Assignment 4 -- Introduction to Polymorphism

Front Matter

Collaboration Statement

The last page of the course syllabus contains a statement of the department collaboration policy with regard to projects. Violation of the collaboration policy can result in failure of the course.

Reminder: Hand in Rules

Make sure you READ THESE INSTRUCTIONS THOROUGHLY

1. All project due dates are specified in Brightspace
2. All projects handed in must be NetBeans Projects or able to run via command line arguments outlined in a README file at the root directory.
3. ANY submission before the project's deadline will make all submissions eligible for 100% of the points on a re-submitted assignment
4. If you hand in a project late, you are not allowed to correct and resubmit. This is the only penalty for late submissions
5. For projects, you will hand in what is specified by individual assignments. Most of the time it will be YOUR ENTIRE PROJECT AS A `.zip` file. Name the archive with your last name only. Ex: `droppo.zip`.

Links to sections in the document. I've done my best to present the information required in order as to when it will become useful, but you may need to still go between sections.

The Grading Rubric

Grade Earned	Requirements
D	`main` method is the only `static` method in *any* class. `main` method in the driver class checks command line arguments. Usage message is correct for the number of required arguments. `main` method in the driver creates an object of the class and calls a `run()` method. All code handed in is syntactically correct -- the code compiles and the driver class runs without crashing.
C	Everything above plus `Player.java` is fully and correctly implemented. `Skater.java` works correctly with its unit tests. `Goalie.java` works correctly with its unit tests.
B	Everything above plus `Team.java` is complete and works with its unit tests. `League.java` constructor reads the data correctly. All team stats are correct when running the program's driver class.
A	Everything above plus `updatePlayerStats()` is correctly calculated and reported. League-wide statistics are correctly calculated and reported. All code is formatted correctly. Blocks are indented correctly. Revision History is complete All methods have complete JavaDoc comments.

Background

The Professional Women's Hockey League is a relatively new sports league, starting with a 2023-2024 season. Because it is a relatively new and small league in the established sport of Hockey it is relatively simple to gather all of the players' data and calculate the statistics.
For this project we will be loading a JSON "database" of the team rosters for the entire PWHL, and doing our own statistics calculations for the teams and players in the league.
The game of hockey is relatively simple, the curious can find the PWHL rules online but the simple idea is to move a small round puck across the ice and into the opposing teams goal.

In service of this basic game structure, each player on the ice that isn't a goalie has the following basic stats:

- Games Played [GP]
- Goals/Assists [G/A]
- Points [P] - The number of goals and assists combined
- +/- [plus/minus] - How many times a player is on the ice when the team scores minus how many times they are on the ice when the opposing team scores

- Penalty Infraction Minutes [PIM] - The number of minutes a player has spent in a penalty box
- Powerplay Goals/Assists [PPG/PPA] - The number of goals scored during a powerplay for their team
- Shorthanded Goals/Assists [SHG/SHA] - The number of goals or assists scored during an opposing team's powerplay.
- Points per Game [PtG] - The number of points scored per game
- Faceoffs Won [FOW] - The number of faceoffs a player has won.
- Faceoffs [FO] - The number of faceoffs a player participated in.
- Won Faceoff Percentage [WF%] - The percent of faceoffs won.
- Penalty Minutes per Game [PIMPG] - The average number of PIMs per game
- Shots [SH] - The number of shots a player has taken
- Shot Percent [SH%] - The percentage of the player's shots that have ended in a goal
- Shootout Goals/Assists [SOG/SOA] - The number of goals or assists scored during a shootout.

Goalies on the other hand have different statistics:

- Games Played [GP]
- Minutes Played [Min]
- Goals Against [GA]
- Shutouts [SO] - The number of games played where all shots were blocked
- Goals Against Average [GAA]
- Wins/Losses [W/L/OT]
- Overtime [OT] - The number of games ending in overtime.
- Saves [SVS]
- Shots Against [SA]
- Shots Received in Shootout [SOA] - The number of shootouts against the goalie.
- Shootout Goals Against [SOGA] - The number of goals earned during a shootout.
- Shootout Save Percent [SO%] - The percent of shots saved during a shootout.
- Goals/Assists [G/A]
- Penalty Infraction Minutes [PIM]

In order to determine rankings each team also has its own statistics as well:

- Games Played -[GP]
- Points - [PTS]
- Wins/Losses [W/L]
- Overtime Wins/Losses [OTW/OTL]

- Goals For/Against [GF/GA]

That sounds like a lot, but we should be able to simplify how we can conceptualize what needs to be done. Using *polymorphism* we only need a few classes to build a model of the PWHL league's statistics.

