Queen Mary University of London

Object Oriented Programming Project

Horse Race Simulation

ECS414U

Scenario

You have taken over a Java project previously maintained by Pony McRaceface, who is no longer available. Your task is to further develop a **horse race simulation**. The existing codebase includes two main classes, `Horse` and `Race`, which you can access on the same QM+ page as these instructions.

While the `Race` class is nearly complete, it contains some errors that require fixing. However, the `Horse` class has yet to be developed. Your goal is to complete and refine these components based on the following specifications:

`Horse` Class

The `Horse` class represents individual race participants. Each horse has:

1. **Name & Symbol:** Every horse has a unique name (used for announcing the winner) and a single-character symbol (used for race visualisation).
2. **Race Participation:** Horses can fall during the race, leading to their elimination.
3. **Confidence Rating:** Each horse has a confidence level between **0 and 1**:
    - A **higher confidence** means the horse runs faster but is also more prone to falling.
    - A **lower confidence** means the horse runs slower but is more stable.
4. **Performance Adjustment:**
    - Winning a race **slightly increases** a horse's confidence.
    - Falling during a race **slightly decreases** it.

`Race` Class

The `Race` class manages the simulation using `Horse` objects and provides a command-line visual representation of the race.

1. **Race Configuration:**
    a. The race length is configurable (measured in meters or yards).
    b. The number of lanes is adjustable, with some lanes potentially remaining empty.
2. **Race Execution:**
    - Horses are assigned to lanes before the race begins.
    - The race runs in real-time with an **animated display** in the command-line terminal.
    - For a reference, check `simulation.mov`
3. **Race Completion:** Once the race finishes, the results are displayed in the terminal.

Part I: Textual Racing Simulator

Task 1: Write the `Horse` Class to meet to the following specifications:

1. The class should contain the following five **fields** to store the relevant information:
   i. The name of the horse (e.g., "PIPPI LONGSTOCKING")
   ii. A single Unicode character that represents the horse (e.g., 🐴)
   iii. The distance travelled by the horse, stored as a whole number
   iv. A flag indicating whether the horse has fallen during the race.
   v. The confidence rating of the horse, represented as a decimal number between 0 and 1.

2. The **constructor** should have the following signature:

```
public Horse(char horseSymbol, String horseName, double horseConfidence)
```

3. The class should provide the following **public methods**:

   - `fall()`: Sets the horse as fallen.
   - `getConfidence()`: Returns the horse's confidence rating.
   - `getDistanceTravelled()`: Returns the distance travelled by the horse.
   - `getName()`: Returns the name of the horse.
   - `getSymbol()`: Returns the character used to represent the horse.
   - `goBackToStart()`: Resets the horse to the start of the race.
   - `hasFallen()`: Returns `true` if the horse has fallen, false otherwise.
   - `moveForward()`: Increments the distance travelled by the horse by 1.
   - `setConfidence(double newConfidence)`: Sets the confidence rating of the horse to the given value.
   - `setSymbol(char newSymbol)`: Sets the horse's symbol used to the specified character.

4. **Encapsulation** should be implemented to safeguard data in your class.

Important Note: It is crucial that the Horse class exactly matches the specifications provided. Any deviation could cause compatibility issues with the Race class written by McRaceface.

Report Requirements:

- **Encapsulation Explanation**: Explain how encapsulation is used to safeguard data in your class. Specifically, point out which methods are accessor (getter) methods, and which are mutator (setter) methods. You should explain how these methods work together to ensure that the horse's data is both accessible and protected from unwanted changes.
- **Testing Evidence**: Include screenshots of your tests with detailed explanations. Describe the tests you conducted to verify the correctness of each method, such as testing the `moveForward()` and `fall()` methods, or ensuring that confidence cannot exceed the allowed bounds (0 to 1).

Task 2: Improve the `Race` Class

Begin by thoroughly examining the `Race` class provided by McRaceface, which includes various features covered in the lectures.

a) The `startRace()` method serves as the `main` method. Enhance this method to display the name of the winning horse in the terminal once the race concludes.

b) Conduct thorough testing of the `Race` class to identify potential areas for improvement. Document any problems you encounter in the code and attempt to resolve as many as possible. You will receive credit for identifying issues, even if you are unsure of the solutions. Additionally, propose alternative solutions for the issues, whether or not you successfully implement them. Below is a sample snapshot for a three-horse race[1]:

```
==================
|         🏇       | PIPPI LONGSTOCKING (Current confidence 0.6)
|               🏇 | KOKOMO (Current confidence 0.6)
|           ❌[2]   | EL JEFE (Current confidence 0.4)
==================

And the winner is… KOKOMO!
```

In your report:

- List identified issues and explain them.
- Provide updated code for your `Race` class with explanations of changes.

