JAVA ACADEMY - XIDERAL MONTERREY, N.L

EXERCISE - WEEK 2 JAVA STREAMS

Made by:
Luis Miguel Sánchez Flores

August 24, 2024

PROBLEM:

At a local soccer league in Monterrey, the Avengers team finds itself in a difficult situation: **In their most recent season, the team were just points away from relegation, and had one of the worst goal differentials in the club's history.**

After an emergency meeting with the board of directors, the decision was made: a new coach was needed, alongside an experienced coaching staff full of skilled collaborators.

After a while, the team managed to sign one of the most outstanding managers of the league thanks to his offensive tactics and being able to train his players to an extraordinary level.
To improve the new project's chances of success, **the club also hired scouts to try and help the new coach into signing players who might fit in the new team**, with the ultimate goal of getting back to the top of the league like in the club's glory days.

Introduced in Java version 8, Streams are a sequence of functional-style operations used to process, manipulate and transform data collections in a concise and declarative manner. Streams are very useful to **perform a wide range of operations on an extensive set of elements**, such as filtering, mapping or reducing elements, in a pipeline fashion, conducting **lazy evaluations** in which the data is either created, processed or transformed when necessary, enabling it to handle even infinite amounts of data efficiently.

Streams work with **lambda expressions** that provide a clear and short way to create instances of **functional interfaces** which leverages the power of functional programming to turn functions as a first-class citizen and describe what actions the Stream must do, without specifying how it should perform them.
All in all, Streams are a powerful tool to process and manipulate large amounts of data through a simplified, readable and efficient code that is easy to work with.

Streams are composed of three parts:

- **Source**: Defines the elements to be processed by the Stream. Only one source can be used for the Stream. Common methods for creating the source are **Stream.empty()**, **Stream.of() and Stream.generate().**

- **Intermediate operations**: Operations that work on the elements and transform the Stream into another. As such, intermediate operations can be chained together to form a pipeline of operations. Any number of intermediate operations can be applied. Some of the intermediate operations are **filter(), map(), sorted() and limit().**

- **Terminal operation**: Operation in charge of producing the end result of the Stream. Only one terminal operation can be used, and is located at the end of the pipeline. Common operations include **count(), max(), min(), forEach() and collect()**

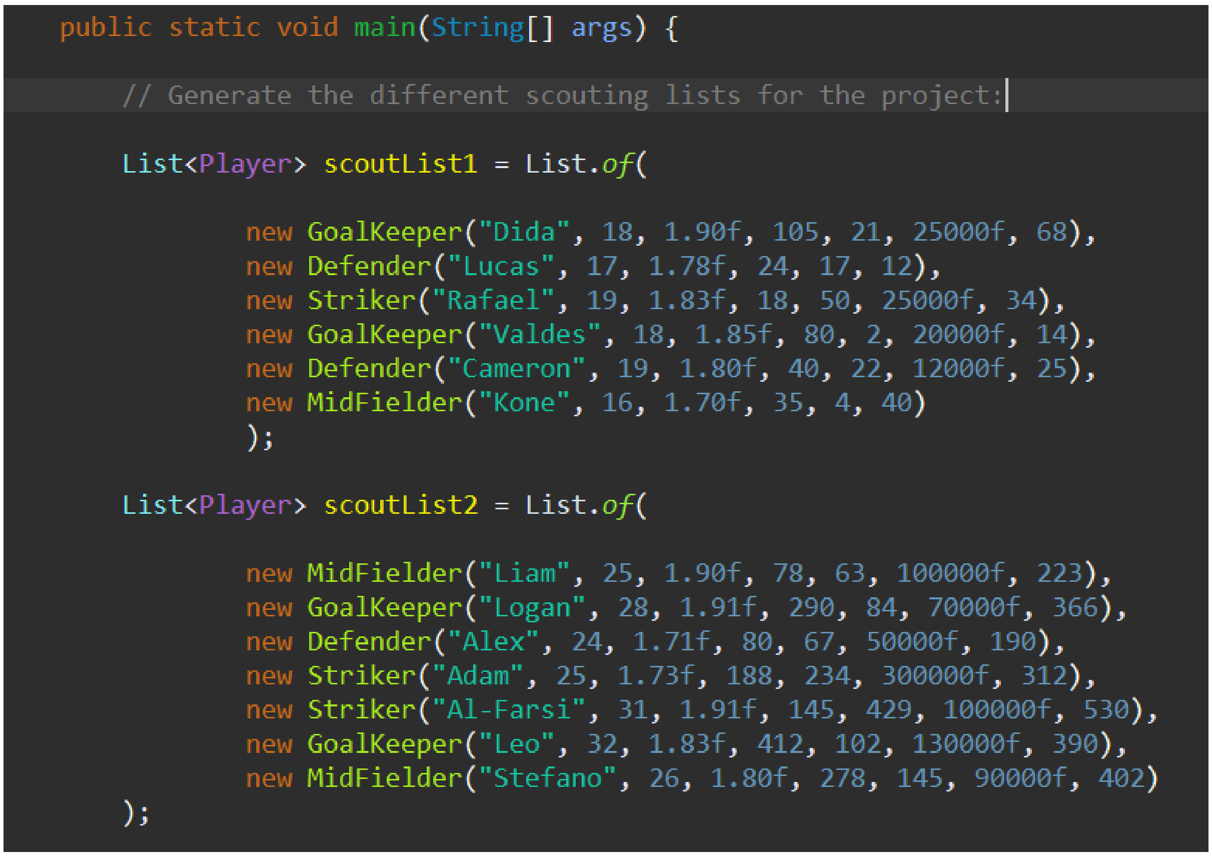
SCENARIO #1 - LIST OF OPTIMAL PLAYERS:

