

# HORSE RACE SIMULATOR REPORT - **ECS414U**

**Mohanprasad Baskaran**

**SID: 210054400**

# Contents

## **1. Horse Class**

|  |   |
|--|---|
| 1.1 Code.....  | 2 |
| 1.2 Explanation of Encapsulation in the Class.....         | 4 |
| 1.3 Identification of Accessor and Mutator Methods.....    | 4 |
| 1.4 Testing Process with Screenshots and Explanations..... | 5 |

## **2. Race Class**

|  |    |
|--|----|
| 2.1 Code.....  | 7  |
| 2.2 Explanation of Modifications.....                      | 10 |
| 2.3 Testing Process with Screenshots and Explanations..... | 12 |

# Horse Class

## 1.1 Code

```
1  /**
2   * The Horse class represents a horse participating in a race.
3   * Each horse has a name, symbol, confidence level, distance travelled,
4   * and a status indicating whether it has fallen during the race.
5   *
6   * @author Mohanprasad Baskaran
7   * @version Final 24/04/24
8   */
9  ✓ public class Horse {
10     private String name;
11     private char symbol;
12     private int distanceTravelled;
13     private boolean fallen;
14     private double confidence;
15
16     //Constructor
17  ✓ public Horse(char horseSymbol, String horseName, double horseConfidence) {
18         this.symbol = horseSymbol;
19         this.name = horseName;
20         this.confidence = Math.min(Math.max(horseConfidence, 0), 1);
21         this.distanceTravelled = 0;
22         this.fallen = false;
23     }
24     // Marks the horse as fallen.
25     public void fall() {
26         this.fallen = true;
27     }
28
29     // Retrieves the confidence level of the horse.
30     public double getConfidence() {
31         return confidence;
32     }
33
34     // Retrieves the distance travelled by the horse.
35     public int getDistanceTravelled() {
36         return distanceTravelled;
37     }
38
39     // Retrieves the name of the horse.
40     public String getName() {
41         return name;
42     }
43 }
```

```

43
44     // Retrieves the symbol of the horse.
45     public char getSymbol() {
46         return symbol;
47     }
48
49     // Resets the distance travelled and fallen status of the horse.
50     public void goBackToStart() {
51         this.distanceTravelled = 0;
52         this.fallen = false;
53     }
54
55     // Checks if the horse has fallen.
56     public boolean hasFallen() {
57         return fallen;
58     }
59
60     // Moves the horse forward if it hasn't fallen.
61     ✓ public void moveForward() {
62         if (!hasFallen()) {
63             distanceTravelled++;
64         }
65     }
66
67     // Sets the confidence level of the horse within the range [0, 1].
68     public void setConfidence(double newConfidence) {
69         this.confidence = Math.min(Math.max(newConfidence, 0), 1);
70     }
71
72     // Sets the symbol of the horse.
73     public void setSymbol(char newSymbol) {
74         this.symbol = newSymbol;
75     }
76 }

```

## **1.2 Explanation of Encapsulation in the Class**

Encapsulation is essential for safeguarding data; this was done by controlling the access to the class's properties through methods. The attributes (name, symbol, distanceTravelled, fallen, confidence) were declared as private which means that they can only be accessed from within the class itself, hence direct access from outside the class is not possible. This is used as a prevention for external code not being able to modify the state of the object without going through the respective methods. Public methods were provided to access and change the private attributes, for example getName() method allows for the classes to only receive name of the horse without modifying it directly. Encapsulation was also useful for range checking to be implemented in the class methods, for example the setConfidence() method ensures that the confidence level is restricted to be within the range of [0,1] using Math.min() and Math.max() methods, making sure the confidence level is always within this range. Encapsulation is also an important device to serve as keeping information hidden, for instance the details of the class implementation is hidden from external code, we only need to interact with Horse class without knowing the details of the implementation.

