Design:

Main interaction classes:

First we will have our main “Ozlympic" class, which will first initialize all variables and then start the loop menu. A “Driver” class must be implemented to perform these actions, the driver class will have a method to read content from files in order to initialize all games, officials and athletes, which will be stored as properties in the same driver class. The “Driver” class will also have a main menu method which will constitute of a while loop which will run until the user selects the option required for the program to end.

By selecting an option, the Driver class will call a specific method of a different class, a Menu class, which contains mainly static variables and methods. This class contains static methods and variables so that any method from another class can access the variables and make changes if needed. The Menu class contains all methods that will actually display anything on screen. In addition, the Menu class contains static variables for the current game selected and the predicted athlete as well as accessors and mutators for the same variables.

Game classes:

The “Game” class has three subclasses which will be the three types of games that can be played in the Ozlympic games. In addition to having an inheritance relationship with these three classes, the Game class has a composition relationship with an array of athletes that represent the athletes that will compete in said game, as well as an “Official” which is unique for each game.

The “SwimGame”, “RunningGame” and “CycleGame” classes are subclasses of the Game class and contain simply a constructor which will validate that only certain types of athletes can be added to that game’s description (e.g. Swimmers for a SwimGame object).

The Game class is an abstract class, so no generic “Game” objects can be created, the type of game to be run has to be specified by one of its child classes.

Participant classes:

The Participant class is an abstract class, therefore no generic “Participant” object can be created. Since the information of all athletes is read in the same way as the information of the officials, we use the constructor of this class to initialize the object of the future official or athlete. The way this class's constructor works is that it takes in a string which will be the string read from the file in the way "IDINameIAgeIState", and the constructor method will decode the string into parameters for the current object.

Official

The official class exists as a subclass of the Participant class. The Official objects exist only to be assigned to a particular game, which is why the only other parameter that these class has is the integer variable "taken" which tells us whether or not the official is already mediating a game.

Athlete

The Athlete class is also an abstract class as no real objects should be generated to be generic Athlete objects, but all should have a discipline involved. Within this class, an abstract compete method will be defined. Also, this class implements an "Comparable" interface which will allow us to sort the array of athletes depending on a parameter. This class adds two new parameters which are the score, meaning the time that the athlete obtained in its last race, and the points, which denote the total number of points accumulated during the games.

As subclasses for the Athlete class, we have the Swimmer class, the Sprinter class, the Cyclist class and the SuperAthlete class. Three interfaces exist as well which are CanSwim, CanRun and CanCycle. each athlete has a corresponding interface which will define a specific method that will run in the particular class's compete() method. The SuperAthlete class implements all three interfaces as that class can participate in all three types of games.

When a game is ran, depending on the type of athletes that correspond to that game, a random number will be generated in the range according to the interface implemented in the compete() method. Those results are stored in the current Game object's Athlete array. Since the athlete has an implemented comparable interface, the array of Athletes (competitors) will be sorted using Arrays.sort(), by the parameter specified, which in this case is the current score parameter.

After ordering the array of athletes of each game, points will be awarded to the first place (array's cero index), second place (array's first index) and third place (array's second index). Then, as the user's prediction was stored as an Athlete object, both the prediction and the first place's ID will be checked against each other, if they match a message indicating the prediction was correct will be displayed. An alternative that was also explored for the process of comparing the two objects was to use the .equals() method.

Information

Since the information in an object persists for as long as the object persists, all game information is stored in objects. the Driver class which is called at the beginning of the program initializes an array of athletes, an array of officials, and an array of games (which on its own also initializes a small array of athletes and one official).

The information that we need to always have like the names, IDs, countries etc.. Are stored in simple text files which are read by our program.

The current game to run and the user's prediction are stored in a static variable in the Menu class, as we will constantly be accessing and changing the values of these parameters.