[1] Additionally, a video version is available on the project's page.

[2] ❌ denotes a fallen horse

Part II: GUI Module Development

In this phase, you'll enhance your Racing application by introducing a user-friendly graphical user interface (GUI), building on the foundation laid in Part I. The GUI should include the following essential features:

1. **Interactive Track Design**: Allow users to design and customise racing tracks, including:
   i. *Customisable lane count*: Users should be able to select how many lanes are available on the track.
   ii. *Customisable track length*: Users can adjust the track's total length, either in meters or any other unit of choice.
   iii. *Configurable track shapes*: Users can choose the shape of the track, such as:
      a. Oval: Horses maintain consistent motion with gradual turns.
      b. Figure-eight: The track features intersecting paths, requiring horses to navigate sharp turns at specific points, affecting their movement dynamics (e.g., slowing down at the intersection).
      c. Other custom shapes: Allow the creation of arbitrary track shapes. The movement mechanics should adapt based on the shape chosen. For example, in a circular track, horses may move at a constant speed, while on a zigzag track, they would need to adjust speed more frequently, depending on sharpness of corners.
   iv. *Weather & Track Conditions*: Introduce different track conditions (e.g., muddy, dry, icy) that affect the horses' performance in distinct ways. These conditions should influence factors like:
      a. Speed: For example, a muddy track may slow down the horses, while a dry track might allow for faster speeds.
      b. Risk of falling: Some track conditions (like icy tracks) may increase the likelihood of a horse falling.
      c. Confidence: Horses' confidence ratings could be impacted by weather conditions (e.g., a higher risk of falling on an icy track could decrease a horse's confidence).

2. **Customisable Horses**: Provide users with a rich set of options to personalise their horses, moving beyond the simple letter symbol from Part I. The customisation should include the following features:
   i. *Breed Selection*: Allow users to choose from a predefined list of horse breeds (e.g., Thoroughbred, Arabian, Quarter Horse, etc.). Each breed could potentially affect the horse's attributes such as speed, stamina, or confidence.
   ii. *Coat Colour*: Users should be able to select from a range of coat colours (e.g., brown, black, grey, white, etc.), which can be visually represented on the horse during the race.
   iii. *Symbol Representation*: In addition to the letter symbol from Part I, allow users to select a unique visual representation of the horse, such as a custom character or emoji (e.g., 🐎, 🦄) to be displayed during the race.
   iv. *Equipment and Accessories*: Enable users to customise their horses with different types of equipment or accessories such as:
      a. Saddle: Option to choose between different saddle designs or colours.
      b. Horseshoes: Ability to select horseshoe styles, which could influence performance (e.g., regular, lightweight).
      c. Other accessories: Include items like bridles, blankets, or hats, which add visual flair and enhance the racing experience.
   v. *Attributes Impact*: Some customisation choices (e.g., breed, equipment) may affect the horse's performance. Specify how different elements (like breed or equipment) influence attributes such as speed, endurance, or confidence.

3. **Statistics and Analytics**: Implement a comprehensive statistics and analytics module to track and display detailed performance data for each race, horse, and betting activity. The statistics module should capture various aspects of each race and allow users to view, analyse, and interpret horse and race performance. The key features to be implemented are as follows:
   i. **Performance Metrics**: For each horse, calculate and display the following metrics after every race:
      a. **Average Speed**: Calculate the average speed of the horse throughout the race (considering the track length and the time taken to complete the race).
      b. **Finishing Time**: Display the total time taken by the horse to complete the race.
      c. **Win Ratios**: Track the number of wins versus the total number of races a horse has participated in, and calculate the horse's win percentage.

d. **Confidence Changes**: Track and display how the horse's confidence rating changes after each race, factoring in whether the horse won or fell.

ii. **Track Records**: Maintain a record of track-specific performance, including:
   a. **Best Times for Each Track**: Store and display the fastest finishing time for each track used in the race.
   b. **Track Conditions Impact**: Optionally, if track conditions affect performance (e.g., wet or dry), include the ability to display how those conditions may have impacted horse times or performance.

iii. **Historical Data**: For each horse, track and store historical race data, including:
   a. **Past Race Performances**: List each race the horse has participated in, showing details such as date, position finished, performance metrics (speed, time), and any incidents (e.g., falls).
   b. **Trends**: Track changes in performance over time, allowing users to identify patterns such as improving or declining performance.

iv. **User Interface**: Implement a clear, user-friendly interface where users can:
   a. **View Horse Statistics**: Allow users to view a summary of each horse's performance, including average speed, win ratio, and race history.
   b. **Compare Horses**: Provide functionality for users to compare multiple horses based on their performance metrics.
   c. **Graphical Display**: Optionally, include charts or graphs (e.g., bar charts, line graphs) to represent performance trends over time, such as speed or win ratios.

v. **Betting Analytics** (optional): Track betting history and provide insights into betting activity, including:
   a. **Betting Trends**: Show users how often they place bets on certain horses and whether they have been successful.
   b. **Betting Success Rates**: Display the user's success rate in betting (wins vs losses).

