Project 3 - EuroLeague

In this project, you are asked to implement functions that will enable you to create, organize, play, and get statistics for a basketball tournament.

The objective of this project is to apply object oriented design techniques such as the representation of data along with functions using classes, implementing abstraction to hide internal data and present necessary functions for a scalable API, utilize inheritance for borrowing common features, keep track of changes in multiple instances, and in general design a fully functional software project to show what you have learned throughout the class.

Background Information about Basketball

Basketball played between two teams of five players each for a total of 4 periods. At the end of 4 periods, the team with **the most baskets becomes the winner**. If their scores are equal at the end of the 4th period, an extra period is played. This continues until their scores are not equal at the end of a period. So, although rarely, we might see a 5-period game or even a 6-period game.

In a league, there are several teams (numbers vary between leagues or even seasons), but let's assume there are 20 teams. In the beginning of the season a **fixture** is prepared where each team plays against all other teams **twice**. One in their home court, and one in the away court.

An example fixture for 4-team league is given below:

First half of the season has 3 weeks (number of teams - 1) and all teams complete their games against all others once.

```
Week 1:
team1 vs. team2
team3 vs. team4

Week 2:
team1 vs. team3
team4 vs. team2

Week 3:
team1 vs. team4
team2 vs. team3
```

Second half of the season may create a new schedule or just copy this schedule with teams swapping homes.

```
Week 4:
team2 vs. team1
team4 vs. team3

Week 5:
team3 vs. team1
team2 vs. team4

Week 6:
team4 vs. team1
team3 vs. team2
```

After each week, teams will score baskets, concede baskets, and there will be winners and losers. At the end of the season, the team with **the most wins will be the champion**. If the wins are equal, the team with the biggest score difference (scored - conceded) will be the champion.

Person class:

This is a base class that will hold the basic information about a person, and basic methods for printing and comparison.

Methods	**Explanation**

`__init__(name, lastname)`	`__init__` method should take name and lastname as parameters.
	
	`inst  = Person('Dimitris', 'Diamantidis')`
	`inst2 = Person('Juan Carlos', 'Navarro')`

`get_name()`	This method will **return** the full name of the player with space in between. (name + ' ' + lastname).
	
	`>>> inst.get_name()`
	`'Dimitris Diamantidis'`

`__str__()`	This method will **return** the full name of the player with space in between. (name +' '+ lastname) (same as the `get_fullname` method.)
	
	`>>> print(inst)`
	`'Dimitris Diamantidis'`

`__lt__()`	This method will implement less than operation between two Person instances. It will compare against their last names, and if the last names are the same it will compare against their names.
	
	`>>> inst < inst2`
	`True`

Player class:

This is a subclass that inherits from Person class and will hold information about the player, his match contributions for each week and total performance.

- Each player will have a **shooting power** associated with him, and this number will be randomly chosen when creating a new player using the given limits. **This number indicates how many shots a player will make for a total of 10 shots for a given quarter.**
- Each player will contribute to a match based on this shooting power, and each match is at least 4 periods, so the player will contribute to the score at least 4 times. Each time he contributes. The contribution score is explained in the Match class.
    - For example, a player with 6 shooting power can contribute 3 points in the first period, 9 points in the second period, 5 points in the third period and 7 points in the last period for a total of 24 points.

Methods	**Explanation**

`__init__(name, lastname)`	`__init__` method should take name and lastname as parameters. It should set the internal variables for name and last name. Additionally, It should generate a shooting power randomly with an **integer** value between **`[4,8]`** (both included).
	
	`inst  = Player('Dimitris', 'Diamantidis')`
	`inst2 = Player('Juan Carlos', 'Navarro')`

`reset()`	This method will clear player values except the shooting power, name, and team. (Basically it will revert back to the *original* state after initialization)
	
	`>>> inst.get_points()`
	`162`
	`>>> inst.get_name()`
	`'Dimitris Diamantidis'`
	`>>> inst.get_power()`
	`8`
	`>>> inst.reset()`
	`>>> inst.get_points()`
	`0`
	`>>> inst.get_name()`
	`'Dimitris Diamantidis'`
	`>>> inst.get_power()`

	8

`get_id()`	This method will return the unique player ID. This ID should be an internal instance counter that gets assigned to the next player that is created. The player ID should **start from 1.** (Note that this is the counter for Player class)
	
	`>>> inst.get_id()`
	`1`
	`>>> inst2.get_id()`
	`2`

`get_power()`	This method will return the player's shooting power as an integer.
	