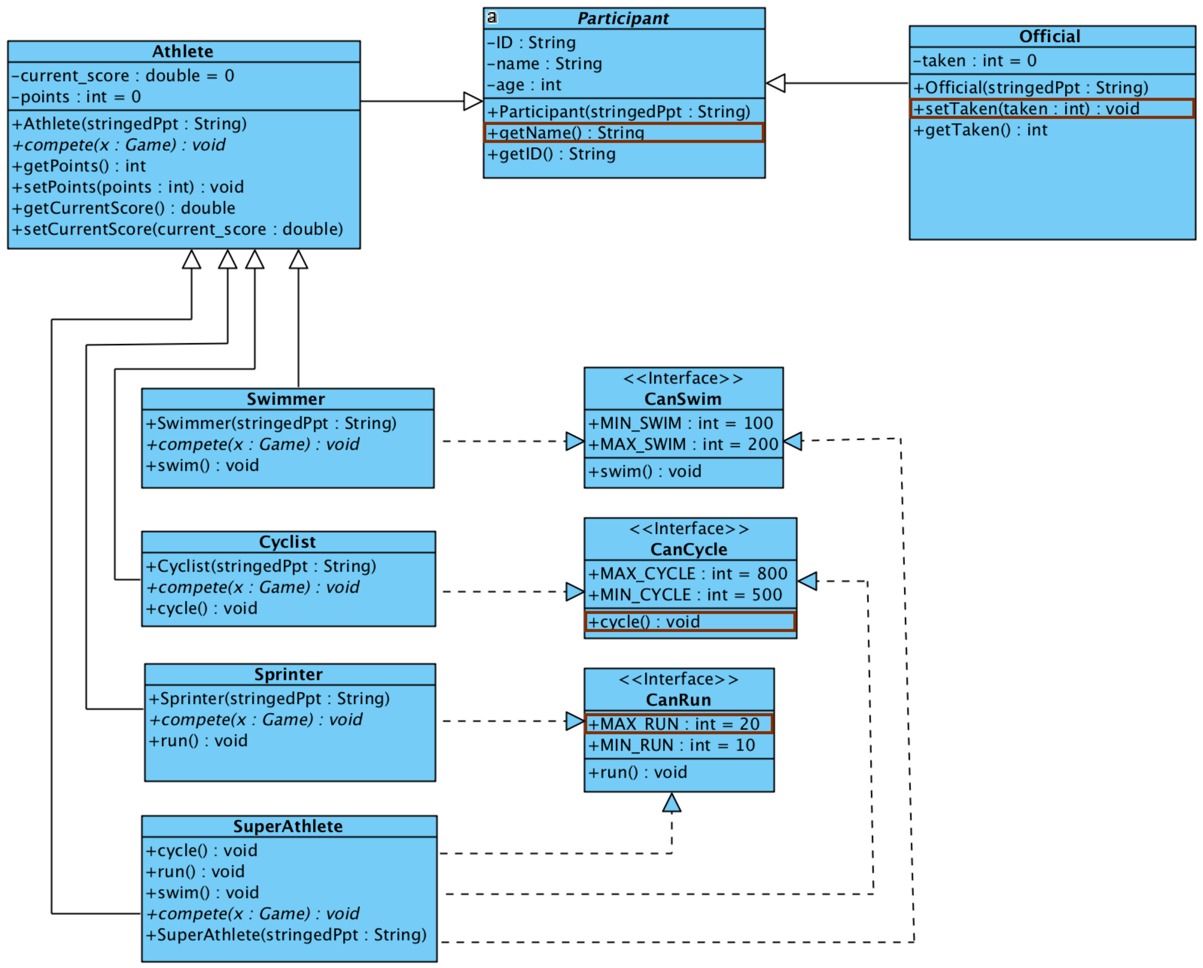
Answers to questions

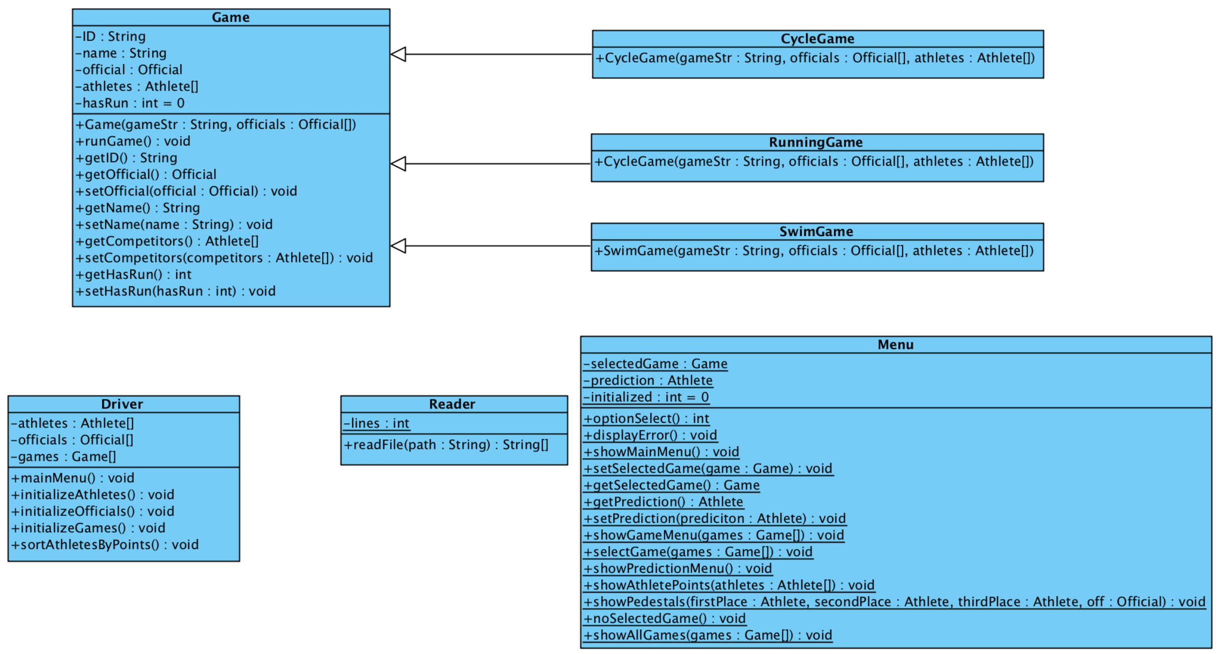
1. Core information of games and athletes, like names and IDs will be stored in files. Secondary information like which athlete corresponds to which game or the points obtained by each athlete will be stored in Objects, therefore that information will not exist after the game is closed. User selections and predictions will also be stored in variables, but in this case static variables are chosen for easy access.