## **1.3 Identification of Accessor and Mutator Methods**

Accessor Methods:

1. getConfidence()
2. getDistanceTravelled()
3. getName()
4. getSymbol()
5. hasFallen()

Mutator Methods:

1. fall()
2. goBackToStart()
3. moveForward()

4. setConfidence(double newConfidence)
5. setSymbol(char newSymbol)

#### 1.4 Testing Process with Screenshots and Explanations

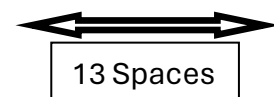
```
// Creates horses
Horse horse1 = new Horse(horseSymbol:'J', horseName:"JACKHORSE", horseConfidence:0.5);
Horse horse2 = new Horse(horseSymbol:'D', horseName:"DARKHORSE", horseConfidence:0.3);
Horse horse3 = new Horse(horseSymbol:'K', horseName:"KINGHORSE", horseConfidence:0.8);
```

Above shows the customised characteristics of the horse, consisting of the symbol of horse, name of horse and the confidence rating of the horse. This was tested to be successful as shown below, 'J', 'D' and 'K' can be seen in the race confirming that the correct symbols are being used. The name and confidence rating is also shown to be successful, for example 'JACKHORSE' and its Move Probability (confidence rating) is '0.5' just as inputted. Therefore, this deems the getConfidence(), setConfidence(), getName(), and getSymbol() methods as successful.

```
=====
|           J           |
|           D           |
|           K           |
|=====|
Move Probability for JACKHORSE: 0.5
Fall Probability for JACKHORSE: 0.025
Move Probability for DARKHORSE: 0.3
Fall Probability for DARKHORSE: 0.03499999999999996
DARKHORSE moved forward to 11!
Move Probability for KINGHORSE: 0.8
Fall Probability for KINGHORSE: 0.009999999999999998
```

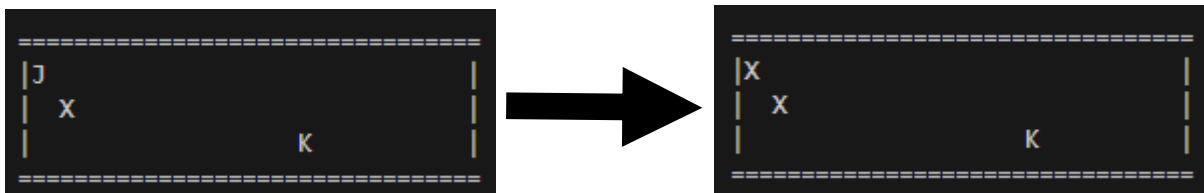
Distance travelled was also test and as shown in the example below, the distance is said to be 13, and when the spaces between the K and the start line was counted it was indeed 13, therefore getDistanceTravelled() was accurate.

```
Distance Travelled : 13
|           K           |
```



```
// Creates horses
Horse horse1 = new Horse(horseSymbol:'J', horseName:"JACKHORSE", -0.5);
```

Here the confidence rating has been set to a negative value, because of this the code automatically input the closest acceptable range which in this case is 0, and a value above 1 would be redirected to 1. For '-0.5' this means that the confidence rating is actually 0, this is shown below since, a confidence rating of 0 means that the move probability is also 0, hence 'J' stays in the same position until the fall probability is activated, the fall probability will remain as 0.05 since the move probability is 0. Therefore, you can see the 'J' stay in the same position and eventually fall off indicated by the 'X'.



```
Horse horse1 = new Horse(horseSymbol:'J', horseName:"JACKHORSE", horseConfidence:1.5);
```

When the confidence rating is set to a value above the range at '1.5' as mentioned above, it will be set to the nearest range which would be 1, but here when tested a issue is discovered, the higher the confidence the higher chance of the horse falling is the requirement but this is not the case when tested, due to the equation shown below. If the confidence rating is 1 the move probability is also 1 which is accurate, however since the fall probability is calculated by fall probability base (0.05) \* (1 – move probability), this means that the fall probability is 0.05 \* 0 which is 0, meaning is being calculated to never fall which contradicts the requirement. To fix this we can modify the acceptable range to be between 0 and 0.99, this will ensure that even if a value is above the range, when it gets set to 0.99, 1 – 0.99 will ensure that the fall probability base is multiplied by 0.01 instead of it being 0. We could also keep the range the same and change the equation into being fall probability = fall probability base \* (1.01 – move probability) which would result in the same fixture,

```
double fallProbability = FALL_PROBABILITY_BASE * (1 - moveProbability);
```

