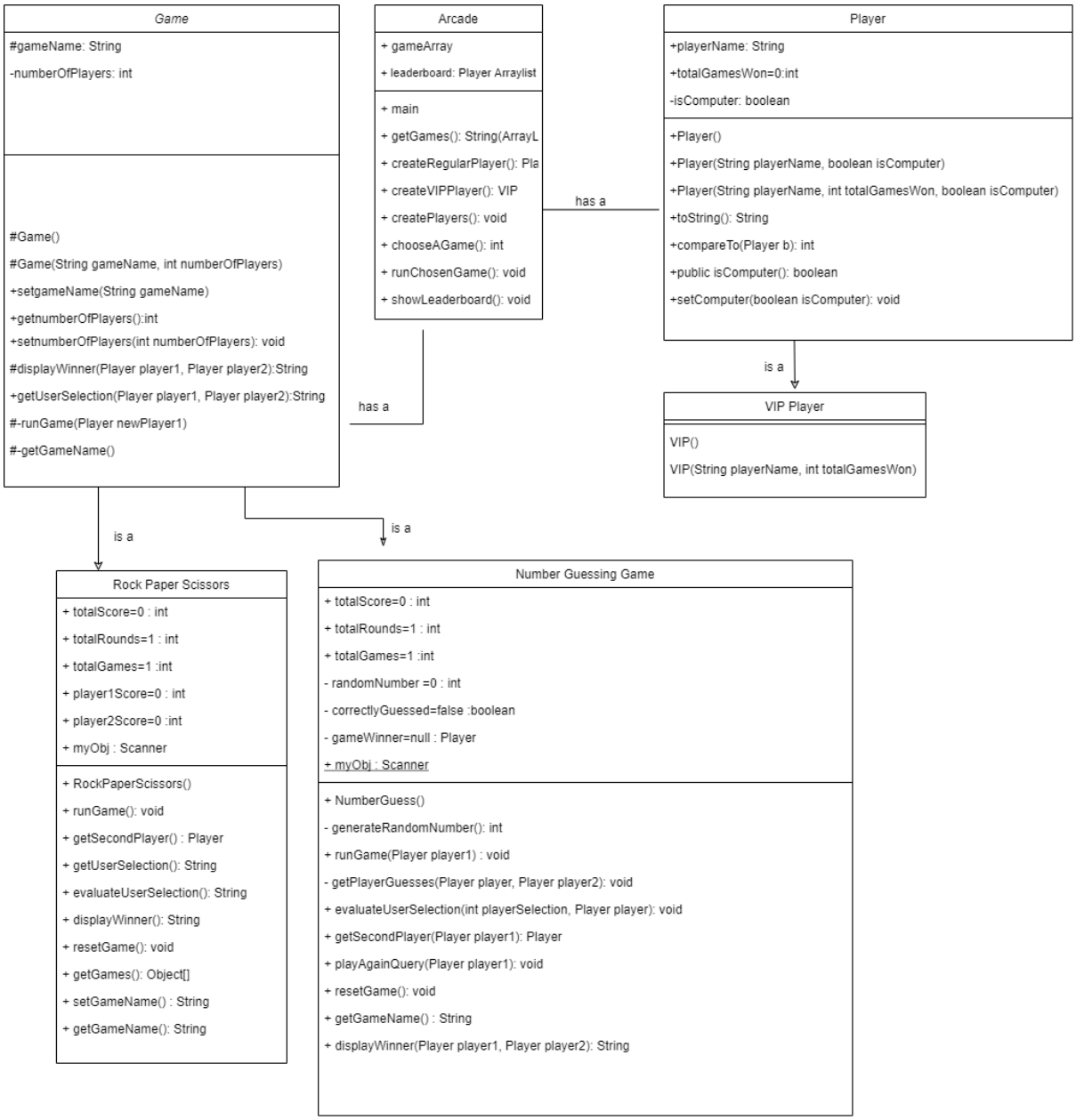
Project Overview

This report begins with a ***UML diagram*** that gives an overview of the program, its classes and their interaction with each other. The UML diagram is followed by a ***discussion*** on the ***classes, programming concepts, design decisions*** and ***data structures*** used. A ***flow chart*** is also provided in order to easily explain logic and functionality of the games and how they sit within the overall structure.