2. Generic generation is forbidden for the Participant class, the Game class and the Athlete class by using abstract classes, since no objects should be generated of these types.

3. When a game is ran, the current game contains an array of athletes that will participate. The compete method is called for each athlete, and the value obtained from it will be stored in the athlete's object. After all athletes have competed the array of athletes will be sorted according to which athlete got the less time in their last compete. Since no object is being deleted or rewritten, all information will remained stored in that object for as long as the program is running.

4. There are two ways in which user prediction can be checked. When the user selects a prediction, that prediction is stored in an object, so to check if the prediction is correct, we retrieve the game's first place, which is currently stored in index zero of the game's 'competitors' variable, and can either compare both IDs (firstPlace.getID() == prediction.getID()) or use the .equals() method to check if the whole object is the same (firstPlace.equals(prediction)).


Participant
-ID : String
-name : String
-age : int
+Participant(stringedPpt : String)
+getName() : String
+getID() : String
Athlete
-current_score : double = 0
-points : int = 0
+Athlete(stringedPpt : String)
+compete(x : Game) : void
+getPoints() : int
+setPoints(points : int) : void
+getCurrentScore() : double
+setCurrentScore(current_score : double)
Official
-taken : int = 0
+Official(stringedPpt : String)
+setTaken(taken : int) : void
+getTaken() : int
Swimmer
+Swimmer(stringedPpt : String)
+compete(x : Game) : void
+swim() : void
<<Interface>>
CanSwim
+MIN_SWIM : int = 100
+MAX_SWIM : int = 200
+swim() : void
Cyclist
+Cyclist(stringedPpt : String)
+compete(x : Game) : void
+cycle() : void
<<Interface>>
CanCycle
+MAX_CYCLE : int = 800
+MIN_CYCLE : int = 500
+cycle() : void
Sprinter
+Sprinter(stringedPpt : String)
+compete(x : Game) : void
+run() : void
<<Interface>>
CanRun
+MAX_RUN : int = 20
+MIN_RUN : int = 10
+run() : void
SuperAthlete
+cycle() : void
+run() : void
+swim() : void
+compete(x : Game) : void
+SuperAthlete(stringedPpt : String)


Game
-ID : String
-name : String
-official : Official
-athletes : Athlete[]
-hasRun : int = 0
+Game(gameStr : String, officials : Official[])
+runGame() : void
+getID() : String
+getOfficial() : Official
+setOfficial(official : Official) : void
+getName() : String
+setName(name : String) : void
+getCompetitors() : Athlete[]
+setCompetitors(competitors : Athlete[]) : void
+getHasRun() : int
+setHasRun(hasRun : int) : void
CycleGame
+CycleGame(gameStr : String, officials : Official[], athletes : Athlete[])
RunningGame
+CycleGame(gameStr : String, officials : Official[], athletes : Athlete[])
SwimGame
+SwimGame(gameStr : String, officials : Official[], athletes : Athlete[])
Driver
-athletes : Athlete[]
-officials : Official[]
-games : Game[]
+mainMenu() : void
+initializeAthletes() : void
+initializeOfficials() : void
+initializeGames() : void
+sortAthletesByPoints() : void
Reader
-lines : int
+readFile(path : String) : String[]
Menu
-selectedGame : Game
-prediction : Athlete
-initialized : int = 0
+optionSelect() : int
+displayError() : void
+showMainMenu() : void
+setSelectedGame(game : Game) : void
+getSelectedGame() : Game
+getPrediction() : Athlete
+setPrediction(prediciton : Athlete) : void
+showGameMenu(games : Game[]) : void
+selectGame(games : Game[]) : void
+showPredictionMenu() : void
+showAthletePoints(athletes : Athlete[]) : void
+showPedestals(firstPlace : Athlete, secondPlace : Athlete, thirdPlace : Athlete, off : Official) : void
+noSelectedGame() : void
+showAllGames(games : Game[]) : void