4. **Virtual Betting System**: Add an exciting betting feature to the horse racing game, where users can place virtual currency bets on the outcome of races. The betting system should allow users to evaluate horses and track conditions, and place informed bets based on real-time data. The system should track users' betting history, including winnings and losses, and offer insights to help improve their betting strategies.

Key Features:

i. **Betting Odds Algorithm**:

a. Develop an algorithm that calculates the odds for each horse in a race. The odds should be dynamically based on:

- **Horse Performance**: Use performance metrics like average speed, win ratios, and confidence rating to influence odds.
- **Track Conditions**: Factor in the type of track (e.g., wet or dry) and its effect on horse performance.
- **Recent Trends**: Consider the horse's past performances in similar conditions.

b. The odds should be updated regularly as conditions change (e.g., if a horse falls during the race or shows strong performance in practice).

c. **Odds Calculation Example**: For instance, a horse with high win ratios and good recent form may have low odds, whereas a horse with inconsistent performances may have high odds.

ii. **Real-Time Access to Odds**:

a. Users should be able to see the **current betting odds** for each horse before placing their bet.

b. The odds should be displayed in a **user-friendly format**, such as a list or table showing each horse and its respective odds.

c. The system must update the odds dynamically as race conditions change (e.g., if new data comes in, like a horse performing better than expected in practice).

iii. **Dynamic Odds Updates**:

a. After users place their bets, the odds should be updated in real-time, reflecting the following:

- **Betting Patterns**: If many users bet on a particular horse, its odds may increase to account for higher risk.
- **Track Conditions**: If the track changes during the race, odds should adjust accordingly.
- **Betting Activity**: Display a **live view of current betting trends**, such as the number of bets placed on each horse or total bet amounts, so users can make informed decisions.

iv. **User Interface for Betting**:

a. Create a **clear and easy-to-navigate betting interface** within the game, including the following elements:

- **Horse Selection**: A list or visual display of horses participating in the race. Users should be able to click on a horse or select a horse by name or symbol.
- **Bet Amount Input**: An input field where users can enter their desired bet amount, using virtual currency. The system should display available virtual currency balance and alert users if they attempt to bet more than they have.
- **Confirm Bet**: A button to confirm the bet, which also displays the calculated odds and potential winnings based on the bet amount.
- **Real-Time Betting Information**: Show live updates of total bet amounts for each horse, betting odds, and any changes in odds.

v. **Betting History and Feedback**:

a. Keep a record of all the **bets placed by the user**, including:

- **Amount Bet**: The amount of virtual currency wagered on each race.
- **Bet Result**: Whether the user won or lost the bet, along with the amount won or lost.
- **Cumulative Winnings**: Track and display the user's overall balance, accounting for both wins and losses.

b. **Feedback on Betting Strategies**:

- After each race, provide feedback to the user on their betting choices, e.g., "You won 3 out of 5 bets this week. Consider diversifying your bets next time."
- Include performance suggestions based on user betting history, such as "You tend to bet on horses with high win ratios. You may want to consider horses with higher odds for larger payouts."

Part III: Git Integration

Integrate Git into your project to track and manage your code efficiently.

1. Initialise Git Repository:
   i. Initialise a local Git repository for your project from the very beginning.
   ii. Name the root folder of your project `HorseRaceSimulator`.
2. Organise Project Structure:
   i. Structure your repository into two main folders:
      a. `Part1/`: Contains all the code for Part I (textual version).
      b. `Part2/`: Contains all the code for Part II (graphical version).