For this case, Streams can be used to provide the coach with a optimal list of potential players for the new team's project: After receiving the reports from each of the recently hired scouts, **the new coach wishes to deduce each of the scouting list into a single list of ideal players for the team, making sure that each player meets the following criteria:**

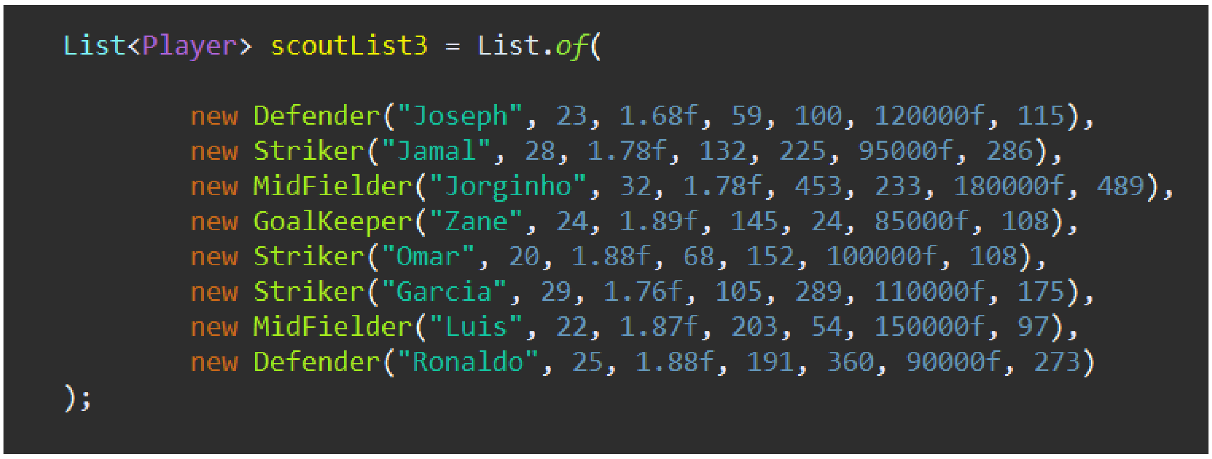
- Must have played at least 30 games, and have a performance rate of at least 0.75 in their respective position.
- Prioritize the younger players, but leave out the youngest player off the list.
- The list should only include the top 10 ideal players for the team.

Based on the established criteria, the following lines of code were developed, including the appropriate Stream operations to achieve the optimal player list desired by the coach:

First, a series of List objects are created to simulate the scouting lists of the team, each of them containing a number of Player objects as shown below:

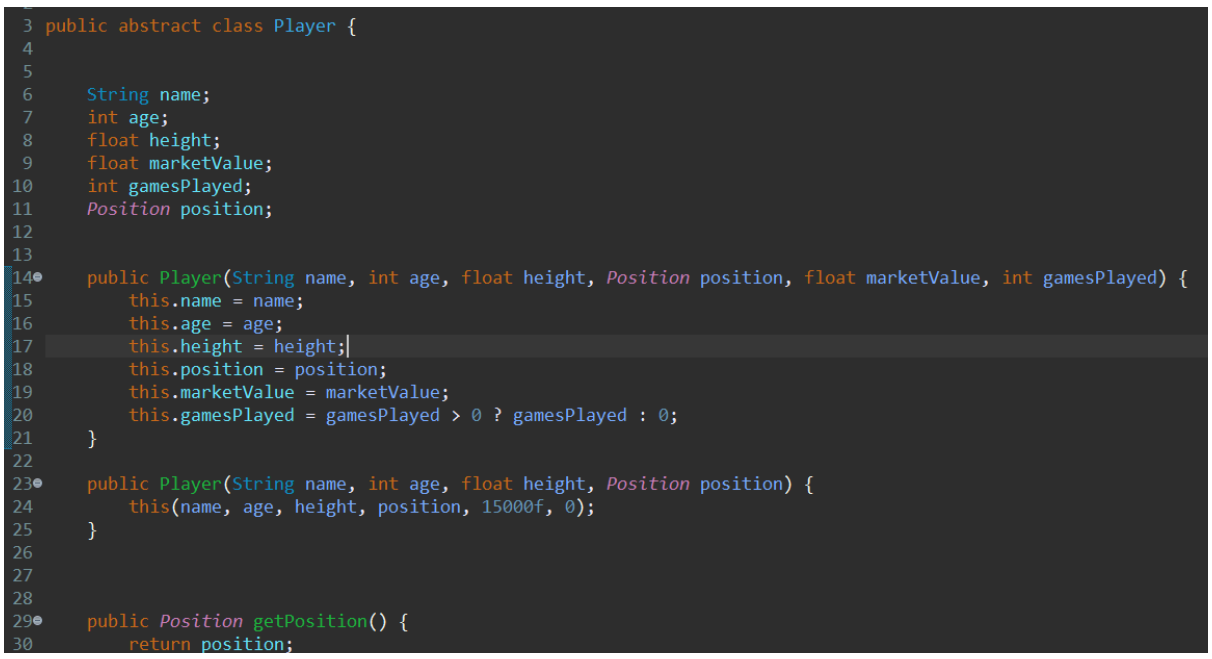


```java
public static void main(String[] args) {

    // Generate the different scouting lists for the project:

    List<Player> scoutList1 = List.of(

            new GoalKeeper("Dida", 18, 1.90f, 105, 21, 25000f, 68),
            new Defender("Lucas", 17, 1.78f, 24, 17, 12),
            new Striker("Rafael", 19, 1.83f, 18, 50, 25000f, 34),
            new GoalKeeper("Valdes", 18, 1.85f, 80, 2, 20000f, 14),
            new Defender("Cameron", 19, 1.80f, 40, 22, 12000f, 25),
            new MidFielder("Kone", 16, 1.70f, 35, 4, 40)
            );

    List<Player> scoutList2 = List.of(

            new MidFielder("Liam", 25, 1.90f, 78, 63, 100000f, 223),
            new GoalKeeper("Logan", 28, 1.91f, 290, 84, 70000f, 366),
            new Defender("Alex", 24, 1.71f, 80, 67, 50000f, 190),
            new Striker("Adam", 25, 1.73f, 188, 234, 300000f, 312),
            new Striker("Al-Farsi", 31, 1.91f, 145, 429, 100000f, 530),
            new GoalKeeper("Leo", 32, 1.83f, 412, 102, 130000f, 390),
            new MidFielder("Stefano", 26, 1.80f, 278, 145, 90000f, 402)
    );
```