# Race Class

## 2.1 Code

```
1  /**
2   * The Race class simulates a horse race with multiple lanes.
3   * Horses compete to reach the finish line, with their progress
4   * determined by their confidence level and a probability of falling.
5   *
6   * @author Mohanprasad Baskaran
7   * @version Final 24/04/24
8   */
9
10 import java.util.concurrent.TimeUnit;
11 import java.lang.Math;
12
13 public class Race {
14     public static final double FALL_PROBABILITY_BASE = 0.05;
15     public int numberOfLanes = 3;
16     public int raceLength;
17     public Horse[] horses; // Array to manage horses
18
19     public Race(int distance) {
20         this.raceLength = distance;
21         this.horses = new Horse[3];
22     }
23
24     // Adds a horse to a specific lane in the race
25     public void addHorse(Horse theHorse, int laneNumber) {
26         if (laneNumber >= 1 && laneNumber <= horses.length) {
27             horses[laneNumber - 1] = theHorse;
28         } else {
29             throw new IllegalArgumentException("Lane number " + laneNumber + " is invalid.");
30         }
31     }
32     // Starts the race
33     public void startRace() {
34         for (Horse horse : horses) {
35             if (horse != null) horse.goBackToStart();
36         }
37
38         boolean raceFinished = false;
39         while (!raceFinished) {
40             int horsesStillInRace = 0;
41             for (Horse horse : horses) {
42                 if (horse != null) {
```



```

43         if (!horse.hasFallen()) {
44             moveHorse(horse);
45             horsesStillInRace++;
46         }
47         if (horse.getDistanceTravelled() >= raceLength) {
48             System.out.println("\nAnd the winner is " + horse.getName());
49             raceFinished = true;
50             break; // Breaks loop if a horse finishes the race
51         }
52     }
53 }
54
55 if (horsesStillInRace == 0) {
56     // If all horses have fallen, end the race.
57     raceFinished = true;
58     System.out.println("\nAll horses have fallen. Race over!");
59 }
60
61 printRace();
62
63 // Slow down the loop
64 try {
65     TimeUnit.MILLISECONDS.sleep(100);
66 } catch (InterruptedException e) {
67     Thread.currentThread().interrupt();
68     System.out.println("Race was interrupted.");
69     return;
70 }
71 }
72 }
73
74 // Moves the given horse forward and checks for falling probability
75 public void moveHorse(Horse theHorse) {
76     if (!theHorse.hasFallen()) {
77         // Print statements to check the probabilities
78         double moveProbability = theHorse.getConfidence();
79         double fallProbability = FALL_PROBABILITY_BASE * (1 - moveProbability);
80
81         System.out.println("Move Probability for " + theHorse.getName() + ": " + moveProbability);
82         System.out.println("Fall Probability for " + theHorse.getName() + ": " + fallProbability);
83
84         if (Math.random() < moveProbability) {
85             theHorse.moveForward();
86             System.out.println(theHorse.getName() + " moved forward to " + theHorse.getDistanceTravelled() + "!");
87         }
88         if (Math.random() < fallProbability) {
89             theHorse.fall();
90             System.out.println(theHorse.getName() + " has fallen!");
91         }
92     }
93 }
94

```

```

95     // Checks if the given horse has won the race
96     public boolean raceWonBy(Horse theHorse) {
97         return theHorse.getDistanceTravelled() >= raceLength;
98     }
99
100     public void printRace() {
101         clearConsole();
102         multiplePrint('=', raceLength + numberOfLanes);
103         System.out.println();
104
105         for (Horse horse : horses) {
106             if (horse != null) {
107                 printLane(horse);
108                 System.out.println();
109             }
110         }
111
112         multiplePrint('=', raceLength + numberOfLanes);
113         System.out.println();
114     }
115
116     // Clears the console to refresh the race display
117     public void clearConsole() {
118         for (int i = 0; i < 50; i++) {
119             System.out.println();
120         }
121     }
122
123
124     public void printLane(Horse theHorse) {
125         int spacesBefore = theHorse.getDistanceTravelled();
126         int spacesAfter = raceLength - theHorse.getDistanceTravelled();
127
128         System.out.print('|');
129
130         multiplePrint(' ', spacesBefore);
131
132         // Prints 'X' if horse is fallen
133         if (theHorse.hasFallen()) {
134             System.out.print('X');
135         } else {
136             System.out.print(theHorse.getSymbol());
137         }
138
139         multiplePrint(' ', spacesAfter);
140
141         System.out.print('|');
142     }
143
144     // Method to print a character multiple times
145     public void multiplePrint(char aChar, int times) {
146         for (int i = 0; i < times; i++) {
147             System.out.print(aChar);
148         }
149     }
150 }
151