   Repository Structure:

```
HorseRaceSimulator/
│
├── .git/            # Git repository folder (hidden)
├── Part1/           # Code for Part I
└── Part2/           # Code for Part II
```

3. **Create GitHub Repository**:
   i. Create a **private repository** on GitHub to host your project code.
   ii. Push your local repository to GitHub.
4. **README File**:
   i. Include a **`README.md`** file in the root of your GitHub repository.
   ii. The `README` should include:
      a. Setup instructions (how to run the project locally).
      b. Dependencies required for the project.
      c. Usage guidelines.
5. **Commit Messages**:
   i. Use **descriptive commit messages** that clearly state the purpose of each change.
   ii. Example: “Fixed issue with Horse class constructor” or “Initial commit for Part 1: Horse class implementation”.
6. **Branching for GUI Development**:
   i. Create a new branch named `gui-development` to work on **Part II: GUI module**.

ii. Implement all changes related to the GUI module in this branch. This keeps the main branch intact and ensures that the graphical version doesn't interfere with the textual version.

7. **Merging the `gui-development` Branch**:

i. After completing the GUI module development, **merge the `gui-development` branch** back into the main branch.

ii. During the merge, resolve any conflicts that arise by reviewing the changes and deciding which version to keep.

.

Your submission

Your submission should be made through the OOP QM+ upload area designated for the Project. Make sure your code is written in a way that is IDE-agnostic, meaning it should compile and run using command-line tools without relying on any specific IDE features or configurations.

- The **textual version** of your Horse Race simulator should be run by calling the `startRace` method located in the **Part1** folder.
- The **graphical version** should be initiated using the `startRaceGUI` method located in the **Part2** folder.

Your submission must include the following:

1. A Report (PDF Format):
    i. Name the report `Report.pdf`
    ii. The report should provide a detailed response to **Part I** of the project.
2. A Zipped Working Directory:
    i. Zip the directory named `HorseRaceSimulator`, which contains your entire project.
    ii. This directory should include:
        i. The `.git` folder, representing the Git repository.
        ii. All source code files, including those in Part1 and Part2.
3. A README File:
    i. The README file must provide clear instructions on how to run your program.
    ii. Ensure that it includes:
        i. Step-by-step setup instructions, including any necessary dependencies.
        ii. Detailed information on how to compile and run both the textual and graphical versions of the simulator.
        iii. Explanation of how to invoke the relevant methods (`startRace` for Part 1, `startRaceGUI` for Part 2).
        iv. Any other relevant details that would help a user to run the program smoothly without requiring any special tools or IDE features.

Please ensure that the directory is complete, and all files required to run the project are included.

Assessment Criteria

Part I (60%)

Horse Class (40%)

1. **Correctness of Horse Class Definition (20%)**

- **Fields and Constructor (10%)**:
  - Correctly defines and implements the five required fields: name, symbol, distance, fall flag, and confidence.
  - Implements the constructor with the correct signature: `public Horse(char horseSymbol, String horseName, double horseConfidence)`.
- **Methods (10%)**:
  - All required methods are implemented correctly, ensuring each method performs its designated task (e.g., fall(), getConfidence(), moveForward(), etc.).
  - Adheres to the specifications for behaviour and signature, with proper functionality of methods like moveForward(), hasFallen(), etc.

2. **Encapsulation (5%)**

- **Encapsulation and Data Protection (5%)**:
  - Correctly implements getters and setters to provide controlled access to the fields, ensuring that encapsulation principles are followed.
  - Implements proper validation for confidence (ensuring it stays within the range 0 to 1) and other fields where necessary.

3. **Code Style and Conventions (5%)**

- **Adherence to Java Conventions (5%)**:
  - The code follows standard Java conventions (naming, indentation, and comments).
  - Variables and methods are named logically and consistently according to Java best practices.

4. **Testing and Validation (10%)**

- **Comprehensive Testing (5%)**:
  - Demonstrates a variety of tests, including edge cases (e.g., confidence bounds, movement, resetting after a fall).
- **Evidence of Testing (5%)**:
  - Provides screenshots or outputs of tests with clear explanations of the results and what the tests verify.

Race Class (20%)

1. **Improvements to Race Class (15%)**

- **Enhancements (10%)**:
  - Implements the feature to display the winner's name correctly at the end of the race.
  - Makes any necessary improvements to race logic, such as fixing bugs, improving race execution, or adding additional race-related functionality.
- **Code Quality and Structure (5%)**:
  - The Race class improvements are well-structured and efficient, ensuring readability and maintainability of the code.

2. **Bug Identification and Resolution (5%)**

- **Identifying Issues (3%)**:
  - Effectively identifies and explains issues in the provided Race class.
- **Proposing Solutions (2%)**:
  - Provides clear and logical solutions or suggestions for fixing identified problems.