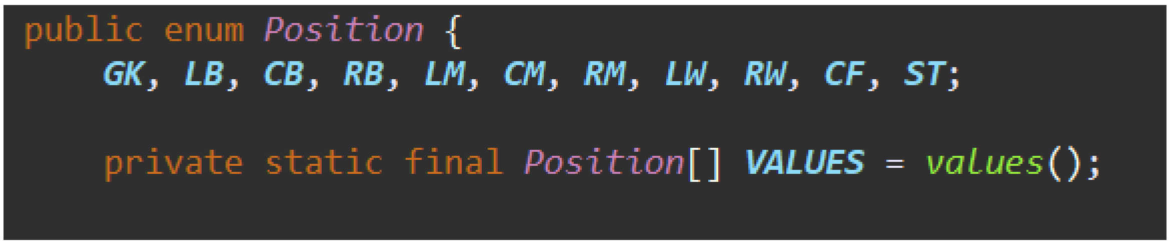
```java
List<Player> scoutList3 = List.of(

        new Defender("Joseph", 23, 1.68f, 59, 100, 120000f, 115),
        new Striker("Jamal", 28, 1.78f, 132, 225, 95000f, 286),
        new MidFielder("Jorginho", 32, 1.78f, 453, 233, 180000f, 489),
        new GoalKeeper("Zane", 24, 1.89f, 145, 24, 85000f, 108),
        new Striker("Omar", 20, 1.88f, 68, 152, 100000f, 108),
        new Striker("Garcia", 29, 1.76f, 105, 289, 110000f, 175),
        new MidFielder("Luis", 22, 1.87f, 203, 54, 150000f, 97),
        new Defender("Ronaldo", 25, 1.88f, 191, 360, 90000f, 273)
);
```

Each element is a class inherited from **Player**, an abstract class that establishes the common properties among the players of a soccer team, such as their name, age, market value, number of matches played and their position:



```java
public abstract class Player {


    String name;
    int age;
    float height;
    float marketValue;
    int gamesPlayed;
    Position position;


    public Player(String name, int age, float height, Position position, float marketValue, int gamesPlayed) {
        this.name = name;
        this.age = age;
        this.height = height;
        this.position = position;
        this.marketValue = marketValue;
        this.gamesPlayed = gamesPlayed > 0 ? gamesPlayed : 0;
    }

    public Player(String name, int age, float height, Position position) {
        this(name, age, height, position, 15000f, 0);
    }



    public Position getPosition() {
        return position;
```

