Horse Racing Simulator Report.

Horse Racing Simulator Report. 1
Development. 4
Horse Constructor. 4
Starting to run the program. 9
Showing lanes. 10
Showing the race in the terminal. 11
Checking for a winner. 12
Moving horses. 12
Checking if race is over 13
Showing the winner. 14
Resetting the horse positions. 15
User Can add their own horses. 16
Showing race details. 17
Handling User Input. 18
User Input Interface. 19
User Options Interface. 20
User Input Class and User Input Validation. 23
Customising the simulation between races. 31
Removing lanes. 31
Adding horses back into correct lanes. 32
Adding lanes. 32
Removing horses. 33
Show full lanes. 35
Add horses. 35
Show empty lanes. 36
Check if all lanes are full. 37
Horse success variables. 37

Win-rate, total races, total races. 38
Getters and setters. 39
Update horse success variables after the race. 42
Unique horse names. 43
File Handling. (Saving Horse Data) 43
Saving horse details. 44
CSV file. 45
Making the CSV into a table for user. 45
Horse Details Table 49
Ask User if they want to save a horse. 49
Showing the horses in the race they can choose to save. 50
Retrieving Horses from the File. 51
Horse Constructor made for String input from file. 53
Updating Horses in the File After Race. 54
Random Horses. 55
Testing Random Horses. 57
Weather Conditions. 59
Weather affects confidence 61
Betting System 61
Betting Class. 62
Betting attributes getters and setters. 63
More horse attributes for betting. 63
Attributes. 64
Getters and Setters. 64
Ask user to place bets. 64
Update to show winner method (to show bets and update balance). 67
Remove bets once race is over 67
Timing the race. 68
Race timer 68

While loop which controls the race. 70
Final Testing. 70
Displaying the winner. 70
Selecting track length. 71
Testing Confidence and Confidence Bounds. 71
User adding their own horses. 73
Empty Lanes. 73
Edge Cases for lane choice and horse choice. 74
Random horses added to race. 75
Horses added from file to race. 77
Horsed added from race to file. 78
Horse confidence affected by weather. 79
Horse confidence affected by losing/ winning race. 80
Lanes removed. 82
Lanes added. 83
Horses removed. 84
Horses added. 85
Changing the length of the race. 86
Betting 86
Bets lost. 87
Bets won. 87
Betting amount edge cases. 88
Horse symbol must be a singular character. 89
Race timer. 89
Testing Conclusion: 91
Java Conventions 92
Bug Identification and Resolution 92
Improvements to Race Class 93
Conclusion: 94

Development.

Horse Constructor.

I first started this project by starting my work on the Horse class. I completed the constructor and the getter and setter methods for all the private attributes which belong to the horse class. These attributes included the name of the horse and the character/ symbol for the horse and the confidence level of the horse. The horse confidence is a double which is in between 0 and 1. I applied this restriction to the horse confidence by setting it to 1 if it happens to go above 1 for any reason and setting it to 0 if it ever goes below 0. This ensures that it always is still between 0 and 1 throughout the lifetime of the program. The horse also has an attribute for the distance travelled which is a whole number, this also needs a getter and setter method as this is used to display the horse at the correct place on the track and it's also needed to check which horse has won the race. Furthermore, this needs to be reset after every race, so all horses start the race from the start point. Horses are also able to fall during the race, so I need a Boolean variable to keep track of this. This also requires getter and setter methods so they can be accessed outside of the horse class. If a horse falls during the race or it doesn't finish the race first, then I have implemented a feature which decreases the confidence by 10% or increases the confidence by 10% if they win (more on this later). Also to add horses to the lanes I have added an attribute which gives horses a lane number, this also allows for empty lanes later in the program.

```
// Fields of Horse
private char horseSymbol;              // The character symbol representing the horse
private final String horseName;        // The name of the horse
private double horseConfidence;        // The confidence rating of the horse (0.0 to 1.0)
private int horseDistance;             // The distance the horse has traveled in the race
private boolean horseFallen;           // Flag indicating whether the horse has fallen
private int laneNumber;                // The lane number assigned to the horse
```