```

## 2.2 Explanation of Modifications

```
public class Race {  
    private static final double FALL_PROBABILITY_BASE = 0.05;  
    private int numberOfLanes = 3;  
    private int raceLength;  
    private Horse[] horses; // Array to manage horses  
  
    public Race(int distance) {  
        this.raceLength = distance;  
        this.horses = new Horse[3];  
    }  
}
```

One of the major improvements that was implemented consists of now being able to modify the number of lanes and the length of the race which will be showcased in the testing section of this report. Originally, the code had each lane being represented by a separate variable 'lane1Horse' etc, this was limiting the race's properties since it would be extremely inconvenient to create a larger number of lanes. However now we use an array instead to manage the horses in the race as shown in 'Horse[] horses', this allows for a higher number of horses and using the parameter 'numberOfLanes' we can also determine the number of lanes. This ensures that we can modify the race to any number of horses and lanes enhancing the maintainability and efficiency of the code. In the new code, the methods are divided into smaller functions and variables are encapsulated using private access modifiers as shown through the code.

```
public void startRace() {  
    for (Horse horse : horses) {  
        if (horse != null) horse.goBackToStart();  
    }  
  
    boolean raceFinished = false;  
    while (!raceFinished) {  
        int horsesStillInRace = 0;  
        for (Horse horse : horses) {  
            if (horse != null) {  
                if (!horse.hasFallen()) {  
                    moveHorse(horse);  
                    horsesStillInRace++;  
                }  
                if (horse.getDistanceTravelled() >= raceLength) {  
                    System.out.println("\nAnd the winner is " + horse.getName());  
                    raceFinished = true;  
                    break; // Breaks loop if a horse finishes the race  
                }  
            }  
        }  
    }  
  
    if (horsesStillInRace == 0) {  
        // If all horses have fallen, end the race.  
        raceFinished = true;  
        System.out.println(x:"\nAll horses have fallen. Race over!");  
    }  
}
```

Before the lanes were hard coded, but ever since the improvement of the horses array we can use a loop to iterate over the array so that the number of lanes depends on the array size. This improved code has a clearer way of terminating the race, as if a horse completes the race the loop is instantly broken using break and the winner is also announced. The winner being announced is also an improvement.

```
private void moveHorse(Horse theHorse) {
    if (!theHorse.hasFallen()) {
        // Print statements to check the probabilities
        double moveProbability = theHorse.getConfidence();
        double fallProbability = FALL_PROBABILITY_BASE * (1 - moveProbability);

        System.out.println("Move Probability for " + theHorse.getName() + ": " + moveProbability);
        System.out.println("Fall Probability for " + theHorse.getName() + ": " + fallProbability);
    }
}
```

The probability aspect was improved to provide a more realistic horse racing simulation, the calculations are separated and made with clarity in mind. The probability is now based on the fall probability base value, which is 0.05, having a base allows the horse confidence level to have a much more affect on the race. This code also includes print statement to print the probabilities of the horse movement and falling.

```
private boolean raceWonBy(Horse theHorse) {
    return theHorse.getDistanceTravelled() >= raceLength;
}

private void printRace() {
    clearConsole();
    multiplePrint(aChar: '=', raceLength + numberOfLanes);
    System.out.println();

    for (Horse horse : horses) {
        if (horse != null) {
            printLane(horse);
            System.out.println();
        }
    }

    multiplePrint(aChar: '=', raceLength + numberOfLanes);
    System.out.println();
}
```

The code above simplifies the 'racewonBy' method by simply returning the result of 'theHorse.getDistanceTravelled() >= raceLength' which eliminated the need for using a if-else statement in the original code. The improvements made here replaces the number of lanes (3) originally from being hard coded to now being replaced by 'numberOfLanes' which makes the code way more adaptable and simpler to react to change made in the number of lanes. Instead of using print statements for each lane, we can just iterate the horses' array which make straightforward this process. The clear console method is used to clear the terminal window to make sure everything is clean.