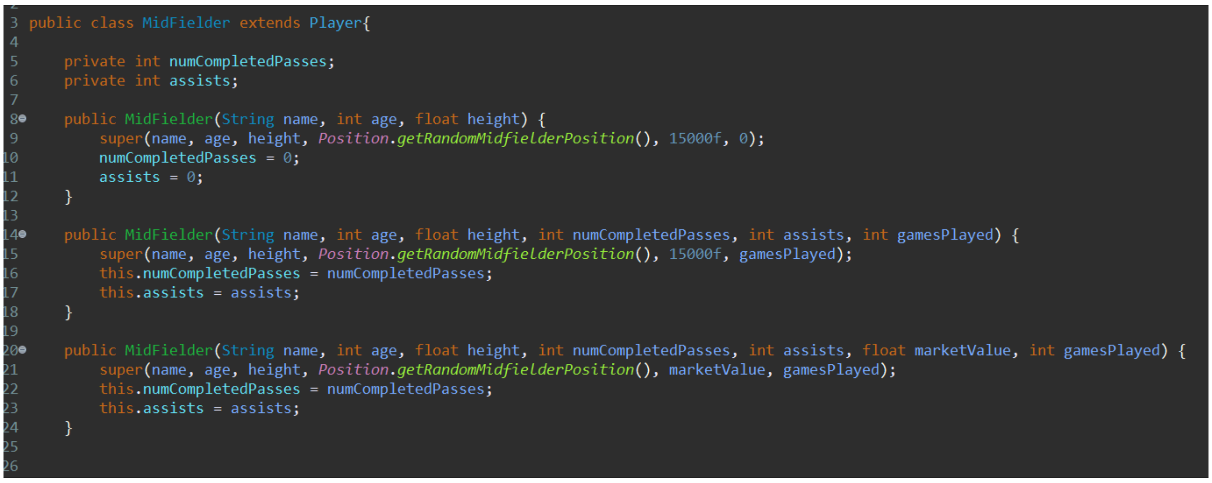
The position variable of a Player is defined by the Enum “**Position**” that includes common roles in a soccer team:



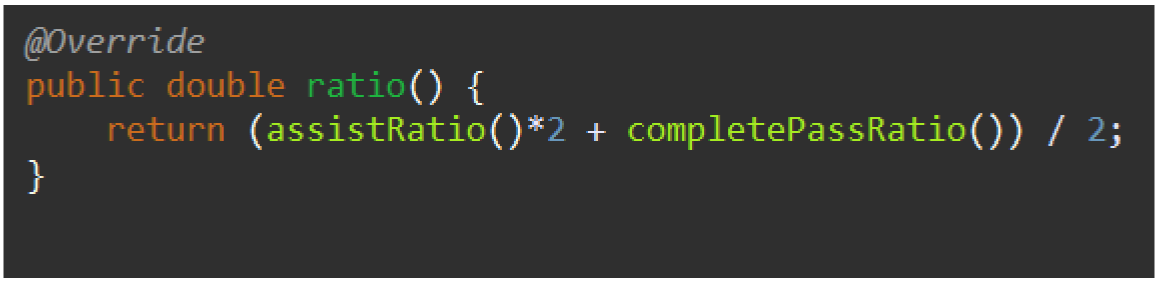
```java
public enum Position {
    GK, LB, CB, RB, LM, CM, RM, LW, RW, CF, ST;

    private static final Position[] VALUES = values();
```

Each concrete class of Player (**GoalKeeper, MidFielder, Defender and Striker**) contains exclusive properties that help calculate their performance rate in their respective positions. In the case of the MidFielder, Defender and Striker classes, a random position is assigned to each object using static functions that the Position enum provides with:

Example of properties and performance rate from Midfielder Class:

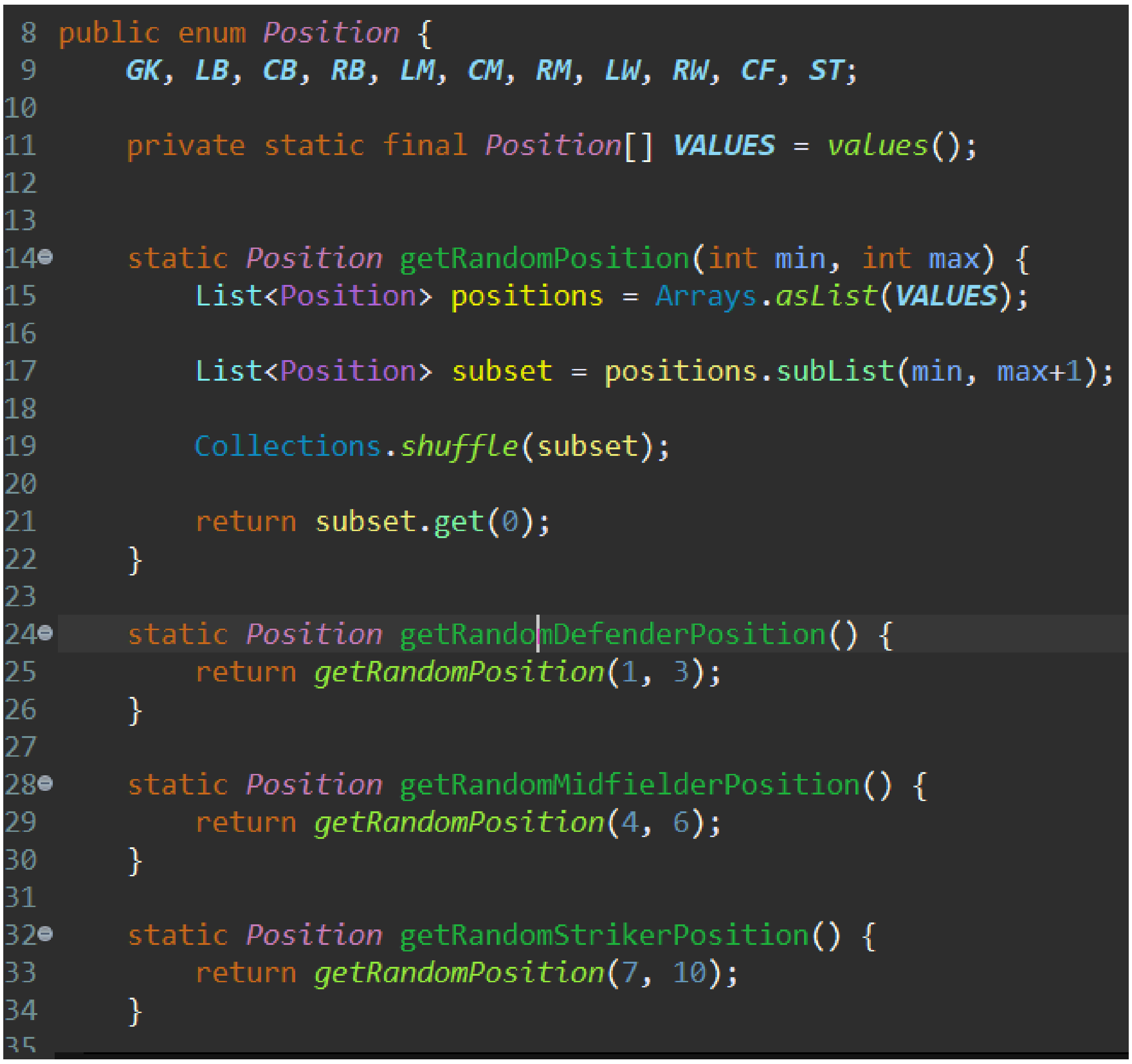


```java
public class MidFielder extends Player{

    private int numCompletedPasses;
    private int assists;

    public MidFielder(String name, int age, float height) {
        super(name, age, height, Position.getRandomMidfielderPosition(), 15000f, 0);
        numCompletedPasses = 0;
        assists = 0;
    }

    public MidFielder(String name, int age, float height, int numCompletedPasses, int assists, int gamesPlayed) {
        super(name, age, height, Position.getRandomMidfielderPosition(), 15000f, gamesPlayed);
        this.numCompletedPasses = numCompletedPasses;
        this.assists = assists;
    }

    public MidFielder(String name, int age, float height, int numCompletedPasses, int assists, float marketValue, int gamesPlayed) {
        super(name, age, height, Position.getRandomMidfielderPosition(), marketValue, gamesPlayed);
        this.numCompletedPasses = numCompletedPasses;
        this.assists = assists;
    }


