Lab 1: Implementing a Class

Tània Pazos and Aniol Petit

Pompeu Fabra University

21414: Object Oriented Programming

October 22, 2023

Lab 1: Implementing a Class

The following paper describes the implementation and subsequent verification of the Player and Team classes in Java. For the program to be considered effective, it should be able to (i) define a Player class, (ii) define a Team class, (iii) implement a Country class, and (iv) define a test class with a main method, where instances of the different classes are created and the various methods are implemented to ensure that there are no errors.

Regarding the Player class, the constructor should correctly set the initial values of all attributes and use the supplied Country class to represent a player's nationality. Figure 1 shows the design of the Player class, where the method `update` correctly updates the statistics of a player as a result of playing a match, and the method `printStats` prints the current statistics of a player after several matches.

As for the Team class, again, the constructor should correctly set the initial values of all attributes, and the Country class should be used to represent the country of a team. Figure 2 shows the design of the Team class, where the method `addPlayer` checks that the gender of the player matches the gender designation of the team, the method `playMatch` correctly updates the statistics of a team as a result of playing a match - taking as arguments the number of goals scored for the team and the number of goals scored against the team-, and the method `printStats` prints the current statistics of a team after several matches.

Figure 1

Player class design

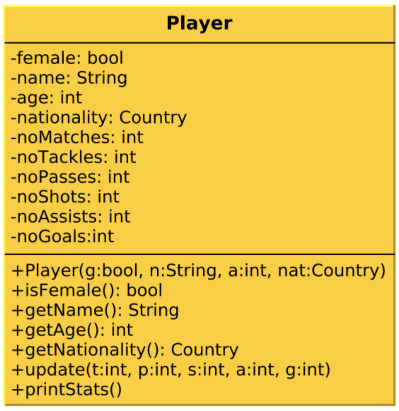
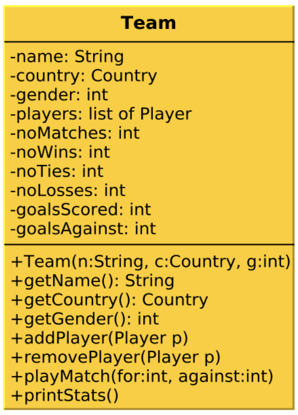


Figure 2

Team class design



Solution Implementation

Player class

The attributes definition followed the design proposed in Figure 1, except for the `female` attribute: in order to make the choice of attributes for both the Player and the Team classes consistent, this attribute was changed to `gender`, represented by an `int`.

As for the methods, the constructor `Player` correctly sets the value for attributes `gender`, `name`, `age`, and `nationality`: the rest of the attributes -`noMatches`, `noTackles`, `noPasses`, `noShots`, `noAssists`, and `noGoals`-, all performance-related, are not included as arguments of the constructor because it does not make sense to provide default values for these statistics. Instead, they are automatically initialized to 0 in the attribute definition. The methods getGender, getName, getAge, and getNationality are included in the program, as all attributes in the Player class have private visibility. The constructor update simply adds one to the number of matches played and, given `noTackles`, `noPasses`, `noShots`, `noAssists`, and `noGoals,` adds the new passed values to the accumulated values. Finally, the method `printStats` simply prints the name of a player followed by all their performance-related statistics.

Team class

Similarly, the attributes definition of the `Team` class sticks to the suggested design in Figure 2. However, it is worth pointing out that the list `players` is implemented as a `LinkedList,` where each element is of type Player, as the following line of code depicts.

```
private LinkedList<Player> players = new LinkedList<>();
```

Again, the constructor `Team` initializes the non-performance-related attributes `name`, `country`, and `gender` with the values passed as parameters; the methods `getName`, `getCountry`, and `getGender` provide access to the values of `name`, `country`, and `gender` -since they are all private attributes. The method `addPlayer` checks, given a `Player,` passed as a parameter, that the player’s gender matches the team’s gender (0 representing male and 1 representing female). If this is the case, the player gets added to the team; otherwise, an error message is displayed. The method `removePlayer` removes a specific player from the team. The method `playMatch` updates the team's statistics after playing a match, awarding 3 points for a win, 1 point for a tie, and no points for a loss, and correctly updating the number of matches played, the number of goals scored, and the number of goals conceded. The method `printPlayers` simply iterates through the list of players and appends the name of each player to the `StringBuilder playerNames` -separated by a comma and a blank space-. Then, it checks whether there are names in the `StringBuilder`; if this is the case, it removes the last two characters of the `StringBuilder` -the last comma and blank space- and prints the names of the players. Otherwise, a message is displayed indicating that there are no players on the team. Finally, the method `printStats` shows several statistics of the team, including the name of the players forming that team, and the number of matches, wins, ties, losses, points, goals scored and goals conceded.

Possible Alternative Solutions

Before defining the attributes for both the `Player` and the `Team` classes, two different data types to express gender were discussed: `int` and `bool`. Ultimately, we chose to express it as an `int`, since doing so allows for the potential incorporation of gender as a non-binary value.

Conclusion

After writing the main method to see how our solution worked, we could see that the outputs were the expected ones in almost all cases: players were added to teams only if both of their genders matched, matches were correctly played and the statistics properly updated for both the teams and players, and players could be efficiently removed from a team.

One problem we encountered was that, even if a team played some games, the `noMatches` statistics from the players were not updated, and when printing player statistics we would always get 0 matches played. That was because we had not added the count of matches played in the `update` method of the `Player` class, and thus we were not getting the expected result.

Once that was fixed, the implementation of the classes was all correct, receiving the expected output for every possible situation.