Project 2

Nelson Villatoro

CMSC215

February 5, 2025

**Program Goals & Objectives**

The purpose of this program is to collect and analyze data (name, age, and height) pertaining to basketball players, and then display the average age of all players and who among is the tallest under the average age.

Design for the program composed of creating and implementing the use of 4 Java classes: Height, Player, Player1

Class purposes are as follows:

Height: Manages and normalizes player height measurement and presentation.

Player: Stores and manages individual basketball player information including name, height, and age.

Player1: Manages basketball player statistics, collecting user input data such as names, ages, and heights. It specifically helps identify the tallest player who is at or below the average age of all players.

Program Development Schedule:  
  
Day 1: Height and Player Class

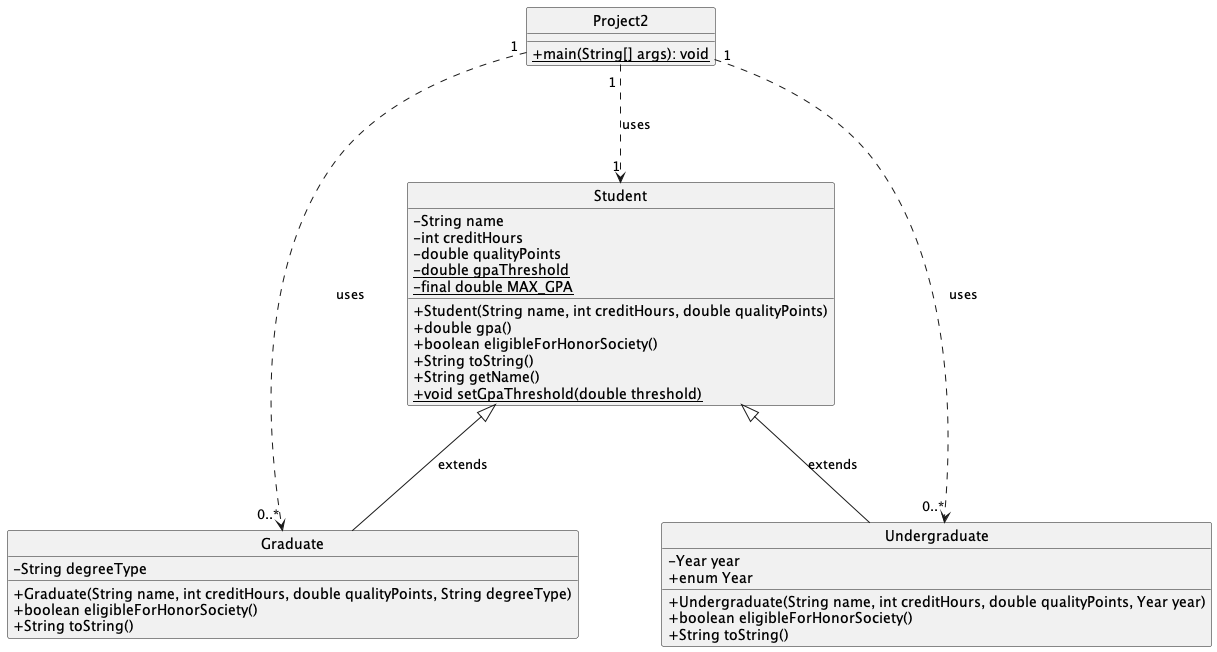
Day 2: Player1 Class

Day 3: Putting all classes together to function as a whole and bootstrapping via Main class.

**Program Functional Requirements**

1. The program must prompt the user to enter player date in the format: Name age height AS: Name age feet inches until an empty string is submitted as the final entry.
2. The program must parse the entered values (i.e., name, age, feet, inches) to be used for data analysis purposes.
3. Each player entry must be stored in an ArrayList for analysis throughout the program.
4. The program must ‘normalize’ height in a manner wherein the value for inches must be under 12 (the number of inches per foot). Ex: User enters 6 for feet, 16 for inches… program will convert this to: 7’ 4” (7ft 4in).
5. Once all player data has been entered, the program must calculate the average age of all player ages entered.
6. The program must display the calculated average age of all players.
7. The program must display who among all players analyzed is the tallest player whose age is less than the avg age.
8. The program must terminate when no user data is entered.

**Honor Society Eligibility UML Class Diagram**

****

**Testing**

I performed testing to verify the program's functionality. Below is a traceability matrix that details the unit tests conducted to ensure proper height normalization, check for input validation, confirm the program's core functionality, and verify that the program terminates as expected.

**Table 1 – Traceability Matrix**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Input/Output | Expected Result | Actual Result | Outcome  (Pass/Fail) |
| TC-01 | James 39 6 13 | Normalization of height | Name: James Age: 39 Height: 7’ 1” | Pass |
| TC-02 | Adam 34 6 | Adequate handling of invalid input | ArrayIndexOutOfBoundsException | Fail |
| TC-03 | Chris 21 6 6  Dan 28 7 2  Troy 21 6 6  Joe 22 6 3 | Output message indicating two players that meet the criteria | Tallest player whose age is less than the average is:  Name: Chris Age: 21 Height: 6' 6" | Pass\* (inconclusive due to two players with the same stats) |
| TC-04 | Blank- No input | “No player data entered.” Followed by program termination. | “No player data entered.” Followed by program termination. | Pass |

**Test Output Screenshots**

**TC-01**

**A screen shot of a computer

Description automatically generated**

**TC-02**

**A screenshot of a computer

Description automatically generated**

**TC-03**

A screenshot of a computer

Description automatically generated

**TC-04**

A screen shot of a computer

Description automatically generated

**Brief Note on Lessons Learned:**

During the development of the program, I gained insight into the power of object-oriented programming. The Height class, for instance (no pun intended), showcased the power of encapsulation by confining the logic necessary for height normalization to be confined within the Height class alone. This allowed for other classes, such as the Player class, to focus on managing player data, with the functionality of the toString method of the Height class in normalizing player height data without needing to concern itself with the logic behind the method. Encapsulation also allows for future modifications to specific aspects of the program should the need arise, without introducing bugs to the entire program, thanks to the separation of logic confined to separate classes within the program.

Furthermore, I learned of the importance of modularity. By being able to split the program up into separate parts, each class gained specialization – such as the Height class, which focused on processing height data and height normalization; the Player class, focusing on personal player data; the Player1 class, with finding the tallest player under the average height among all players, and the Main class tying the entire program together for program execution. Modularity allows for maintainable and dynamic programming.