Main Project Report

Jason Ballantyne

13432788

COMP30820: Java Programming

UCD School of Computer Science

10/05/2021

Brief Explanation of Project:

This report will outline the details of my program that implements three simple games: a lottery game, a coin flip game and a game of rock-paper-scissors. The program allocates points to a victorious player and stores these points on a leaderboard. The program is ran using “App.java” and allows the user to select between being 1. New Player 2. New VIP Player or 3. Quit. The program then prompts the new player to enter his/her name before allowing the player to choose to play one of the games we alluded to earlier. After the game has been completed, the player has an option to select another game or enter -1 to return to the main menu. When the player quits from the main menu, the program displays all the players who have played and their points on a leaderboard, sorted by their point total. If a play has chosen to be VIP Player, they will receive special bonuses. These bonuses include the inclusion of “(VIP)” to the player’s name on the leaderboard and the addition of 2 bonus rounds for each of the games.

```
Player          :               Points
Jane (VIP)          30
Jill                20
Tom                 10
```

As the players are created, their names and points are stored in a text file that is created as “leaderboard.txt” and displayed upon exit of the application. Users are encouraged to have multiple interactions with the games as the number of times users can play the games are infinite.

```
You selected Rock
Computer is choosing one.....
Computer selected one is: Scissor
John won the game and earned 10 points
John, Thank you for playing Rock Paper Scissors game

Hello John. Please choose a game, or -1 to quit:
1: Lottery
2: Coin flip
3: Rock Paper Scissors
```

Technical Breakdown:

As mentioned, the program is initiated by runing “App.java”, here we have our main method that prompts the user to enter what type of player they are, this is the entry point of our java program. Within this file, we also have the following private modifiers:

- sortResults: Arranges the order of players on the leaderboard based on points total.
- showFinalResults: Displays final results in the console.
- writeToFile: Writes final results to leaderboard.txt.
- mainMenu: Prints the main menu.
- gameMenu: Prints the game menu.
- gameSelection: Prompts user to enter a name and add it to the list.

As we learned in lecture 11 on objects and classes, these are all examples of private visibility modifiers that can only be accessed by the declaring class.

“LotteryGame.java”, “CoinFlipGame.java”, and “RockPaperScissorsGame.java” are our game files. These files include examples of getter and setter methods that exist as “getInstance” and “setPlayer”. These files also contain the public modifier “start” which loops over their respective game. As learned within this same lecture (11), getter functions are used to make a private data field accessible whereas setter functions are used to enable a private data field to be updated. The public modifier can be accessed within the class, outside the class, within the package and outside the package.