	`>>> inst.get_power()`
	`8`

`set_team(t)`	This method will set the player's team to the given Team instance.
	
	`>>> team = Team('Fenerbahce', fb_manager, fb_players)`
	`>>> inst.set_team(team)`

`get_team()`	This method will get the player's team as a Team instance.
	
	`>>> a = inst.get_team()`
	`>>> print(a)`
	`'Fenerbahce'`
	
	`# Explanation it prints using Team's __str__ method.`

`add_to_points(x)`	This method will append to the player's points. Which should be done at the end of each match.
	
	`>>> inst.get_points()`
	`110`
	`>>> inst.add_to_points( 34 )`
	`>>> inst.get_points()`
	`144`

`get_points_detailed()`	This method will return the players' all performances for all played weeks as a list.
	
	Example points of the player after Week 5 matches:
	
	`>>> inst.get_points_detailed()`
	`[30, 35, 32, 32, 33]`

	Explanation: In Week 1 the player scored 30 points, in Week 2 35 points, in Week 3 32 points and in Week 4 32 points, and Week 5 33 points.
	
	Example points of the player before playing any matches:
	
	`>>> inst.get_points_detailed()`
	`[]`

`get_points()`	This method will return the sum of players' all points for all played weeks.
	
	Example points of the player after Week 5 matches:
	
	`>>> inst.get_points()`
	`162`
	`>>> inst2.get_points()`
	`166`

`__lt__()`	This method will implement less than operation between two Player instances. It will compare against **their total points**. If their points are the same, it will compare against their last names, and if the last names are the same it will compare against their names.
	
	`>>> inst < inst2`
	`True`

Manager class:

This is a subclass that inherits from Person class and will hold information about the manager and his influence weekly. Manager influence happens each week before the 1st period is played. A random number is generated and this gets added to the score. This number should be added as the manager's influence point for that week.

Methods	**Explanation**
`__init__(name, lastname)`	`__init__` method should take name and lastname as parameters. It should set the internal variables for name and last name. `inst  = Manager('Zeljko', 'Obradovic')`
`reset()`	This method will clear manager values except the name, and team. (Basically it will revert back to the *original* state after initialization) `>>> inst.get_points()`

	35
	>>> inst.get_name()
	'Zeljko Obradovic'
	>>> inst.reset()
	>>> inst.get_points()
	0
	>>> inst.get_name()
	'Zeljko Obradovic'

get_id()	This method will return the unique manager ID. This ID should be an internal instance counter that gets assigned to the next *manager* that is created. The manager ID should **start from 1.** >>> inst.get_id() 1

set_team(t)	This method will set the manager's team to the given Team instance.
	
	>>> team = Team('Fenerbahce', inst, fb_players)
	>>> **inst.set_team(team)**

get_team()	This method will get the manager's team as a Team instance. **>>> a = inst.get_team()** >>> print(a) 'Fenerbahce' # Explanation it prints using Team's __str__ method.
get_influence_detailed()	This method will return the manager's all influence points for all played weeks as a list. Example influence points of the manager after Week 4 matches: >>> inst.get_influence_detailed() [10, 3, -5, -4] **Explanation:** In Week 1 the manager contributed +10 points to the game, in Week 2, +3 points, in Week 3, -5 points and in Week 4, -4 points. Example points of the manager before playing any matches: >>> inst.get_influence_detailed() []

`get_influence()`	This method will return the sum of manager's all influence points for all played weeks.
	
