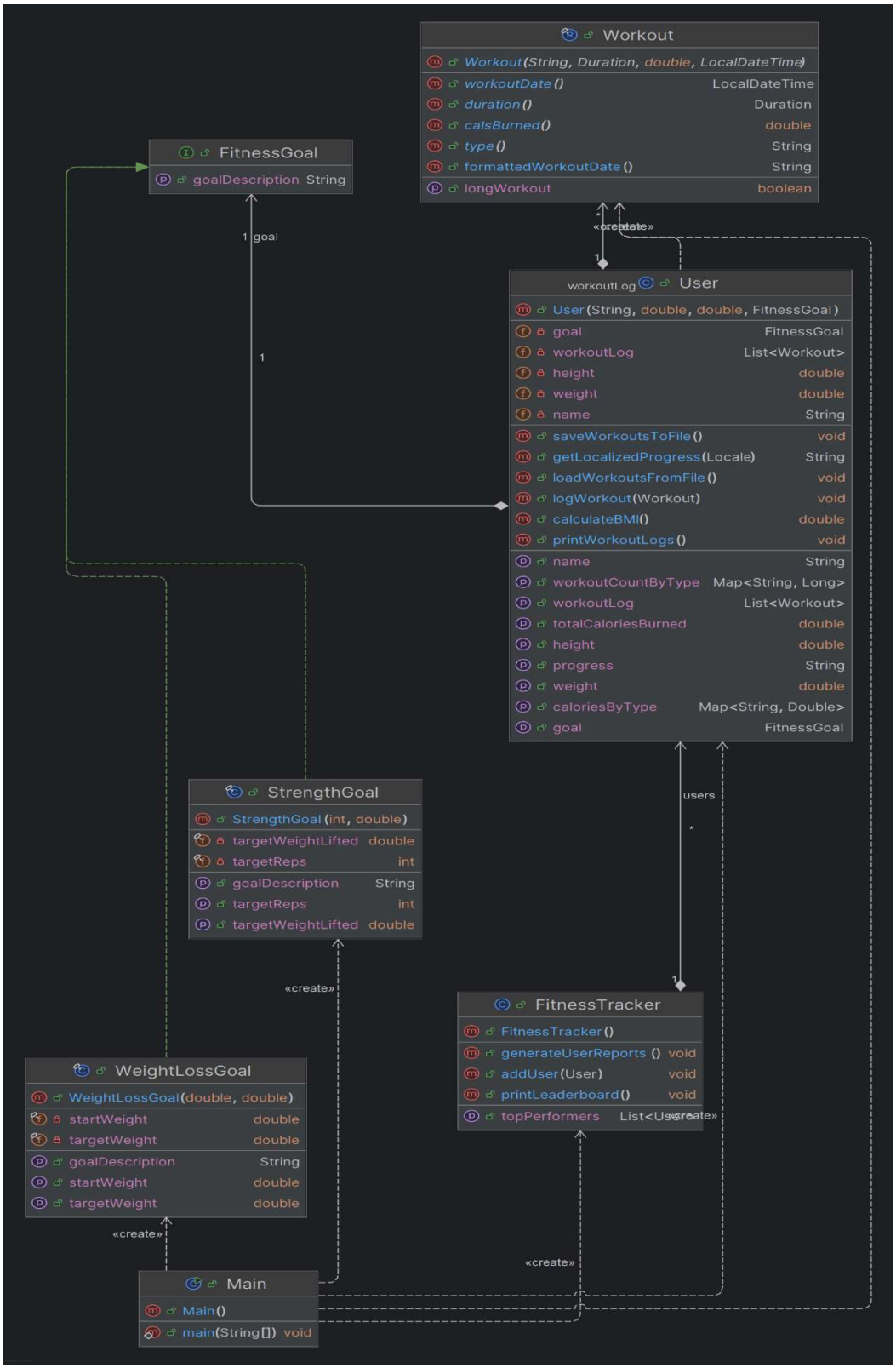
5.2 Feature Implementation Overview:

#	Feature	Implemented In
1	Lambdas	generateUserRepors() / getTopPerformers()
2	Streams – Terminal Operations	getTotalCaloriesBurned()/ getTopPerformers()
3	Streams - collect() (Collectors.toMap, groupingBy, partitioningBy)	getCaloriesByType() - Collectors.groupingBy(), summingDouble()
4	Streams - Intermediate Operations (filter, map, sorted, etc.)	Stream.map() in file saving, potentially filter() if used elsewhere
5	Switch Expressions and Pattern Matching	getProgress() - switch (goal) with pattern matching
6	Sealed Classes and Interfaces	FitnessGoal interface (sealed), WeightLossGoal, StrengthGoal
7	Date/Time API	Workout.formattedWorkoutDate(), daysSinceLastWorkout()
8	Records	Workout record (type, duration, calsBurned, workoutDate)
9	Collections/Generics (e.g. Comparator.comparing)	getTopPerformers() - Comparator.comparingDouble()
10	Concurrency (ExecutorService + Callable)	generateUserReports() - ExecutorService + List<Callable>
11	NIO2	saveWorkoutsToFile(), loadWorkoutsFromFile() - Files.readAllLines(), writeString()
12	Localisation	getLocalizedProgress() - ResourceBundle, Locale
13	Java 22 unnamed Variables	getProgress() – case strength goal – "String"

5.3 Git Repo Log:



6. UML Class Diagram



7. References

1.) Baeldung. Java Record Keyword. Available at: https://www.baeldung.com/java-record-keyword (Accessed: 1 April 2025).

2.) GeeksForGeeks. Sealed Classes in Java. Available at: https://www.geeksforgeeks.org/sealed-classes-in-java/ (Accessed: 1 April 2025).

3.) Oracle. Pattern Matching for switch. Available at: https://docs.oracle.com/en/java/javase/21/language/pattern-matching-switch.html (Accessed: 1 April 2025).

4.) Baeldung. Guide to Java 8 Streams. Available at: https://www.baeldung.com/java-8-streams (Accessed: 1 April 2025).

5.) Java Code Geeks. Java ExecutorService Example. Available at: https://www.javacodegeeks.com/2020/10/java-executorservice-example.html (Accessed: 1 April 2025).

6.) Baeldung. Java NIO2 File API. Available at: https://www.baeldung.com/java-nio-2-file-api (Accessed: 1 April 2025).

7.) JetBrains Academy. Localisation in Java. Available at: https://hyperskill.org/learn/step/11273 (Accessed: 1 April 2025).

8.) OpenJDK. Unnamed Patterns and Variables (JEP 443). Available at: https://openjdk.org/jeps/443 (Accessed: 1 April 2025).

9.) Baeldung. Java 8 Date and Time API. Available at: https://www.baeldung.com/java-8-date-time-intro (Accessed: 1 April 2025).