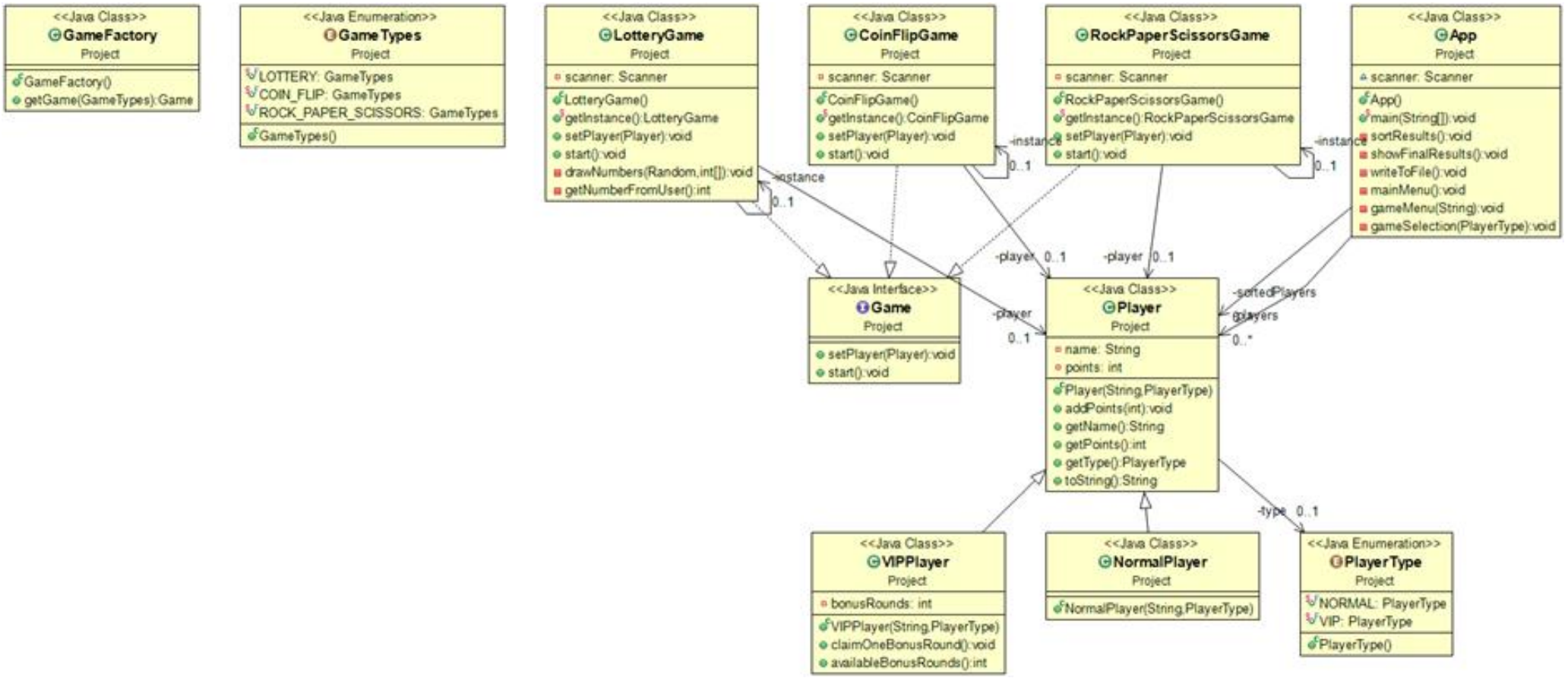
Our “Game.java” uses public interface, as learned in our final lecture, lecture 20, interfaces specify what a class must do and must not do. It is essentially the blueprint of the class. “GameFactory.java” implements a basic public class “GameFactory” that directs the console to the specified game based off of the game type. “GameTypes.java” introduces the implementation of public enum which is a special class that represents a group of unchangeable variables.

“Player.java” contains the following public methods:

- Player: Gets the player’s name, type and points.
- addPoints: Totals the player’s points.
- getName: Returns the player’s name.
- getPoints: Returns the player’s points.
- getType: Returns the player’s type.
- toString: Returns the string of the player’s name and points if they are a “Normal” player. Returns the string of the player’s name + type and points if they are any other type of player, such as a “VIP” Player.

“PlayerType.java” also uses this public enum class which, as described earlier, is a special class that represents a group of constants or unchangeable variables. “NormalPlayer.java” is a subclass of “Player.java” and uses the name and type fields of the parent class. “VIPPlayer.java” is also a subclass of “Player.java” and implements public methods to allocate additional bonus rounds to the player.

UML Class Diagram



UML Class Diagram Discussion:

The UML diagram above provides us with a static view of our application. Within the diagram, a class is represented by a box with 3 compartments. The uppermost one contains the class name. The middle one contains the class attributes and the last one contains the class methods. The design of this project and relationship between classes were built around the implementation of *inheritance* and *dependencies* on classes.

Firstly, we will look at inheritance. As we learned in lecture 14, inheritance enables you to define a general class and later extend it to more specialised classes. If we examine our "Game" class, this is an example of inheritance in action. Our "LotteryGame", "CoinFlipGame" and "RockPaperScissorsGame" classes are all derived from the "Game" class. Similarly, if we observe the "VIPPlayer" and "NormalPlayer" classes, both are derived from the "Player" class. These were intentional design choices in order to create new classes that are built upon existing classes. This method allows us to reuse methods and fields of the parent class.

We will now look at dependencies. Dependencies are directed relationships which depict that an element is dependent on another element for implementation. If we examine our UML diagram, we can see there are several classes that are dependent upon the "Player" class. These include the "LotteryGame", "CoinFlipGame", "RockPaperScissorsGame", and "App" classes. Similarly, we can see the "Player" class itself is dependent upon the "PlayerType" class. Dependencies were critical for the sequencing of tasks and making supporting decisions within this project.

Conclusion

In conclusion, we have briefly explained our project, highlighting how both the minimum and advanced specifications were implemented. We then outlined relevant course concepts and how they were carried out in our project. Finally, a UML class diagram was provided with a corresponding discussion piece detailing how inheritance and dependencies were at the core for driving both design choices and decisions made for this project.