Lab Session 3: Implementing inheritance and polymorphism
24292-Object Oriented Programming

1 Introduction

We are going to implement part of the football application that we designed in Seminar 3. The lab session consists in implementing the Java classes that are part of the application, as well as the relations that exist between these classes. In particular, several classes are related via inheritance. Recall that we use the keyword `extends` to inherit from another class.

On the next page you will find a sample design for the football application, which you can use instead of your own design.

2 Implementing the design

To implement all classes from the new design, it is a good idea to reuse the code from Lab session 2. However, it is necessary to *redefine* some classes from the previous program. In software development, reorganizing code is known as *refactoring*.

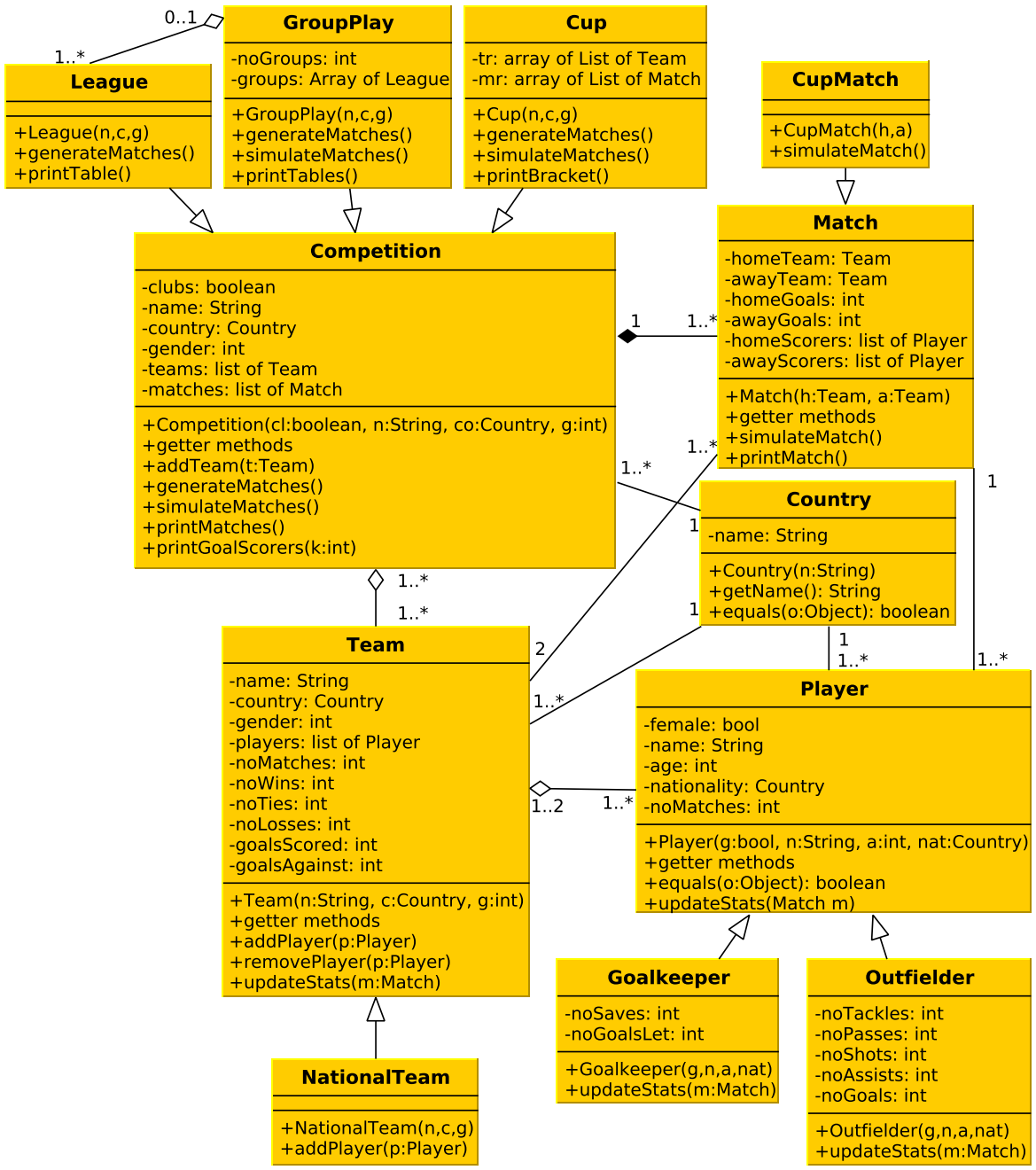
Remember to frequently test your code by creating instances and calling methods from the main method, compiling and executing the code to make sure that it works.

In this lab session we will still not implement the method `printTable` of the League and GroupPlay classes, the method `printBracket` of the Cup class, and the method `printGoalScorers` of the Competition class; this will instead be the topic of the next lab session.

2.1 Implementing the class NationalTeam

The class NationalTeam represents national teams by extending the existing Team class. The only difference is that when attempting to add a player to the team, the code should check that the player has the correct nationality. Hence it is necessary to override the method `addPlayer` of Team.

