Homework 4 – Inheritance and Polymorphism

Authors: Michael, Nicolas, Andrew, Anupama, Lucas, Seo Hyun

Problem Description

This assignment will test your knowledge of abstract classes, components of a good class, the Object class, inheritance, overriding, polymorphism, and dynamic binding.

The World Cup in Qatar is fast approaching! So, you and your friends are planning on running a little friendly competition on who can best predict the results of each match. However, you know that you are not the best at predicting the results of matches. So, to prepare, you have decided to use your CS 1331 skills to write a program that predicts the outcome of matches for you based on the players' ratings.

This program will have a Team class representing a team at the World Cup which contains SoccerPlayer objects (from goalkeepers to forwards). The GameSim class will allow you build players and teams to simulate soccer matches to help you predict the results of the real matches.

Solution Description

SoccerPlayer.java

`SoccerPlayer`'s are not concrete objects, meaning you should not be able to make an instance of one. Every SoccerPlayer should have the following:

- Instance variables that follow encapsulation:
    - `String name;`
        - Represents the name of the soccer player
    - `String country;`
        - Represents the name of the country the soccer player is from
    - `int stamina;`
        - Represents the starting stamina for the player
- Constructor:
    - Should have a `SoccerPlayer` constructor that takes in a name, country, and stamina, in that order.
    - Make sure stamina is a positive non-zero integer value. If any number below 1 is assigned to stamina, assign it to a default value of 20.
- Methods:
    - `toString();`
        - Should override Object's .toString()
        - Should return "<name> from <country> has <stamina> stamina left"
    - `equals();`
        - Should override Object's .equals()
        - A `SoccerPlayer` is equal to another `SoccerPlayer` if they both have the same name and country
        - Should properly handle null and non-`SoccerPlayer` inputs
    - `play();`
        - An **abstract** method which should not return anything.
        - This will be used to represent a player playing a game.
    - `calculateRating();`
        - An **abstract** method which should return a double.
        - This will be used to determine the rating of a player based on their statistics

- `rest();`
  - A concrete method which increases a SoccerPlayer's `stamina` by 5.
- We encourage you to make getter and setter methods as necessary

GoalKeeper.java

A `GoalKeeper` has an is-a relationship with `SoccerPlayer`. Every `GoalKeeper` should have the following:

- Instance variables that follow encapsulation:
  - `int totalSaves`
    - Represents the number of saves for this goalkeeper (shots on target that were blocked by the goalkeeper).
  - `int totalShotsOnTarget`
    - The total number of shots on target on the net defended by this goalkeeper.
  - `double ballHandling`
    - A rating used to determine this Goalkeeper's ball handling skills (a number between 0 – 100, inclusive).
- Constructors:
  - Should have a `Goalkeeper` constructor that takes in name, country, stamina, totalSaves, totalShotsOnTarget, and ballHandling in that order. Because it does not make sense to have more totalSaves than totalShotsOnTarget, you should make sure totalSaves will never be greater than totalShotsOnTarget. If a user tries to create a `GoalKeeper` that has a greater number of totalSaves than totalShotsOnTarget, you should set totalSaves to be equal to totalShotsOnTarget. You should be using appropriate constructor chaining across parent and child classes.
- Methods:
  - `calculateRating();`
    - Calculates a Goalkeeper's ball handling rating based on their statistics. The ball handling rating should be the result of dividing the `totalSaves` by the `totalShotsOnTarget`, multiplying its result by 100. Remember to handle any edge cases with this division. If the player has no shots on target, the rating should be 0. There is no need to round in this method yet.
  - `play();`
    - Decreases a player's stamina by a random amount between 1 and 100 (both inclusive). If a player's stamina is 0, their performance will be affected. See below for details. Note that a player's stamina can never be less than 0.
    - Generates a random number of shots on target for that game (between 0 and 100, inclusive).
    - Based on the number of shots on target, generate a random number of saves for that game (between 0 and the number of shots on target, inclusive). If the player's stamina is 0, the number of saves for the game should be 0.
    - Remember to add-on to the `totalSaves` and `totalShotsOnTarget` variables and update the Goalkeeper's rating based on this.
  - `toString();`
    - Should return "<name> from <country> has <stamina> stamina left. As a Goalkeeper, I have stopped <totalSaves> shots from <totalShotsOnTarget> shots on target and my rating is <ballHandling>".
    - Note that you should round the ballHandling value to the nearest hundredth in this method.
    - Make sure to reuse code as much as possible.