UML DIAGRAM



Arcade Class:

The project has an arcade that is central to the whole project. In essence, it "runs the show", as an Arcade would in reality. It hosts several players in an ***array list*** of type "Players", a leader board and 2 games – 'Rock Paper Scissors' and a 'Number Guessing Game'. The game is run through the "Arcade" ***class***, which has a ***main method*** to manage the flow of program execution. A **static arraylist** data structure of type "Player" (which is a class I defined in the program) has been created to host all of the "Players" objects. I decided that the number of players in the Arcade will be unlimited, therefore I did not want a regular array limiting the amount of Player objects that could be created in memory. In contrast, I decided that I would limit the amount of "games" in the Arcade to 2 games only. For this reason, I saved the list of Arcade games in a regular ***static array*** of size 2. The decision to make the arrays static was because the Players and leaderboard belong to the Arcade class itself rather than one instance of it.

```
public static Game[] gameArray = new Game[2];

public static ArrayList<Player> leaderboard = new ArrayList<>();//(
```

The program execution is managed in the Arcade via a "***Do, while***(!done)" loop. I've created a ***Boolean*** done=false variable within the ***main method*** in the Arcade ***class***. Whilst this remains equal to false, the Arcade will continue to run. This Boolean variable will only equal "true" when a user chooses to leave the arcade/quit, at which point the program will shut down and the final leader board will be displayed to the user.

Maintaining a persistent leader board

When a user "enters" the Arcade, the leader board from previous runs will be ***read*** from a .txt file and the corresponding player objects and their points will be created. This is done through the public static void method "createPlayers()", in the Arcade class. This method is called automatically when the Arcade class begins executing. Players are read in from the text file using "***Files.readAllLines***" and in conjunction with "***java.nio.file.Paths.get***" so that the ***path*** will not be hard coded for any particular operating system. I've made use of an ***alternative*** "***for***" ***loop*** to loop through the text file, which takes the following structure: "for(String line: playerList){….}". Each line is read in one by one as a ***String*** array. I take the item at ***index*** 0 (after ***splitting***) and save it as the players "name". Item at index 1 is saved as the players "totalGamesWon". Both of these ***variables*** are then passed to the ***constructor*** of the Players class to create a new player (which subsequently adds each of the players to the leaderboard). ***Input and output errors*** are caught with a ***try, catch*** statement.

```
try {
    List<String> playerList = Files.readAllLines(java.nio.file.I

    for (String line: playerList) {
        String[] playerFields = line.split(",");
        String name = playerFields[0];
        int gamesWon = Integer.parseInt(playerFields[1]);
```

Showing the leader board

To show the leader board, the method "showLeaderboard()" method has been created. This sorts the leader board using ***Collections.sort.*** (The ***compareto*** method in Players on the leader board compares players on the basis of their total games won). It then ***prints*** the ordered leader board to the screen and also saves the updated leader board to a file using ***FileWRITER*** inside a ***Try,Catch*** *block.* Player names and games won are accessed via their ***getters.***

Creating Players in the Arcade

The user will be asked to choose whether they are a Regular Player or VIP Player (a VIP Player is a Player who starts with a bonus point for "total games won" and whose name appears on the leader board accompanied by "**VIP". The selection is taken in via a ***Scanner*** input and is then managed via ***if/else*** control statement. If a user chooses to create a regular player, the ***constructor*** for the "Player" class is called, the user inputs their name, and a regular player is created. If the User selects option 2 (to create a VIP player), the constructor for the VIPplayer class is called and a VIP player is created. A VIP player is a ***subclass*** of Player, with some slight alterations discussed below. If a user selects option 3 (Quit), the ***Scanner*** is closed, the showLeaderboard() function is called to display the leader board and the program shuts down.

- **Player Class**

  The Player class creates an object of type Player and implements ***Comparable***<Player>, which compares Players on their total games won. It has two ***public class variables*** that are specific to the instance of the Player. This first is "Player Name", the second is "Total Games Won". If the ***default constructor*** is called, a player's name will be set to "Player name unknown", otherwise the Players name will be passed in to ***the arg constructor***.

  "Player" also has a ***private Boolean*** variable called "isComputer" which specifies if the Player is type computer or not. This is ***private***, as it is information that is only relevant to the methods within the Player class and affects game play. This Boolean will be set to "True" if the user chooses to play against the computer instead of against another player. An isComputer Boolean set to true will impact the flow of the game so that ***random*** guesses/plays will be generated by the computer for each round of each game.