```



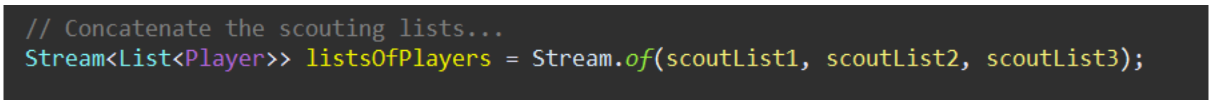
```java
@Override
public double ratio() {
    return (assistRatio()*2 + completePassRatio()) / 2;
}
```

Static functions from Position to randomize defenders, midfielders and strikers.



```java
public enum Position {
    GK, LB, CB, RB, LM, CM, RM, LW, RW, CF, ST;

    private static final Position[] VALUES = values();


    static Position getRandomPosition(int min, int max) {
        List<Position> positions = Arrays.asList(VALUES);

        List<Position> subset = positions.subList(min, max+1);

        Collections.shuffle(subset);

        return subset.get(0);
    }

    static Position getRandomDefenderPosition() {
        return getRandomPosition(1, 3);
    }

    static Position getRandomMidfielderPosition() {
        return getRandomPosition(4, 6);
    }

    static Position getRandomStrikerPosition() {
        return getRandomPosition(7, 10);
    }
```

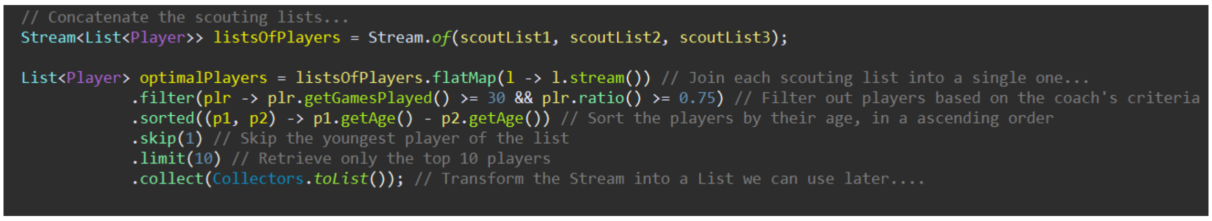
The Stream starts with the use of the **Stream.of()** method, concatenating each of the scouting lists:



```java
// Concatenate the scouting lists...
Stream<List<Player>> listsOfPlayers = Stream.of(scoutList1, scoutList2, scoutList3);
```