And now the constructor looks like this.

```
public Horse(String horseName, double horseConfidence, char horseSymbol, int laneNumber) {
    this.horseName = horseName;
    // Ensure confidence is within the valid range [0.0, 1.0]
    if (horseConfidence > 1.0) {
        this.horseConfidence = 1.0;
    } else if (horseConfidence < 0.0) {
        this.horseConfidence = 0.0;
    } else {
        this.horseConfidence = horseConfidence;
    }
    this.horseSymbol = horseSymbol;
    this.laneNumber = laneNumber;
    this.horseDistance = 0;
    this.horseFallen = false;

}
```

All of these attributes also have getters and setter methods which are also shown below.

```java
/**
 * Marks the horse as fallen.
 */
public void fall() {
    this.horseFallen = true;
}

/**
 * Returns the lane number assigned to the horse.
 *
 * @return the lane number of the horse.
 */
public int getLaneNumber() {
    return this.laneNumber;
}

/**
 * Sets the lane number for the horse.
 *
 * @param newLane the new lane number to assign to the horse.
 */
public void setLaneNumber(int newLane) {
    this.laneNumber = newLane;
}

/**
 * Returns the confidence rating of the horse.
 *
 * @return the confidence rating of the horse (0.0 to 1.0).
 */
public double getConfidence() {
    return this.horseConfidence;
}
```

The fall method returns the Boolean private attribute which decides if the horse has fell or not. The get lane number method returns the lane number of the horse. The set lane number returns the lane number of the horse, and the get confidence method returns the horse confidence of the horse.

And more below.

```java
/**
 * Returns the distance the horse has traveled.
 *
 * @return the distance traveled by the horse.
 */
public int getDistanceTravelled() {
    return this.horseDistance;
}

/**
 * Returns the name of the horse.
 *
 * @return the name of the horse.
 */
public String getName() {
    return this.horseName;
}

/**
 * Returns the character symbol representing the horse.
 *
 * @return the symbol of the horse.
 */
public char getSymbol() {
    return this.horseSymbol;
}

/**
 * Resets the horse's position to the start of the race.
 * The distance traveled is set to 0, and the horse is marked as not fallen.
 */
public void goBackToStart() {
    this.horseDistance = 0;
    this.horseFallen = false;
}
```

The get distance travelled method returns the distance which the horse has travelled. The get name method returns the name of the horse. The get symbol method returns the character representation of the horse. The get back to start method returns the horse to the start line by making sure that the horse has not fallen and the distance travelled is 0.

```java
/**
 * Determines whether the horse has fallen during the race.
 * This state affects the horse's ability to continue moving forward.
 *
 * @return true if the horse is in a fallen state, false if the horse remains upright
 */
public boolean hasFallen() {
    return this.horseFallen;
}

/**
 * Advances the horse's position by one unit along the race track.
 */
public void moveForward() {
    this.horseDistance += 1;
}

/**
 * Updates the horse's confidence rating with value
 * Confidence values are constrained to the range [0.0, 1.0] and rounded to 2 decimal places
 *
 * @param newConfidence The new confidence level
 */
public void setConfidence(double newConfidence) {
    // Clamp value to valid range with informative messaging
    if (newConfidence > 1.0) {
        this.horseConfidence = 1.0;
    } else if (newConfidence < 0.0) {
        this.horseConfidence = 0.0;
    } else {
        // Round to 2 decimal places for consistency in simulation calculations
        this.horseConfidence = Math.round(newConfidence * 100.0) / 100.0;
    }
}
```

The set confidence method above clearly shows what I have said before. The confidence level must be constrained to number [0-1]. So, if it goes above 1 it is reset back to 1 and if it goes below 0 then it is set back to 0. Also, I have rounded the numbers to two decimal places to make it easier to read throughout development and testing.

And the last method is the method which sets the symbol of the horse on the track, this method shows the single ASCII/ Unicode representation of the horse displayed on the

track during the race.

```
/**
 * Updates the visual representation symbol for the horse.
 * The symbol should be a single Unicode character that will be displayed
 *
 * @param newSymbol The character to use for race display purposes
 */
public void setSymbol(char newSymbol) {
    this.horseSymbol = newSymbol;
}
```

- **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.

All the methods for the horse class have been added but more will have added throughout development for extra functionality and features. All the methods shown up until now have been tested and they all preform the tasks which they are intended to be doing.

- **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.

All attributes in the horse class are private, and every attribute has a getter and setter methods encapsulation of data is as important as the validation of data. I have confirmed the confidence in the horse method as well but during the development of the program when I develop user input, I will have methods dedicated to data handling and confirming and error checking.