	Example points of the manager after Week 4 matches:
	
	`>>> inst.get_influence()`
	`4`
	`>>> inst2.get_influence()`
	`-14`

`__lt__()`	This method will implement less than operation between two Manager instances. It will compare against **their total influence points**. If their points are the same, it will compare against their last names, and if the last names are the same it will compare against their names.
	
	`>>> inst < inst2`
	`False`

Team class:

This class will hold information about the team, its players, its matches and its results.

Methods	**Explanation**

`__init__(teamname, manager, players)`	`__init__` method should take team name, manager, and player list as parameters. Note that you should create a new list from the players to avoid aliasing problems from mutation.
	
	`fb_players = [Player('Jan', 'Vesely'), Player('Achille', 'Polonara'), Player('Marko', 'Guduric'), Player('Marial', 'Shayok'), Player('Nando de', 'Colo')]`
	
	`fb_manager = Manager('Sasa', 'Dordevic')`
	
	`team1 = Team('Fenerbahce Beko Istanbul', fb_manager, fb_players)`
	
	`team2 = Team('Maccabi Playtika Tel Aviv', ta_manager, ta_players)`

`reset()`	This method will clear all team statistics. (Basically it will revert back to the *original* state after initialization) ● Clear manager's influence points ● Clear players' points ● Clear team's fixture `>>> inst.reset()`

`get_id()`	This method will return the unique team ID. This ID should be an internal instance counter that gets assigned to the next team that is created. The team ID should **start from 1.**
	
	`>>> team1.get_id()`
	`1`

`get_name()`	This method will return the team name.
	
	`>>> team1.get_name()`
	`'Fenerbahce Beko Istanbul'`

`get_roster()`	This method will return the team players as a list of **Players** instances.
	
	`>>> a = inst.get_roster()`
	`>>> print(a)`
	`[<__main__.Player object at 0x10315ec40>,`
	`<__main__.Player object at 0x10315eca0>,`
	`...]`

`get_manager()`	This method will return the team's manager as a **Manager** instance.
	
	`>>> a = inst.get_manager()`
	`>>> print(a)`
	`'Sasa Dordevic'`
	
	`# Explanation: the name comes from the __str__ method of the Team object.`

`add_to_fixture(m)`	This method will append the given **Match** instance to the team's own fixture.
	
	`>>> match = Match(team1, team2, 3)`
	`>>> team1.add_to_fixture(team1)`
	`>>> team2.add_to_fixture(team2)`
	`>>> print(match)`
	`>>> match.play()`
	`>>> print(match)`

`get_fixture()`	This method will return the team's own fixture as a list of Match instances.

`add_result(s)`	This method will add the given result to its results and update the relevant internal lists / counters (i.e. wins / losses / conceded / scored, etc.)
	
	The parameter s is a tuple which should be in the form of `(team's own score, opposite team's score)`
	
	`>>> inst.get_scored()`
	`1301`
	`>>> inst.get_conceded()`
	`870`
	`>>> inst.get_wins()`
	`31`
	`>>> inst.get_losses()`
	`2`
	`>>> inst.add_result( (102, 95) )`
	`>>> inst.get_scored()`
	`1403`
	`>>> inst.get_conceded()`
	`965`
	`>>> inst.get_wins()`
	`32`
	`>>> inst.get_losses()`
	`2`

`get_scored()`	This method will return the team's total scored baskets as an integer.
	
	`>>> inst.get_scored()`
	`1403`

`get_conceded()`	This method will return the team's total conceded baskets as an integer.
	
	`>>> inst.get_conceded()`
	`965`

`get_wins()`	This method will return the team's total wins as an integer.
	
	`>>> inst.get_wins()`
	`32`

`get_losses()`	This method will return the team's total losses as an integer.
	
	`>>> inst.get_losses()`
	`2`

`__str__()`	This method will **return** the name of the team just like the `get_name()` method.
	
	`>>> print(inst)`
	`'Fenerbahce Beko Istanbul'`

`__lt__()`	This method will implement less than operation between two Team instances.
	● It will compare against their total points.
	● If their total points are equal, it will compare against their score difference total.
	● If that is equal as well, return either True or False (does not matter).
	