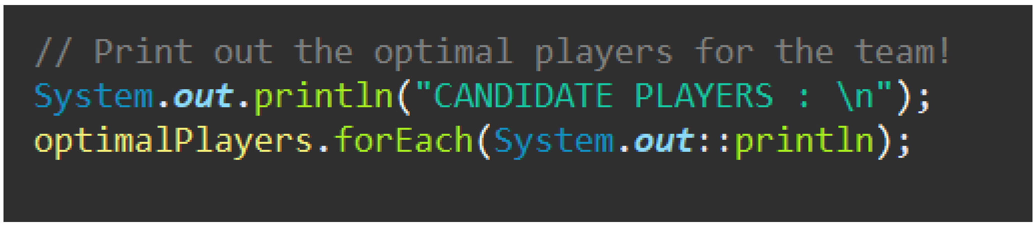
We then proceed to carry out the intermediate operations of the Stream, starting out with the **flatMap()** operation to “flatten” the scouting lists into a single list. The players are then filtered according to the coach's criteria using the **filter()** method, removing those who have played less than 30 games and have a performance below 0.75. The list of players is then sorted through the **sorted()** method that uses the age variable of the players to place the younger ones at the top of the list. Given the coach doesn’t want to consider the youngest player of the list, it can easily be removed with the **skip(1)** method that only discards the first element of the stream. The last intermediate operation called is the **limit(10)** operation, which retrieves only the first 10 elements (players) of the Stream.

The Stream is ended with the terminal operation that is **collect()** that ultimately transforms the stream into a Collection type (in this case, the Stream gets transformed into a List object containing the optimal players we were looking for):

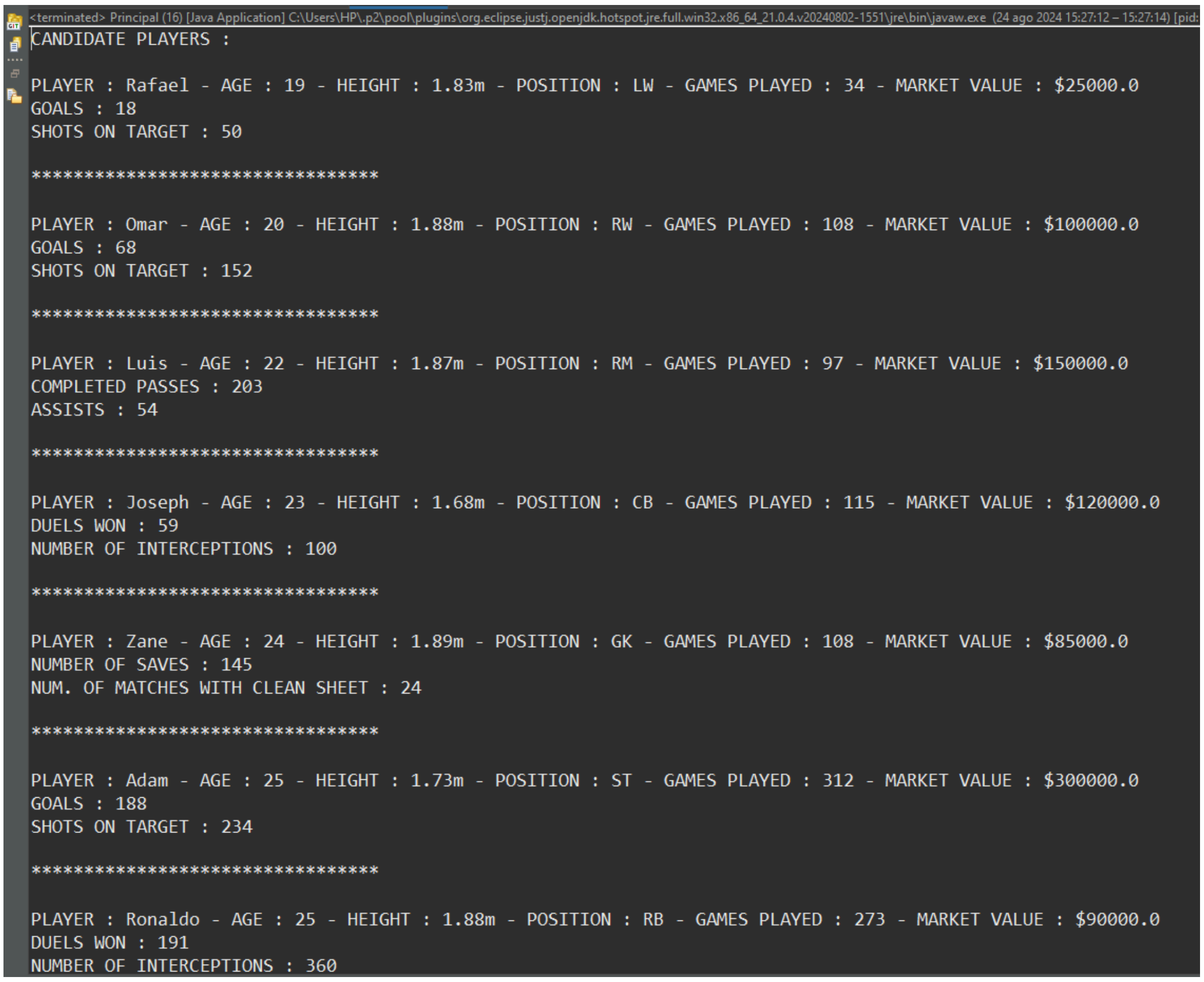


```java
// Concatenate the scouting lists...
Stream<List<Player>> listsOfPlayers = Stream.of(scoutList1, scoutList2, scoutList3);

List<Player> optimalPlayers = listsOfPlayers.flatMap(l -> l.stream()) // Join each scouting list into a single one...
            .filter(plr -> plr.getGamesPlayed() >= 30 && plr.ratio() >= 0.75) // Filter out players based on the coach's criteria
            .sorted((p1, p2) -> p1.getAge() - p2.getAge()) // Sort the players by their age, in a ascending order
            .skip(1) // Skip the youngest player of the list
            .limit(10) // Retrieve only the top 10 players
            .collect(Collectors.toList()); // Transform the Stream into a List we can use later....
```