Project Kit

You Provide

- NOTHING

I Provide

- A NetBeans project with files for:
  - A completed driver class.
  - A partially completed `League.java` class.
  - A `Goalie.java` class with unit tests in its `main` method.
  - A `Player.java` class with unit tests in its `main` method.
  - A `Skater.java` class with unit tests in its `main` method.
  - A `Team.java` class with unit tests in its `main` method.
- pdata#.json - a JSON representation of a database of player statistics.
- PlayHockeyFullUML.png - the UML for the recommended solution to this assignment.
- PWHLPolymorphism.puml - the source PlantUML text for the PlayHockeyFullUML.png image.

Reading the Data

Parsing the JSON File

Each filename MUST be passed in as a system argument.

Rules of a JSON File

- EVERY object MUST be encapsulated in curly brackets `{}`. This means that the first and last characters (and lines in our case) will be `{` and `}` respectively.
- Objects are made of key/value pairs. These pairs are separated by a colon like `key:value`.
- Each key/value pair must be separated by a comma (`,`).
- Strings are wrapped in double quotes (`"`)

- Arrays are denoted by square brackets (`[` and `]`)
- The value can be its own JSON object denoted by curly brackets.
  More information on JSON objects can be found here on Wikipedia.[1]

Our JSON File Structure

Our JSON file will follow the rules outlined above, with the following extra limitations.

The League

The league will have one attribute, an array of teams with the following pattern[2]:

```
{
        "Teams": [
                TEAM1,
                TEAM2,
                TEAM3
        ]
}
```

The Team(s)

Each Team will have 8 attributes, structured as follows[3]:

```
{
        "Name": "TEAM CITY",
        "GP": 10,
        "W": 2,
        "L": 3,
        "OTW": 4,
        "OTL": 1,
        "Players": [
                PLAYER1,
                PLAYER2,
                PLAYER3
        ]
}
```

The Player(s)

The Players' JSON object may look different if the player is a skater or a goalie, so I'll be providing examples of both. However it should be noted that inside the Teams' "Players"

attribute skaters and goalies may be mixed together.[4] They will have a boolean attribute "isSkater" that may be true or false.

Skater

```
{
        "Name": "Alex Carpenter",
        "Number": 25,
        "isGoalie": False,
        "Position": "F",
        "GP": 16,
        "G": 8,
        "A": 10,
        "+/-": 0,
        "PIM": 0,
        "PPG": 3,
        "PPA": 5,
        "SHG": 1,
        "SHA": 0,
        "FOW": 174,
        "FO": 291,
        "SH": 60,
        "SOG": 1,
        "SOA": 3
}
```

Notice that these aren't all of the stats for a skater, we will need to calculate the missing stats ourselves. Details on how to calculate each of these stats can be found in `Skater.java`.

Goalie

```
{
        "Name": "Elaine Chuli",
        "Number": 20,
        "isGoalie": True,
        "Position": "G",
        "GP": 6,
        "Min": "359:23",
        "GA": 8,
        "SO": 0,
        "W": 5,
        "L": 1,
        "OT": 0,
        "SVS": 182,
```

```
        "SA": 190,
        "SOA": 0,
        "SOGA": 0,
        "G": 0,
        "A": 0,
        "PIM": 0
}
```

Again, like with the skater object not all the stats for the player will be here. The missing statistics will be calculated by you, and details will be found in `Goalie.java`.

Your Task

In the files provided you should find a UML diagram of each class' fields and methods as well as skeletons for a complete project. There are unit tests for `Player`, `Skater`, `Goalie`, and `Team`. Those should be left at the end of the files and ***NOT CHANGED***. These are the same unit tests I will be running when grading your assignments.
You will be defining the classes listed in the sub-headers below. I've listed them in the order I recommend you attempt them, as certain classes may need to be completed before you can test others. ***READING THE INPUT FILE SHOULD BE WRITTEN AFTER YOU'VE COMPLETED THE PLAYER, SKATER, GOALIE, AND TEAM CLASSES***

`Player.java`

This abstract class should be the super class of both `Skater` and `Goalie`. This class provides *protected* attributes shared across the two classes[5] and a protected constructor. By making the attributes protected we can ensure that they are only modified when the method `update()` gets called. Making the fields protected also means that you will need to provide `get`/accessor methods for these fields. You may also need a method that gets the players' position, either "Forward" for "F", "Defence" for "D", or "Goalie" for "G".
This class should also have a `toString()` method that returns the shared statistics in the following order: `GP: gp G: g A: a PIM: pim` with no ending linefeed.