	Example for two teams with `team1 = 24` points and `team2 = 20` points.
	
	`>>> team1 < team2`
	`False`
	
	Example for two teams with `team1 = 20` points and `team2 = 20` points. `team1` scored `1000` points total and conceded `865` points which nets `+135`. `team2` scored `970` points total and conceded `800` points which nets `+170` points. (`20 == 20`, so `135 < 170`)
	
	`>>> team1 < team2`
	`True`

Match class:

This class will hold information about a match between two teams. Matches should be generated in the beginning of the season, and will be played according to the given rules.

Note that when adding match instances to the relevant lists (such as team's own fixture, or season's fixture) you should not copy or create a new instance. Just add it as an alias so that one change can appear in other places.

Methods	**Explanation**
`__init__(home_team, away_team, week_no)`	`__init__` method should take home team instance, away team instance, and week number that the match will be played on. **`team1 = Team`**`('Fenerbahce Beko Istanbul', fb_manager, fb_players)` **`team2 = Team`**`('Maccabi Playtika Tel Aviv', ta_manager, ta_players)`

	`inst = Match(team1, team2, 15)`
`is_played()`	This method will return `True` if the match is played, and `False` otherwise.

`play()`	This method will play the match. It will play 4 periods, more if the scores are equal at the end of any period after 4. This method should not return anything.
	
	In the beginning of the match, generate each manager's influence points for the teams. This is the starting score.
	
	Next, play 4 periods. In each period, get each player's shooting power, and add it with a *mood* integer randomly generated between **`[-5, 5]`** for each player. The sum of all these forms each team's points for that period.
	
	At the end of 4 periods, if the scores are equal, play one more period, and repeat this until the scores are not equal.
	
	You should update each player's performance after the match is played.
	You should update each manager's influence points after the match is played (or as soon as it is generated).
	
	If this is called twice, the second call should not do anything, since the match is already played.
	
	Example:
	
	Let's assume we have 2 teams with 3 players each and their shooting power is given next the them (for simplicity, normally there will be 5 players each):
	
	`>>> home_score = 0`
	`>>> away_score = 0`
	
	`home_team:`
	`- lion1 - 8`
	`- lion2 - 7`
	`- lion3 - 5`
	
	`away_team:`
	`- tiger1 - 4`

```
- tiger2 - 8
- tiger3 - 8
```

First, we start with the managers' addition. Between `-10, 10`:

```
>>> home_manager_point =
random.randint(10, 10)
-5
>>> away_manager_point =
random.randint(10, 10)
8
```

team1 manager gets -5
team2 manager gets 8

We update scores, and managers influence points:

```
>>> home_score += home_manager_point
-5
>>> away_score += away_manager_point
8
```

Next, we play the first period. For this let's calculate team1's players points. For each player we generate a random number between [-5, 5] to decide on the mood of the player for that period.

- For lion1, his score is 8, and he generates -3 which sums up to 5. So lion1 scores 5 points.
- For lion2, his score is 7, and he generates 5 which sums up to 12. So lion2 scores 12 points.
- For lion2, his score is 5, and he generates 0 which sums up to 152. So lion3 scores 5 points.
- The total period performance for the home team becomes 5 + 12 + 5 = 22.
- This is repeated for the away team as well.

These steps are repeated for 4 periods, and let's assume the period points for the teams are as follows:

```
home_period_scores = 22, 20, 32, 15
away_period_scores = 30, 12, 14, 16
```

When we sum up all these with the addition of manager points, we get:

```
home_score = -5 + 22 + 20 + 32 + 15 = 84
away_score = 8 + 30 + 12 + 14 + 16 = 80
```

	So the home team wins this game by (84, 80) points. (scored, conceded) If the scores were equal, they would play one more period to see if the equality is cancelled.

`get_match_score()`	This should return the match final score as a tuple with home team score first and away team score second.
	