With a few lines of code, We were able to transform the different scouting lists into an optimal list of players who are candidates for the new team project, which is printed below:



```java
// Print out the optimal players for the team!
System.out.println("CANDIDATE PLAYERS : \n");
optimalPlayers.forEach(System.out::println);
```



```
<terminated> Principal (16) [Java Application] C:\Users\HP\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.4.v20240802-1551\jre\bin\javaw.exe  (24 ago 2024 15:27:12 – 15:27:14) [pid:
CANDIDATE PLAYERS :

PLAYER : Rafael - AGE : 19 - HEIGHT : 1.83m - POSITION : LW - GAMES PLAYED : 34 - MARKET VALUE : $25000.0
GOALS : 18
SHOTS ON TARGET : 50

*********************************

PLAYER : Omar - AGE : 20 - HEIGHT : 1.88m - POSITION : RW - GAMES PLAYED : 108 - MARKET VALUE : $100000.0
GOALS : 68
SHOTS ON TARGET : 152

*********************************

PLAYER : Luis - AGE : 22 - HEIGHT : 1.87m - POSITION : RM - GAMES PLAYED : 97 - MARKET VALUE : $150000.0
COMPLETED PASSES : 203
ASSISTS : 54

*********************************

PLAYER : Joseph - AGE : 23 - HEIGHT : 1.68m - POSITION : CB - GAMES PLAYED : 115 - MARKET VALUE : $120000.0
DUELS WON : 59
NUMBER OF INTERCEPTIONS : 100

*********************************

PLAYER : Zane - AGE : 24 - HEIGHT : 1.89m - POSITION : GK - GAMES PLAYED : 108 - MARKET VALUE : $85000.0
NUMBER OF SAVES : 145
NUM. OF MATCHES WITH CLEAN SHEET : 24

*********************************

PLAYER : Adam - AGE : 25 - HEIGHT : 1.73m - POSITION : ST - GAMES PLAYED : 312 - MARKET VALUE : $300000.0
GOALS : 188
SHOTS ON TARGET : 234

*********************************

PLAYER : Ronaldo - AGE : 25 - HEIGHT : 1.88m - POSITION : RB - GAMES PLAYED : 273 - MARKET VALUE : $90000.0
DUELS WON : 191
NUMBER OF INTERCEPTIONS : 360
```

SCENARIO #2 - THE DIRECTIVE'S CONDITIONS:

As the coach was about to start contacting the player candidates for the project, the team directives arrived to deliver some news: While they're pleased with the coach's ambitions to return the team into the top of the league, **they decided to start small with the project**, and instead work with a short list of candidates first. Therefore, the **directives established the following conditions as the priorities for the time being:**

- **The list must only contain strikers.**
- The strikers must have a market value less or equal than $110,000
- Do not consider the **oldest striker on the list**

- Sort the players by their **number of goals achieved**
- For now, limit the list by the **top 3 matching strikers.**

After the discussion, the coach now wishes for the list of the matching strikers to be drawn from the optimal list of his desired players, in order to begin the contact and signing process with the corresponding strikers.

Given this task, we can reapply Streams to easily find the strikers that meet both the coach's criteria and the directive's conditions.
To do this, the Stream is started off from **the list of players obtained in the previous scenario**, as the List interface offers the **stream()** method to directly transform the List object into a Stream of Players we would like to work on.

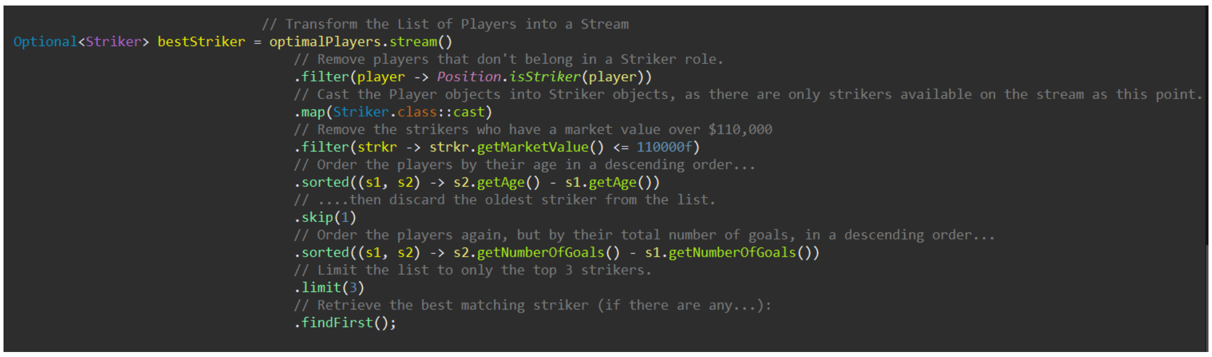
With the Stream obtained, we filter out the players that **do not belong in a Striker role**, using a Predicate in which the **isStriker()** method from Position is applied to obtain the boolean value if the corresponding player has an attacking role.