To check if two countries are equal, it is a good idea to override the method `equals` that Country inherits from Object, similar to the `equals` method you


GroupPlay
-noGroups: int
-groups: Array of League
+GroupPlay(n,c,g)
+generateMatches()
+simulateMatches()
+printTables()
Cup
-tr: array of List of Team
-mr: array of List of Match
+Cup(n,c,g)
+generateMatches()
+simulateMatches()
+printBracket()
CupMatch
+CupMatch(h,a)
+simulateMatch()
0..1
1..*
League
+League(n,c,g)
+generateMatches()
+printTable()
Competition
-clubs: boolean
-name: String
-country: Country
-gender: int
-teams: list of Team
-matches: list of Match
+Competition(cl:boolean, n:String, co:Country, g:int)
+getter methods
+addTeam(t:Team)
+generateMatches()
+simulateMatches()
+printMatches()
+printGoalScorers(k:int)
Match
-homeTeam: Team
-awayTeam: Team
-homeGoals: int
-awayGoals: int
-homeScorers: list of Player
-awayScorers: list of Player
+Match(h:Team, a:Team)
+getter methods
+simulateMatch()
+printMatch()
1
1..*
1..*
1..*
1
1
Country
-name: String
+Country(n:String)
+getName(): String
+equals(o:Object): boolean
1..*
1..*
1
1
1..*
Team
-name: String
-country: Country
-gender: int
-players: list of Player
-noMatches: int
-noWins: int
-noTies: int
-noLosses: int
-goalsScored: int
-goalsAgainst: int
+Team(n:String, c:Country, g:int)
+getter methods
+addPlayer(p:Player)
+removePlayer(p:Player)
+updateStats(m:Match)
2
1..*
1..2
1..*
Player
-female: bool
-name: String
-age: int
-nationality: Country
-noMatches: int
+Player(g:bool, n:String, a:int, nat:Country)
+getter methods
+equals(o:Object): boolean
+updateStats(Match m)
Goalkeeper
-noSaves: int
-noGoalsLet: int
+Goalkeeper(g,n,a,nat)
+updateStats(m:Match)
Outfielder
-noTackles: int
-noPasses: int
-noShots: int
-noAssists: int
-noGoals: int
+Outfielder(g,n,a,nat)
+updateStats(m:Match)
NationalTeam
+NationalTeam(n,c,g)
+addPlayer(p:Player)

have seen in theory. Two countries are equal if they have the same name.

2.2 Implementing the class CupMatch

The class CupMatch represents cup matches by extending the existing Match class. The only difference is that when simulating a match, the match is not allowed to end in a tie. Hence in the case of a tie, the simulation has to continue (e.g. extra time, penalties) until there is a winner. To do so it is necessary to override the method `simulateMatch` of Match. Make sure that you generate goal scorers also for the extra goals scored.

2.3 Implementing the classes Goalkeeper and Outfielder

The two classes Goalkeeper and Outfielder represent the corresponding type of player, by extending the Player class. In doing so, the statistics that was previously kept in the Player class should be *moved* to the Outfielder class, since only this type of player has the associated statistics. Goalkeepers should instead define their own statistics. Otherwise, the Player class remains the same as before.

2.4 Implementing competitions

The class Competition can be defined by reusing attributes and methods from the previous League class. Only some methods (such as `generateMatches` and `printTable`) should remain in League (which now inherits from Competition). For now, the method `generateMatches` can be left empty in Competition, since the idea is always to use one of the subclasses. Unlike before, the Competition class needs a way to distinguish between club competitions and international competitions (which is done in the sample design using the Boolean attribute `clubs`). Note that the method `addTeam` has to take into account the type of competition, i.e. only allow club teams in club competitions or national teams in international competitions.

The League class should be very similar to before, only that it now inherits from Competition, and contains the method `generateMatches` that specifically generates the matches for a league.

The Cup class should generate and simulate matches for multiple rounds, until there is only one winner left. To do so it is a good idea to maintain arrays `tr`, storing the list of teams of each round, and `mr`, storing the list of matches of each round. In the first round, the list of teams includes all participating teams. It is a good idea to *randomize* the teams of the first round (e.g. using the method `Collections.shuffle`). In subsequent rounds, the list of teams are those that won their matches in their previous round (plus any team that was not paired with another team, in the case of an odd number of teams).

Finally, the GroupPlay class can be implemented by maintaining an array of League, such that each group is an instance of league. Again, it is a good

idea to *randomize* the teams that are assigned to each league. The methods of League can then be used to generate and simulate matches.

3 Documentation

Apart from the source code, you should also hand in a document that outlines the solution of the problem. To elaborate the document you can use the following guidelines regarding the content:

1. An introduction where the problem is described. For example, what should the program do? Which classes do you have to define? Which methods do you have to implement for these classes?

2. A description of possible alternative solutions that were discussed, and a description of the chosen solution and the reason for choosing this solution rather than others. It is also a good idea to mention the related theoretical concepts of object-oriented programming that were applied as part of the solution.

3. A conclusion that describes how well the solution worked in practice, i.e. did the tests show that the classes were correctly implemented? You can also mention any difficulties during the implementation as well as any doubts you might have had.

The source code and documentation should all be uploaded to a directory `Lab3` of your Git repository **prior to the next lab session**.