  This class ***overrides*** the ***toString*** method to display the players name. It also over rides the ***compareTo*** method to compare players based on their total games won.

  It has a ***getter*** and ***setter*** for the private "isComputer" variable. The player name and totalGameswon are easily accessible as they are both public. I've decided to keep totalGamesWon public with the faith that users will play honestly and not update the totalGamesWon to give themselves an advantage in the game. It would also be possible to set this to private and create ***getters*** and ***setters*** for it to make it slightly less accessible to dishonest play. (However, I wanted to demonstrate the user of a private versus public variable for the purpose of this assignment).

  When a Player is created, it automatically adds the ***instance*** itself to the leaderboard by referring to itself as ***this:***

```
this.totalGamesWon=0;
//Add the player to list of players on the arcade leaderboard
Arcade.leaderboard.add(this);
```

- **VIP class**

  This class ***extends*** player and offers some additional benefits. When a VIP player is created, their "totalGamesWon" will start on 1 (versus 0 for a regular player). In effect, they start the game

with an advantage when compared to other players. Their name will also appear with "***VIP*" beside it on the leaderboard.

It has two constructors. One ***no arg constructor*** which sets the default name to "VIP Players name unknown" and starting points to "1" and another which sets the player name in the **super class** variable to the player name (followed by ***VIP***).

Beginning a game

After the player has been created, they will be asked to choose from two games: Rock Paper Scissors or a Number Guessing Game. This choice is collected with a ***scanner*** input and the is managed through a ***switch*** statement. ***Input MismatchExceptions*** are managed with a ***try catch*** block. This operation is modularised into a method called "runChosenGame", which is called by the main method in Arcade when the program starts executing. If a user selects game1 (which is set to RockPaperScissors at the moment, but is built in a way that it could be easily updated), then the game 1 will begin to execute Rock Paper Scissors by calling the ***public void method*** *runGame()* that belongs to the relevant game that was chosen. If a user inputs 2, the above process will occur for the Number Guessing game.

Second Player or Computer?

When the user has chosen a game, they will be asked to choose between playing against the computer or another player. If they opt to play against the computer, no further action is required, and the game will begin. If instead they choose to play someone else, the name of the other player must be inputted and a second "Player" object will be created. All players start with "0" wins (except for VIP players that are awarded a +1 win from the beginning).

<u>Game Class</u>

The game class is an ***abstract class*** which sets the blueprint of its subclasses. Most games will share common characteristics, such as a name and number of players. A game cannot be instantiated by itself, without some rules and logic created for it, hence the decision to make this an ***abstract*** class.

When a game is created, the constructor checks that there is enough room for the game in the Arcade. (Remember – we used a fixed array of size 2 in the Arcade to limit the amount of games it could hold.) If there are already two games in the array, it will print a message which reads "No room in the arcade…". Note, this will not occur with the game I have created as there are only two games and the logic in the Arcade would not lead to this occurring. However, this is included to ensure the project is ***scalabale.*** I've made the gameName variable ***protected*** so that it can only be accessed by the methods in the package.

Rock Paper Scissors Class & Number Guessing Game Class

These classes ***extend*** the ***abstract*** game class. They include a number of concepts that were covered in class, including ***static variables, constructors, Random class, void methods, object creation,if/else statements, overriding toString, calls to the superclass, modularization (by*** *breaking the tasks into smalle/resusable methods****), do/while loops, Scanner class, a range of data types (****int, boolean, String, char, string and characater operations (such as charAt()/toLoweCase())****.***

Rock Paper Scissors will choose the winner of four rounds. The Number Guessing game will continue until the number has been correctly guessed.When a game has finished executing, a player will be offered the option to play again or return to the main menu. If they choose to return to the main menu, they can choose to play another game or can view the arcade leaderboard. Alternatively, they can also choose to quit. On quitting, a player will be presented with the final leaderboard before the game shuts down.

I've decided to include a flowchart to explain the execution of the game, as I believe this will explain the game better than words can:

Flowchart - Game Logic:

The below flow diagram has been created to provide a quick and easy demonstration to the corrector on the flow of the program and the corresponding design decisions made.