Starting to run the program.

Now that the Horse class has some functionality, I decided to create a Class Main with a main function to create an instance of race to run the program.

The next thing I decided to do was put the Horses in Array Lists this decision was made so that it is easy to add horses anywhere and it is easy to remove horses from anywhere. Although the Array List is called ‘horses’ the null indexes within the Array List are empty lanes within the program as they are skipped.

Showing lanes.

Because of this I had to make some changes to the print Lane method, if the Horse given to the method is null then the empty lane is still printed, and I will show an empty lane next to the lane. Otherwise, if there is a horse then I print the lane and the horse and next to the lane on the right-hand side I will display the lane number and the name of the horse and the confidence of the horse. This is so that it easy to keep track of. The array of horse's index from 0 is the lane number of the horse I add one to the index. The edited method is shown below.

```
/**
 * Prints a single lane with the horse's current position.
 *
 * @param theHorse the horse in the lane to print.
 */
private void printLane(Horse theHorse) {
    if (theHorse == null) {
        System.out.print(c:'|');
        multiplePrint(aChar:' ', raceLength+1);
        System.out.print(c:'|');
        System.out.print(s:" Empty Lane");
    } else {
        int spacesBefore = theHorse.getDistanceTravelled();
        int spacesAfter = raceLength - theHorse.getDistanceTravelled();

        System.out.print(c:'|');
        multiplePrint(aChar:' ', spacesBefore);

        if (theHorse.hasFallen()) {
            System.out.print(c:'\u2322');
        } else {
            System.out.print(theHorse.getSymbol());
        }

        multiplePrint(aChar:' ', spacesAfter);
        System.out.print(c:'|');
        System.out.print(" Lane: "+theHorse.getLaneNumber());
        System.out.print(", "+theHorse.getName()+" (Current Confidence "+theHorse.getConfidence()+")");
    }
}
```

Within this method there are other method which are needed to print multiple characters this method is shown below.

```java
/**
 * Prints a character multiple times.
 *
 * @param aChar the character to print.
 * @param times the number of times to print the character.
 */
private void multiplePrint(char aChar, int times) {
    for (int i = 0; i < times; i++) {
        System.out.print(aChar);
    }
}
```

Now with these two methods I was able to display the horses running, but for this a while loop was used, and the condition was only stopped once a horse has finished the race. The race is finished if and only if all horses have fallen or one has finished the race.

```java
/**
 * Prints all lanes in the race.
 */
private void printAllLanes() {
    for (Horse horse : horses) {
        printLane(horse);
        System.out.println();
    }
}
```

The method above prints the lane for all the horses within the race. This is done using a for loop, for each horse in horses I give the horse to the print Lane method to display the lane.

Showing the race in the terminal.

```
/**
 * Prints the current state of the race.
 */
private void printRace() {
    System.out.print(c:'\u000C');
    multiplePrint(aChar:'=', raceLength + 3);
    System.out.println();
    printAllLanes();
    multiplePrint(aChar:'=', raceLength + 3);
    System.out.println();
}
```

And this is the method which call the earlier method to draw the rest of the race, this method is called during the while loop which controls the race.

Checking for a winner.

To check if a horse has won the race, I need a method which compares the length of the race with the position of the horse, in other words in compares the length of the race with the distance the horse has travelled this method is shown below.

```
/**
 * Determines if a horse has won the race.
 *
 * @param theHorse the horse to check.
 * @return true if the horse has won, false otherwise.
 */
private boolean raceWonBy(Horse theHorse) {
    return theHorse.getDistanceTravelled() == raceLength;
}
```

This method will return true, and this will show that the race is over.

Moving horses.

To move the horses once the race has started this is the method which controls that.

```java
/**
 * Moves a horse forward or makes it fall based on its confidence.
 *
 * @param theHorse the horse to be moved.
 */
private void moveHorse(Horse theHorse) {
    if (!theHorse.hasFallen()) {
        if (Math.random() < theHorse.getConfidence()) {
            theHorse.moveForward();
        }

        if (Math.random() < (0.1 * theHorse.getConfidence() * theHorse.getConfidence())) {
            theHorse.fall();
        }
    }
}
```

If the horse has not fell, then a random number will decide of the horse is allowed to move forward by one unit. A greater confidence means that there is a higher chance of the horse moving, but a higher confidence will also mean the horse has a greater chance of falling.