Part II (20%)

1. **Interactive Track Design (5%)**

- **Track Customisation (2%)**
  - Users can configure the **number of lanes** and **track length** successfully.
  - Track length and lane count should update dynamically based on user input.
- **Track Shape Selection (2%)**
  - At least **three different track shapes** (e.g., Oval, Figure-eight, Custom) are available and affect horse movement appropriately.
  - Movement logic correctly adapts to different shapes (e.g., slowing on sharp turns).
- **Weather & Track Conditions (1%)**
  - Weather effects (e.g., muddy, icy) are implemented and impact **horse performance** correctly (e.g., speed reduction, confidence changes, risk of falling).

2. **Customisable Horses (5%)**

- **Horse Visual Customisation (2%)**
  - Users can select horse **breed, coat colour, and symbol representation** (e.g., emoji or custom character).
  - Visuals update correctly in the race display.
- **Equipment & Accessories (1%)**
  - Users can equip different saddles, horseshoes, and accessories that affect horse attributes.
- **Impact of Customisation (2%)**
  - Certain choices (e.g., breed, equipment) **affect performance** attributes like **speed, endurance, or confidence** in a meaningful way.

3. **Statistics & Analytics (5%)**

- **Performance Metrics (2%)**
  - The system correctly **calculates and displays** race performance data (e.g., **average speed, finishing time, win ratio**).
  - Confidence changes are tracked across races.
- **Track-Specific Records (1%)**
  - Best times and performance variations **based on track conditions** are recorded and accessible.
- **Historical Data & Comparisons (2%)**
  - Users can view past race data for each horse.
  - Optional comparison features (e.g., charts for trends in speed, confidence, or win ratio).

4. **Virtual Betting System (5%)**

- **Betting Odds Algorithm (2%)**
  - Odds dynamically reflect horse stats, **track conditions**, and recent trends.
  - Real-time updates occur as race conditions change.
- **User Betting Experience (2%)**
  - Users can place and confirm bets with a **clear interface** displaying odds, bet amounts, and potential winnings.
  - Betting history tracks past bets and outcomes.
- **Dynamic Betting Adjustments (1%)**
  - Betting trends influence odds in real time (e.g., if many users bet on a horse, its odds adjust).

Part III (20%)

1. **Repository Setup & Structure (4%)**

- **Correct Repository Initialisation (2%)**
  - A **local Git repository** was initialised at the start of the project.
  - The root folder is correctly named **HorseRaceSimulator**.
- **Organised Directory Structure (2%)**
  - Project is correctly structured into **Part1/** (textual version) and **Part2/** (graphical version).
  - No unnecessary files or folders are included in the repository.

2. **GitHub Repository & Remote Sync (4%)**

- **GitHub Repository Setup (2%)**
  - A **private GitHub repository** was created and linked to the local repository.
  - The initial push successfully transferred local files to GitHub.
- **Consistent Pushes & Remote Syncing (2%)**
  - Changes were regularly pushed to the GitHub repository to ensure version control.

3. **README Documentation (4%)**

- **Project Setup Instructions (2%)**
  - README includes clear **instructions** on how to run the project locally.
- **Dependencies & Usage Guidelines (2%)**
  - README lists all required **dependencies** and their installation steps.
  - Basic **usage instructions** are provided for users.

4. **Commit Quality & History (4%)**

- **Meaningful Commit Messages (2%)**
  - Commit messages are **clear, descriptive, and relevant** to the changes made.
  - Example: "Refactored Race class to improve track condition handling" (✅)
  - Poor example: "Updated file" or "Fixed stuff" (❌)
- **Frequent & Logical Commits (2%)**
  - Commits are **frequent and well-structured**, rather than large, irregular commits.

5. **Branching & Merging (4%)**

- **GUI Development Branch (2%)**
  - A separate **gui-development** branch was created for Part II.
  - All GUI-related changes were committed in this branch.
- **Successful Merging & Conflict Resolution (2%)**
  - The **gui-development** branch was properly merged into the **main branch**.
  - Any **merge conflicts** were resolved systematically.

For each criterion your work will be evaluated as:

Judgement	*Mark*	*Description*
Not attempted	0	No attempt was made.
Fail	1 - 3	Attempt made, but significant errors or omissions indicate insufficient understanding of the course material.
Basic	4 – 5	Overall correct work, demonstrating a basic understanding of the course material. May contain minor errors.
Good	6 – 7	Work reflects a deep understanding of the course material, accompanied by independent thinking and application. Errors, if any, are minor and infrequent.
Excellent	8 – 9	Work demonstrates a thorough understanding of the course material, complemented by independent research and critical thinking. Errors are minimal or non-existent.
Exceptional	10	Work surpasses the standard expected at first-year undergraduate level, showcasing outstanding comprehension, analysis, and originality.