### 2.3 Testing Process with Screenshots and Explanations

```
=====
|                               J |
|      X                          |
|                               K |
|                               |
=====
Move Probability for JACKHORSE: 1.0
Fall Probability for JACKHORSE: 0.0
JACKHORSE moved forward to 30!

And the winner is JACKHORSE
```

Tested the code to see if the improvements work, as shown in the image on the left the probabilities are printed, the distance of the horse that's the furthest is also printed successfully. And finally, the winner is correctly announced as per the requirements suggested.

```
=====
|                                     J |
| X                                     |
|                                     |
|           X                         |
|                                     |
=====
Move Probability for JACKHORSE: 1.0
Fall Probability for JACKHORSE: 0.0
JACKHORSE moved forward to 70!

And the winner is JACKHORSE
```

```
int raceLength = 70;
```

The length of the race was made to be '70' here, and the code correctly outputs this as shown by the terminal on the left, where you can see 'JACKHORSE moved forward to 70!'

```
// Creates horses
Horse horse1 = new Horse(horseSymbol:'J', horseName:"JACKHORSE", horseConfidence:0.5);
Horse horse2 = new Horse(horseSymbol:'D', horseName:"DARKHORSE", horseConfidence:0.3);
Horse horse3 = new Horse(horseSymbol:'K', horseName:"KINGHORSE", horseConfidence:0.8);
Horse horse4 = new Horse(horseSymbol:'P', horseName:"PURPLEHORSE", horseConfidence:0.6);
Horse horse5 = new Horse(horseSymbol:'L', horseName:"LOSEHORSE", horseConfidence:0.1);

// Add horses to the race
race.addHorse(horse1, laneNumber:1);
race.addHorse(horse2, laneNumber:2);
race.addHorse(horse3, laneNumber:3);
race.addHorse(horse4, laneNumber:4);
race.addHorse(horse5, laneNumber:5);
```

Here the tests were conducted to see how my code deals with more lanes whilst keeping the length long, therefore 2 more horses were created and added to the race, the number of lanes was also increased to 5. Since the lanes and the lengths were longer, it was apparent that a horse finishing a race was less likely due to the length much longer.

```
=====
|               X               |
|      X               |
|               X               |
|                               P |
|X                               |
=====

Move Probability for PURPLEHORSE: 0.6
Fall Probability for PURPLEHORSE: 0.020000000000000004
PURPLEHORSE has fallen!
```

```
private int numberOfLanes = 7;
```

```
=====
|               X               |
|           X                   |
|                   X           |
|       X                   K   |
| X                           |
|=====
Move Probability for KINGHORSE: 0.8
Fall Probability for KINGHORSE: 0.009999999999999998
KINGHORSE moved forward to 70!

And the winner is KINGHORSE
```

Error testing was conducted here to see how the program reacts to the number of lanes being longer than necessary, we inputted the same 5 horses but inputted '7' as the number of lanes, which is incorrect information. The image on the left shows that the code adjusted and ignored the extra lanes and only created 5 lanes for the 5 horses.

```
int raceLength = 10;

// Creates a race
Race race = new Race(raceLength);

// Creates horses
Horse horse1 = new Horse(horseSymbol:'J', horseName:"JACKHORSE", horseConfidence:0.5);
Horse horse2 = new Horse(horseSymbol:'D', horseName:"DARKHORSE", horseConfidence:0.3);
//Horse horse3 = new Horse('K', "KINGHORSE", 0.8);
```

```
=====
|      X      |
|      D      |
|      |      |
=====
Move Probability for DARKHORSE: 0.3
Fall Probability for DARKHORSE: 0.03499999999999996
DARKHORSE moved forward to 10!

And the winner is DARKHORSE
```

The horses were minimised to '2' horses and the race length was minimised as '10', this created the following output in the terminal, here often due to probability the most confident horse would complete the race.

```
Run | Debug
public static void main(Strin
int raceLength = 1000;
```

When the race length is put to a very high number such as '1000', the code still runs and creates a messy terminal output, introducing a range of values can solve this issue similar to the set confidence we can do a set race length method in the horse class.

### Mistakes and future improvements

First thing that comes to mind is the use of hardcoding the number '3' for the number of lanes and using that inside the array for horses, instead the 'numberOfLanes' should be place inside the array so only once manually it's needed for modification. To improve this race class further, instead of using a simple array, introducing an arrayList can be very beneficial, as there would be no length limitations and can be easily integrated with the other classes. Currently my code throws a generic IllegalArgumentException, instead I should provide information to the user on why a specific error is occurring, this would improve the overall usability of my code making it easier for users to identify errors.