	`>>> inst.get_match_score()`
	`(88, 92)`

`get_teams()`	This method will return the **home Team instance and away Team instance as a tuple.** (`home, away`)
	
	`>>> inst = Match(team1, team2, 2)`
	`>>> print(inst)`
	`team1 vs. team2`
	
	`>>> print( inst.get_teams() )`
	`(<__main__.Team object at 0x1032b0160>, <__main__.Team object at 0x1032aaf70>)`

`get_home_team()`	This method will return the home **Team instance.**
	
	`>>> inst = Match(team1, team2, 2)`
	`>>> print( inst.get_home_team() )`
	`team1`
	
	`# Explanation: team1 comes from the __str__ method of the Team object.`

`get_away_team()`	This method will return the away **Team instance.**
	
	`>>> inst = Match(team1, team2, 2)`
	`>>> print( inst.get_away_team() )`
	`team2`
	
	`# Explanation: team2 comes from the __str__ method of the Team object.`

`get_winner()`	This method will return the **Team instance** that won the game if the match is played. If not, **return None.**
	
	`>>> inst = Match(team1, team2, 2)`
	`>>> print( inst.get_winner() )`
	`None`
	`>>> inst.play()`
	`>>> print( inst.get_winner() )`
	`team1`

	`# Explanation: team1 comes from the __str__ method of the Team object.`

`__str__()`	This method will print **`team1 vs. team2`** if the match is not yet played. If it is played it will return **`team1 (score) vs. (score) team2`** instead.
	
	`>>> print(inst)`
	`'Fenerbahce Beko Istanbul vs. Maccabi Playtika Tel Aviv'`
	
	`>>> inst.play()`
	`>>> print(inst)`
	`'Fenerbahce Beko Istanbul (90) vs. (79) Maccabi Playtika Tel Aviv'`

Season class:

This is the class where everything is tied together. Its purpose is to create teams, assign players and managers to the teams, create the season fixture, play the season matches and get statistics about the season.

Methods	**Explanation**
`__init__(teams, managers, players)`	`__init__` method should take the filenames for the teams, managers and the players. ● For each team name in the teams file, it should create a team. ● Pick and create 5 players from the players file in order, and assign them to that team. ● The players that are used in one team should not be used in other teams. ● Assign a manager to a team in order. ● After generating all teams in an internal list, It should shuffle the list for randomization. ● It should then call the `build_fixture()` method to generate the season fixture. Additionally, it should hold all the managers, players and teams as a list internally for displaying later just in case. `>>> inst = Season('teams.txt', 'managers.txt', 'players.txt')`

`reset()`	This method will reset all statistics about the season and return to the original state after initialization. You may -if you want- rebuild the fixture again, but not necessary.

`build_fixture()`	This function should generate the season fixture for the teams using round robin tournament. Pick a scheduling algorithm from the following link and implement that algorithm. (Either **circle** or **berger**'s methods can be implemented)
	
	`https://en.wikipedia.org/wiki/Round-robin_tournament`
	
	Internal representation is up to you, but each match should be created using the `Match` class.
	
	Note that `build_fixture()` should be called in the initialization time (inside `__init__`) however, if we call it again, it should reset the fixture, and recreate it.
	
	General fixture rules apply;
	● Teams should only play with each other twice. Once as a home team, once as an away team.
	● Each week all teams should play with each other.
	● You can repeat the first half of the season (meaning one set of matches completed) for the second half of the season with the home and away teams reversed.
	○ As an example for a 10 team season; Team1 (H) can play with Team2 (A) in the first week, and Team2 (H) can play with Team1 (A) in the 10th week.
	● An example fixture for 4 teams is given in the beginning of this document.
	
	`>>> inst.build_fixture()`

`get_week_fixture(week_no)`	This function should get the list of matches that will be played on the given week. It should return as a list of `Match` instances. If the `week_no` is 0 or more than the number of weeks in a season, it should return `None`
	
	`>>> a = inst.get_week_fixture(0)`
	`>>> print(a)`
	`None`
	
	`>>> a = inst.get_week_fixture(1)`
	`>>> print(a)`

	[<__main__.Match object at 0x10315ec40>, <__main__.Match object at 0x10315eca0>]
	
	>>> print(a[0])
	team1 vs. team2
	
	>>> print(a[1])
	team3 vs. team4
	
	>>> inst.play_week()
	
	>>> print(a[0])
	team1 (118) vs. (116) team2
	
	>>> print(a[1])
	team3 (125) vs. (135) team4

get_week_no()	This method will return the next week's number.
	