`Goalie.java`

A subclass of `Player` holding additional private hitting statistics specific to goalies. You will need to build a constructor, accessors, mutators and a toString method that returns the player's information as a string like[6]:

```
Min: min GA: ga SO:so GAA: gaa W: w L: l OT: ot SVS: svs SA: sa SAV%: sav%
SOA: soa SOGA: soga SO%: so% GP: gp G: g A: a PIM: pim
```

How to Calculate the Missing Goalie Stats:

Goals Against Average

$$GAA = (GA/Min) * 60$$

The Goals Against Average is the number of goals against divided by the number of minutes played multiplied by the length of a regulation game.[7]

Save Percentage

$$SAV\% = SVS/SA$$

The save percentage is the ratio of saved shots compared to all attempts

Saving Percentage in Shootout

$$SO\% = 1 - (SOGA/SOA)$$

The saving percentage in shootout is the ratio of goals saved by the goalie in the shootout compared to the number of shootout attempts. We can get this by taking the remaining fraction of the goals scored over the attempts.
UNIT TEST WHEN YOU'VE FINISHED THIS CLASS[8]

`Skater.java`

A subclass of `Player` holding the additional private player statistics that are specific to non-goalies. You will be responsible for the constructor, accessors, mutators and a `toString()` that returns the player's information as follows:

```
GP: gp G: g A: a PIM: pim PTS: pts +/-: plusMinus PPG: ppg PPA: ppa SHG: shg
SHA: sha Pt/G: ptg FOW: fow FO: fo WF%: wf% PIMPG: pimpg SH: sh SH%: sh%
SOG: sog SOA: soa SOGW: sogw SO%: so%
```

How to Calculate the Missing Skater Stats

Points

$$PTS = G + A$$

The number of points a player has is the combination of their goals and assists.

Points per Game

$$PtG = PTS/GP$$

A player's Points per Game is equal to the average number of points scored per game[9].

Penalty Minutes per Game

$$PIMPG = PIM/G$$

A player's penalty minutes per game is the average number of minutes spent in a penalty box per game[10].

Won Faceoffs Percentage

$$WF\% = FOW/FO$$

The won faceoffs percentage is the ratio of faceoffs won compared to total faceoffs.

Shooting Percentage

$$SH\% = G/SH$$

The shooting percentage is the ratio of goals compared to shot attempts.

Shootout Percentage

$$SO\% = SOG/SOA$$

The shootout percentage is the ratio of successful goals in a shootout compared to the total number of shootout attempts.

UNIT TEST WHEN YOU'VE FINISHED THIS CLASS.

`Team.java`

This class should hold a `Map<Integer, Player>` for the team's roster as well as the team-wide stats. Because each player's number is guaranteed to be unique across teams we'll be using that for the key in our map. Each team will have the stats from the database in addition to a `points` field that is the team's score in the league. The `toString()` method should return a string formatted similar to the following:

```
NAME: name GP: gp PTS: W: w OTW: otw OTL: otl L: l
```

How to Calculate Team Points

Team points are unique to the PWHL and each teams score can be calculated by the following equation:

$$points = (W * 3) + (OTW * 2) + (OTL * 1)$$

Each team gets 3 points per win, two points per overtime win, and one point per loss to get their league score.

UNIT TEST THIS CLASS BEFORE MOVING ONWARDS[11]

`League.java`

This class holds a `Map<String, Team>` where the string key is the ***lowercase*** name of the team object stored as the value. It should define a constructor and all of the methods that the PWHLDriver class calls. Stubs are already written for all required methods. Change the stubs when you implement the method and do not comment methods out.

The League's Statistics

`update()`

Allows an operation that facilitates an update to the statistics after a game. The command choice is indicated by U followed by the data to use for the update, formatted as follows. The first string should be the team name which will be followed by zero or more lines of player data to update separated by whitespace. For goalies this would look like:

```
<number> <GP> <G> <A> <PIM> <Min> <GA> <SO> <W> <L> <OT> <SVS> <SA> <SOA>
<SOGA>
```

and for skaters this would look like:

```
<number> <GP> <G> <A> <PIM> <+/-> <PPG> <PPA> <SHG> <SHA> <FOW> <FO> <SH>
<SOG> <SOA>
```

This method should work for all types of players and should also update the calculated statistics for that player. While inputting player data a line that starts with `-1` should stop polling for player information and assume the next thing to be read in is a team name. When inputting a team name a `-1` should end the update loop.

You shouldn't assume that you will receive update values for all players on a team. A team name followed immediately by `-1` is valid input.

In the driver, upon seeing the U, should call `League.update()` that passes in the scanner being used for input. The method will read the rest of the input, perform the updates, and print out the new stats with 1 line per player updated to the scanner that was passed in. Finally,

`League.update()` should return a response that says `Updated m players on teamName` *FOR EACH* team updated where *m* is the number of players updated.