As the Stream now only contains striker players, we can easily cast them from a Player to a Striker object using the **map()** operation, transforming from a Stream of type Player into a Stream of type Striker, which will help us later on.

Another **filter** operation is applied to **eliminate the strikers that have a market value bigger than $110,000,** as established by the team's directives.

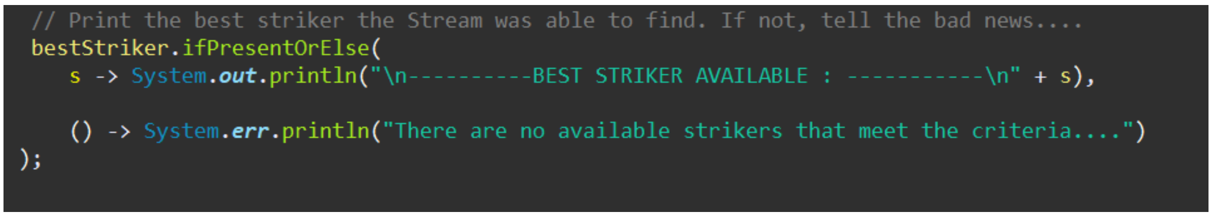
A **sorted()** operation is used to order the list of strikers by their age, but this time in a descending order, as to eliminate the oldest striker of the stream by applying the **skip()** operation. Thanks to the Striker class casting done beforehand, we can apply the **getNumberOfGoals()** from the strikers to carry out the **sort operation** in which the players are rearranged by their total number of goals in a descending order.

The last intermediate operation is the **limit()** method, to only retrieve the top 3 strikers from the list.
The terminal operation used for this Stream is the **findFirst()** method, which returns an Optional value that either contains the first element of the stream (the best matching striker of the list) or an empty value if the stream is empty after carrying out the pipeline (meaning that there are no strikers from the scouting lists that match the conditions of both the coach and the team directives). The Optional is stored in the **bestStriker** variable:

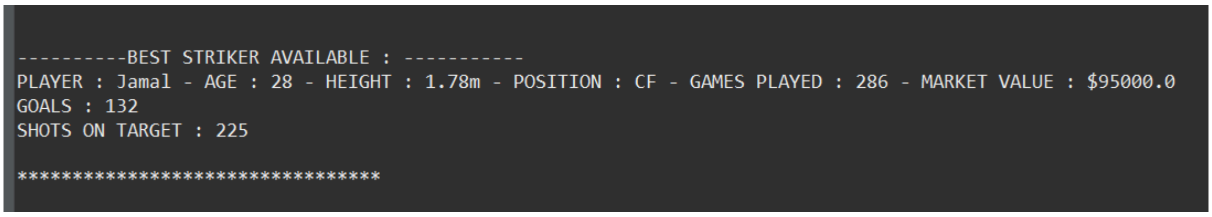


```
                                // Transform the List of Players into a Stream
Optional<Striker> bestStriker = optimalPlayers.stream()
                                    // Remove players that don't belong in a Striker role.
                                    .filter(player -> Position.isStriker(player))
                                    // Cast the Player objects into Striker objects, as there are only strikers available on the stream as this point.
                                    .map(Striker.class::cast)
                                    // Remove the strikers who have a market value over $110,000
                                    .filter(strkr -> strkr.getMarketValue() <= 110000f)
                                    // Order the players by their age in a descending order...
                                    .sorted((s1, s2) -> s2.getAge() - s1.getAge())
                                    // ....then discard the oldest striker from the list.
                                    .skip(1)
                                    // Order the players again, but by their total number of goals, in a descending order...
                                    .sorted((s1, s2) -> s2.getNumberOfGoals() - s1.getNumberOfGoals())
                                    // Limit the list to only the top 3 strikers.
                                    .limit(3)
                                    // Retrieve the best matching striker (if there are any...):
                                    .findFirst();
```

With the end result at hand, we apply the **ifPresentOrElse** method of the Optional variable to print out the corresponding messages in case if the best matching striker is present or not. If so, it prints out the best striker found, if not, it prints that no striker meeting the criteria has been found:



```
 // Print the best striker the Stream was able to find. If not, tell the bad news....
 bestStriker.ifPresentOrElse(
    s -> System.out.println("\n----------BEST STRIKER AVAILABLE : -----------\n" + s),

    () -> System.err.println("There are no available strikers that meet the criteria....")
);
```



```
----------BEST STRIKER AVAILABLE : -----------
PLAYER : Jamal - AGE : 28 - HEIGHT : 1.78m - POSITION : CF - GAMES PLAYED : 286 - MARKET VALUE : $95000.0
GOALS : 132
SHOTS ON TARGET : 225

*********************************
```

As with the previous scenario, with a couple of lines of code, it was enough to find the best matching striker for the new team’s project, starting from the previous list of optimal players, thanks to the power of Java Streams that allows us to work with a large set of elements in a simple but effective way.