Checking if race is over.

```java
/**
 * Checks if the race is finished.
 *
 * @return true if a horse has won or all horses have fallen, false otherwise.
 */
private boolean raceFinished() {
    boolean allFell = true;
    for (Horse horse : horses) {
        if (horse == null) continue;
        if (raceWonBy(horse)) {
            return true;
        }
        if (!horse.hasFallen()) {
            allFell = false;
        }
    }
    return allFell;
}
```

One of the other ways the race could finish is if all the horses end up falling over before finishing the race. So, this method uses the earlier method shown to check if any horse has won while checking to make sure that there is at least one horse still running.

Below is the while loop which I have been referring to.

```
while (!finishedRace) {
    moveAllHorses();
    printRace();
    finishedRace = raceFinished();

    // Wait bwtween horse moves
    try {
        TimeUnit.MILLISECONDS.sleep(timeout:100);
    } catch (InterruptedException e) {}
} // END RACE
```

This moves all horses until one of the horse wins or all horses fall all while printing the race to the terminal.

After all these changes the Horse are now able to complete a full race. In this example run the horse called ‘Horse 4’ with symbol ‘4’ has won the race. This example also shows the empty lanes. This is because there are 6 lanes and only 4 horses. So, 2 lanes have been able to remain empty. In this example a horse has also fallen, in lane 4 the horse called ‘Horse 3’ has fell less than halfway through the race and this is shown by the horse symbol now being ‘?’.

```
===========================
|                   1      | Lane: 1, Horse 1 (Current Confidence 0.06)
|                      2   | Lane: 2, Horse 2 (Current Confidence 0.06)
|                          | Empty Lane
|         ?                | Lane: 4, Horse 3 (Current Confidence 0.06)
|                          | Empty Lane
|                         4| Lane: 6, Horse 4 (Current Confidence 0.06)
===========================
```

Showing the winner.

The next step now is to display the winning horse to the screen for the user to see.

```java
/**
 * Displays the winner of the race and adjusts the confidence of the horses.
 */
private void showWinner() {
    for (Horse horse : horses) {
        if (horse == null) continue;
        if (raceWonBy(horse)) {
            System.out.println("\n\n"+horse.getName()+" has won the Race!");
            horse.setConfidence(horse.getConfidence()*1.1);
        } else {
            horse.setConfidence(horse.getConfidence()*0.9);
        }
    }
}
```

The method above now displays the winning horse and increases the confidence by 10% which I mentioned previously. The all the horses which did not win get their confidence decreased by 10%.

Resetting the horse positions.

Once the race is over if the user wants to play again then they will need to horses' positions to be reset back to the start, this is done by this method, which is called before every race to ensure the horse positions are all 0 for a fair race.

```java
/**
 * Resets the position of all horses to the start line.
 */
private void resetHorsesPosition() {
    for (Horse horse : horses) {
        if (horse == null) continue;
        horse.goBackToStart();
    }
}
```

The next step was to allow the user to add their own horses into the race. So, they need to be able to customize the race.

User Can add their own horses.

```
/**
 * Initializes the horses and lanes for the race.
 */
private void createHorses() {
    int inputLanes = chooseNumberOfLanes(statement:"How many lanes would you like [2 - 8]: ");
    for (int numberOfLanes = horses.size(); numberOfLanes < inputLanes; numberOfLanes++) {
        horses.add(numberOfLanes, element:null);
    }
    int inputHorses = chooseNumberOfHorsesGivenNumberOfLanes(statement:"How many horses would you like: ", inputLanes);
    inputHorses = inputHorses - Horse.horseCounter;
    for (int numberOfHorses = 0; numberOfHorses < inputHorses; numberOfHorses++) {
        int lane = pickOneOfTheLanes("\nChoose a lane to add this horse to [1 - "+inputLanes+"]: ", inputLanes);
        while (horses.get(lane-1) != null) {
            System.out.println("Lane "+lane+" is taken by another horse.");
            lane = pickOneOfTheLanes("\nChoose a lane to add this horse to [1 - "+inputLanes+"]: ", inputLanes);
        }
        horses.set(lane-1, createHorse(lane));
    }
}
```

The method to create Horses ask the user how many lanes they want. The number of lanes is restricted from 2 to 8. The reason 2 is the least number of lanes is because I want to make it so that at least 2 horses must start the race. Then this will create the same number of lanes the user asked for by adding null elements to the horses Array List. Then the next question the user is asked is how many horses they want. This is restricted by the number of lanes which they chose. So, if the user has chosen to have 5 lanes, they cannot have 8 horses. So, an example which would be valid is to have 6 lanes and 3 horses and this would mean 3 lanes remain empty. Then they can choose which lane they want to add each horse to. Once they chose a valid lane which is not out of range or is not take then a horse will be created and added to the index which is lane-1.