- `equals();`
  - A `Goalkeeper` is equal to another `Goalkeeper` if they both have the same name, country, total saves, total shots on target, and ball handling rating.
  - Make sure to reuse code as much as possible.
  - Should properly handle null and non-`Goalkeeper` inputs

Defender.java

A `Defender` has an is-a relationship with `SoccerPlayer`. Every `Defender` should have the following:

- Instance variables that follow encapsulation:
  - `int totalPasses`
    - Represents the total number of passes initiated by this `Defender`.
  - `int totalPassesCompleted`
    - Represents the total number of passes completed (received by another player of the same team).
  - `double defenseRating`
    - Represents this Defender's overall rating (a number between 0 – 100, inclusive).
- Constructors:
  - Should have a `Defender` constructor that takes in name, country, stamina, totalPasses, totalPassesCompleted, and defenseRating in that order. Because it does not make sense to have more totalPassesCompleted than totalPasses, you should make sure totalPassesCompleted will never be greater than totalPasses. If a user tries to create a `Defender` that has a greater number of totalPassesCompleted than totalPasses, you should set totalPassesCompleted to be equal to totalPasses. You should be using appropriate constructor chaining across parent and child classes.
- Methods:
  - `calculateRating();`
    - Calculates a Defender's defense rating based on their statistics. The defense rating should be the result of dividing the `totalPassesCompleted` by the `totalPasses`, multiplying its result by 100. Remember to handle any edge cases with this division. For the purposes of this homework a simple if check should suffice. There is no need to round in this method yet.
  - `play();`
    - Decreases a player's stamina by a random amount between 1 and 100 (both inclusive). If a player's stamina is 0, their performance will be affected. Note that a player's stamina can never be less than 0.
    - Generates a random number of passes (between 0 and 100, inclusive).
    - Based on the number of passes, generate a random number of completed passes for that game (between 0 and the number of passes, inclusive).
      - Note that Defenders have a 10% chance of outperforming in a particular game. If this happens, they will complete all passes.
      - If the player's stamina is 0, the number of completed passes should be 0 regardless of the outperforming chance.
    - Remember to add-on to the `totalPasses` and `totalPassesCompleted` variables and update the Defender's rating based on this.
  - `toString();`
    - Should return "<name> from <country> has <stamina> stamina left. As a Defender, I have completed <totalPassesCompleted> passes from <totalPasses> initiated passes and my rating is <defenseRating>".

        - Note that you should round the defenseRating value to the nearest hundredth in this method.
        - Make sure to reuse code as much as possible.
    - `equals();`
        - A `Defender` is equal to another `Defender` if they both have the same name, country, total passes, total completed passes, and defense rating.
        - Make sure to reuse code as much as possible.
        - Should properly handle null and non-`Defender` inputs

Attacker.java

An `Attacker` has an is-a relationship with `SoccerPlayer`. Every `Attacker` should have the following:

- Instance variables that follow encapsulation:
    - `int totalGoals;`
        - Represents the total number of goals this player has scored.
    - `int totalShotsOnTarget;`
        - Represents the total number of shots on target this player has had.
    - `double attackingRating;`
        - Represents this Attacker's overall rating. (a number between 0 – 100, inclusive).
    - `CelebrationMove celebrationMove;`
        - Represents this player's celebration move when scoring goals. The only values it can be is DIVE, CARTWHEEL, and SIUU.
        - Create your enum within `Attacker.java`
- Constructors:
    - Should have an `Attacker` constructor that takes in name, country, stamina, totalGoals, totalShotsOnTarget, attackingRating, and celebrationMove in that order. Because it does not make sense to have more totalGoals than totalShotsOnTarget, you should make sure totalGoals will never be greater than totalShotsOnTarget. If a user tries to create a `Attacker` that has a greater number of totalGoals than totalShotsOnTarget, you should set totalGoals to be equal to totalShotsOnTarget. You should be using appropriate constructor chaining across parent and child classes.
- Methods:
    - `calculateRating();`
        - Calculates an Attacker's defense rating based on their statistics. The attacking rating should be the result of dividing the `totalGoals` by the `totalShotsOnTarget`, multiplying its result by 100. Remember to handle any edge cases with this division. For the purposes of this homework a simple if check should suffice. There is no need to round in this method yet.
    - `play();`
        - Decreases a player's stamina by a random amount between 1 and 100 (both inclusive). If a player's stamina is 0, their performance will be affected. Note that a player's stamina can never be less than 0.
        - Generates a random number of shots on target (between 0 and 100, inclusive).
        - Based on the number of shots on target, generate a random number of goals (between 0 and the number of shots on target, inclusive).
            - Note that if a player's celebration move is SIUU, they are more likely to score goals. The number of scored goals should be 10% higher if that is the case. Note that the number of goals scored can never be higher than the shots on target.