HINT: You will most likely need to include new methods for this. I'd recommend `Player.updateStats(String)` [12], `Player.calcStats()` [13], `team.updatePlayer(String): String`.

General Implementation Notes

The driver program provided should make a call to `League`'s constructor which is where the parsing of the file will happen. It is set up so that all the driver does is provide an interface to call the proper `League` methods for the basic assignment and as-such stubs for each of the methods called in this way exist already. YOU SHOULDN'T need to comment anything out! Once the data has been loaded, the driver program enters a loop that gives the user the opportunity to do any of the following three operations:

1. Look up the stats of an individual player:
   `public String lookup(String team, int playerNum)`
   The user enters the name of the team t, and is then prompted for the player number n. The possible responses by the lookup method for the League are written to stdout and should be:
   a. `No team for city t in the league.`
   b. `No player with number n is on the roster for the t team.`
   c. The statistics for the player formatted as described in either `Goalie.java` or `Skater.java` as appropriate.
2. Calculate the stats for a team as a whole.
   `public String calcTeamStats(String team)`
   The user is prompted for the name of a team t. The possible responses of the `calculateTeamStats()` method are written to stdout and are as follows:
   a. `No team for city t in the league.`
   b. The team statistics for the team as well as a full roster.
3. Calculate the team stats for the league.
   `public String calcLeagueStats()`
   The user isn't prompted for anything and one of the following is printed out to stdout
   a. `No teams are in the League.`
   b. Each team's team stats

Output Format

When outputting a player's information you should only print two lines. The first line being the player's general information like:

`t's player number n is a pos named nm and has the stats:`
followed by the player's `toString()` on the next line. This should look like:

```
Montreal's player number 20 is a goalie named Elaine Chuli and has the
stats:
Min: 359:32 GA: 8 SO: 0 GAA: 1.34 W: 5 L: 1 OT: 0 SVS: 182 SA: 190 SAV%:
0.958 SOA: 0 SOGA: 0 SO%: 0.000 GP: 06 G: 0 A: 0 PIM: 0
```

When outputting the team's statistics and roster the roster should be broken up into skaters and goalies. This output should read more like the following:

```
NAME: Montreal GP: 17 PTS: 30 W: 7 OTW: 3 OTL: 3 L: 4
SKATERS
29 - GP: 16 G: 8 A: 9 PIM: 8 PTS: 17 +/-: 9 PPG: 0 PPA: 1 SHG: 0 SHA: 0
Pt/G: 1.06 FOW: 212 FO: 370 WF%: 57.3 PIMPG: 0.50 SH: 50 SH%: 16.0 SOG: 1
SOA: 6 SOGW: 0 SO%: 16.7

GOALIES
20 - Min: 359:32 GA: 8 SO: 0 GAA: 1.34 W: 5 L: 1 OT: 0 SVS: 182 SA: 190
SAV%: 0.958 SOA: 0 SOGA: 0 SO%: 0.000 GP: 06 G: 0 A: 0 PIM: 0
```

One thing to note is that the goalie's stats are rounded to the nearest thousandth place whereas skater stats are rounded to the nearest tenth[14]. You should be able to get this same behaviour using java's `String.format()` method.[15]

What to Hand In

Zip your entire NetBeans project folder into this project and upload the zipped directory to Brightspace.

Programming Style

Your programs will be evaluated for formatting, use of meaningful identifiers, and documentation at each step.
For this assignment, the revision history must be complete, formatting/indentation must be correct, the coding conventions must be adhered to, and the JavaDoc for each method must be complete. You MUST add your name to the revision history above my name and you must NOT replace the author, because he is the original author.
Incomplete NetBeans generated JavaDoc does not count as complete documentation.
Let me know if you suspect any typographical errors.

Sample Output

COMING SOON!

---

1. From what I've been able to find this is the best collection of all of the information about JSON files. ↩
2. Where it says TEAM1/2/3 will be replaced by the team's JSON objects. This is merely for demonstration purposes. ↩
3. Like with the league, each player will be its own nested JSON object. ↩
4. I will not guarantee that all the skaters will come before the team's goalies or vice versa. ↩
5. Outlined in the UML. ↩
6. Using the abbriviations found in Background . ↩
7. The PWHL has three 20 minute periods. ↩
8. This is also a good time to make tweaks to Player as well. ↩
9. Total points divided by total games. ↩
10. Total minutes divided by total games. ↩
11. I didn't provide any this time, it is up to you to unit test this code. ↩
12. As an abstract method implemented in `Goalie` and `Player` ↩
13. As an abstract method implemented in both `Goalie` and `Player` ↩
14. Except for Points per Game which is rounded to the hundredth. ↩
15. These output examples are using the data from the PWHL website at 4:00 PM on March 15 2024. ↩