	>>> inst = Season('teams.txt', 'managers.txt', 'players.txt')
	>>> inst.get_week_no()
	1
	>>> inst.play_week()
	>>> inst.get_week_no()
	2

play_week()	This method will play the current week's matches starting from week 1. Each time it is called, it will move to the next week. If all the fixture matches are played, it should not do anything.
	
	After each play you should sort your internal lists (player, manager, teams) so that they are always in order.
	
	>>> a = inst.get_week_fixture(1)
	>>> print(a)
	[<__main__.Match object at 0x10315ec40>, <__main__.Match object at 0x10315eca0>]
	
	>>> print(a[0])
	team1 vs. team2
	
	>>> print(a[1])
	team3 vs. team4
	
	>>> inst.play_week()
	
	>>> print(a[0])
	team1 (118) vs. (116) team2

	`>>> print( a[1] )` `team3 (125) vs. (135) team4`

`get_players()`	This method will return a list of all players in the league as `Player` instances.
	
	`>>> a = inst.get_players()`
	`>>> print(a)`
	`[<__main__.Player object at 0x10315ec40>, <__main__.Player object at 0x10315eca0>, ...]`

`get_managers()`	This method will return a list of all managers in the league as `Manager` instances.
	
	`>>> a = inst.get_managers()`
	`>>> print(a)`
	`[<__main__.Manager object at 0x10315ec40>, <__main__.Manager object at 0x10315eca0>, ...]`

`get_teams()`	This method will return a list of all teams in the league as `Team` instances.
	
	`>>> a = inst.get_teams()`
	`>>> print(a)`
	`[<__main__.Team object at 0x10315ec40>, <__main__.Team object at 0x10315eca0>, ...]`

`get_best_player()`	This method returns the best player at that week as a `Player` instance.
	
	`>>> a = inst.get_best_player()`
	`>>> print(a)`
	`Name Lastname`
	
	`# Explanation: Name Lastname comes from the __str__ method from Player (or Person) class`

`get_best_manager()`	This method returns the best manager at that week as a `Manager` instance.
	
	`>>> a = inst.get_best_manager()`
	`>>> print(a)`
	`Name Lastname`
	
	`# Explanation: Name Lastname comes from the __str__ method from Manager (or`

	`Person) class`

`get_most_scoring_team()`	This method returns the most scoring team at that week as a `Team` instance.
	
	`>>> a = inst.get_most_scoring_team()`
	`>>> print(a)`
	`team1`

`get_champion()`	This method returns the champion team as a Team instance at the end of the season, If the season is not complete, it should return None.

General Notes:

You should start implementing your classes one-by-one and test them individually before moving to the next class.

- **Person**, **Player**, **Manager** classes should be very straight-forward as they are mostly similar to what we worked on in the class.
- **Team** class is a little more involved such as
    - when we pass the players, we need to set each player's teams
    - we need to keep track of our wins and losses from game scores and game scores should be in proper order.
- **Match** class requires extra attention because we play the match there with 4 periods and keep track of the final score, team's score, player's score and manager's influence.
- **Season** class, while most of the stuff is easy to implement, building fixtures requires an implementation of the given algorithms, and you need to be careful to get it right.

- The rest of the stuff is figuring out how things are connected and making sure using the right methods for calculations.
- In general, you return the instance, instead of a string. Do not clone or copy the instance, return it as is.
- In general, you should NOT access internal class variables directly. We tried to include as many getters as possible. If you need and want, you can implement and add your own methods. These list are the only ones that we will be testing.

P.S. You can reach an example implementation from `http://10.1.60.120:5002` address. (works only in GTU)

Good luck, have fun!