            - If the player's stamina is 0, the number of goals should be 0 regardless of the player's celebration move.
        - Remember to add-on to the `totalGoals` and `totalShotsOnTarget` variables and update the Attacker's rating based on this.
    - `toString();`
        - Should return "<name> from <country> has <stamina> stamina left. As an Attacker, I have scored <totalGoals> goals from <totalShotsOnTarget> shots on target and my rating is <attackingRating>. My celebration is <celebrationMove>".
        - Note that you should round the attackingRating value in this method to the nearest hundredth.
        - Make sure to reuse code as much as possible.
    - `equals();`
        - An `Attacker` is equal to another `Attacker` if they both have the same name, country, total goals, total shots on target, rating, and celebration move.
        - Make sure to reuse code as much as possible.
        - Should properly handle null and non-`Attacker` inputs.

Team.java

Every Team should have the following:

- Instance variables that follow encapsulation:
    - `String teamName;`
        - Represents the name of the soccer team
    - `SoccerPlayer[] players;`
        - Represents the SoccerPlayers on the team
- Constructor:
    - Should have a `Team` constructor that takes in a teamName
    - Initialize the `players` array to have 11 empty spots
- Methods:
    - `toString();`
        - Should override Object's .toString()
        - Should return "Team name: <teamName>" followed by the `toString()` of each player on the team on a new line.
    - `addTeamMember(SoccerPlayer player);`
        - Should add the passed in player into the `players` array. Players will be added from the 0th to the 11th index consecutively into the array. You can assume that the player passed in will not already exist in the array. If the `players` array is already full, do not add the player.
    - `playAgainstTeam(Team opponent);`
        - This method determines which team would win based on the average rating of each team.
        - Every `SoccerPlayer` on both teams play in the match, so every SoccerPlayer should have their `play` method called. Once their rating is updated, proceed with the next step.
        - The average rating of a team is calculated by adding the calculated ratings of each `SoccerPlayer` in `players` and dividing the sum by the total number of SoccerPlayers on the team.
        - Should return a String of the `teamName` of the team that wins (which is defined as the team that has the highest average rating).
        - In the event of a tie, simply return "Tie".

GameSim.java

- This is the driver class of the program. Use the main method to test out the methods of your other classes any way you want. However, we do have a few suggestions to help you properly test your classes:
  - Create an empty array of length 4 to store teams and populate this array.
  - Create at least one player of each type (e.g. Goalkeeper) and add them to the teams
  - Run each method for a team
  - Run each method for each type of player

Submitting

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- `SoccerPlayer.java`
- `Goalkeeper.java`
- `Defender.java`
- `Attacker.java`
- `Team.java`
- `GameSim.java`

Make sure you see the message stating, "Homework 4 submitted successfully." We will only grade your last submission be sure to **submit *every file* each time you resubmit.**

Checkstyle and Javadoc

You must run Checkstyle on your submission. (To learn more about Checkstyle, examine our course's external website Java Resources page under "CS 1331 Style Guide".) The Checkstyle cap for this assignment is **20 points**. If you don't have Checkstyle yet, download it from Canvas -> Files ->checkstyle-8.28.jar. Place it in the same folder as the files you want to run Checkstyle on. Run checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar yourFileName.java Starting
audit...
Audit done.
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off (limited by the Checkstyle cap mentioned above). The Java source files we provide contain no Checkstyle errors. In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Additionally, you must Javadoc your code.

Run the following to only check your Javadocs:

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both Javadocs and Checkstyle:

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

Import Restrictions

You can import the Random class.

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`

Collaboration

No collaboration is allowed on this assignment. See syllabus for more details.

In addition, note that it is not allowed to upload your code to any sort of public repository. This could be considered an Honor Code violation, even if it is after the homework is due. Only post code on Ed Discussion in a **private** post.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files.
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read the "Import Restrictions" and “Feature Restrictions" to avoid losing points
- Check on Ed Discussion for all official clarifications
- Make sure to test your program manually based on the assignment instructions and expected outputs.

The Gradescope autograder visible to you is **NOT** comprehensive. A few test cases will be hidden when you submit. This is done intentionally so that you learn to write your own test cases for your assignments.