The methods used for user input will be shown and discussed later in this document.

Now that the user can decide how many lanes and how many horses they want let's look at the method which creates these horses.

```java
/**
 * Creates a new horse with the specified name, character, and lane.
 *
 * @param lane the lane number where the horse will be placed.
 * @return the created Horse object.
 */
private Horse createHorse(int lane) {
    String name = inputString(statement:"\nHorse Name: ");
    while (usedName(name)) {
        name = inputString(statement:"Horse Name: ");
    }
    char character = inputCharacter(statement:"Horse Character: ");
    return new Horse(name, horseConfidence:0.25, character, lane);
}
```

To create the horse, I need the user to input the name and symbol of the horse which they would like to use. All horses initially will be given a confidence of 0.25 which will increase or decrease based on performance and the character and lane are passed to the constructor.

Showing race details.

At the end of each race, I also want to be able to display the details of each horse as well as the details of the race. This includes the number if horses in the race, the number of lanes in the race and the length of the race as well as all the details of each horse.

This method is shown below.

```java
/**
 * Displays the current race details including length, number of lanes, and horses.
 */
private void showRaceDetails() {
    System.out.println("\nCurrent length of the race: "+raceLength);
    System.out.println("Current number of lanes: "+horses.size());
    System.out.println("Current number of horses: "+Horse.horseCounter+"\n\n");
    for (Horse horse : horses) {
        if (horse == null) continue;
        System.out.print(horse.getName()+" is in lane "+horse.getLaneNumber());
        System.out.println(", Confidence: "+horse.getConfidence());
    }
    System.out.println(x:"\n");
}
```

This is method is just outputs all the essentials for the user to see.

Handling User Input.

Quite a lot of code has been written so far here is a breakdown of what has been done: The user can now choose how many lanes which they would like to have in the race, then they can choose how many horses they would like in the race, this allows for empty lanes and custom horses made by the user, then the race will start, when the race is over the race details will be displayed and the winner will be displayed, if there is no winner (meaning that all horses fell) then the race will also end. After all this has happened the horse confidence will be adjusted depending on performance.

Now let's move on to how the user input is handled. I have created a user input interface and a user option interface. These two interfaces are implemented by the user input class. The input interface is for the methods which ask for an input such as asking for a number, asking for a string or asking for any type of data type, this interface also has methods to confirm these data types, to make sure they are valid. The user option interface is for when the user is given an option when they are asked a question, so if they are asked a yes or no question or when they have a choice. So, if they have created a group of horses but they want to access a specific one then they will have a choice.

The user input is essential. This is because between races I want the user to be able to change the length of the race, remove horses, add horses, remove lanes and add lanes.

I also want the user to be able to save horse details in files (This will be covered later in this document).

Firstly, I will show the user input interface.

User Input Interface.

```
interface UserInputInterface {

    /**
     * Prompts the user to input a number and validates it.
     *
     * @param statement the prompt message to display to the user.
     * @return the validated integer input from the user.
     */
    public int inputNumber(String statement);

    /**
     * Prompts the user to input a number (double) and validates it.
     *
     * @param statement the prompt message to display to the user.
     * @return the validated double input from the user.
     */
    public double inputDouble(String statement);

    /**
     * Prompts the user to input a single character and validates it.
     *
     * @param statement the prompt message to display to the user.
     * @return the validated character input from the user.
     */
    public char inputCharacter(String statement);

    /**
     * Prompts the user to input a string.
     *
     * @param statement the prompt message to display to the user.
     * @return the string input from the user.
     */
    public String inputString(String statement);
```

This is from the interface for the user input. This shows for all the input, integers, double, character and string. The input statement is the prompt the user is given to give an input to.

The rest of the interface is shown below.

```java
/**
 * Validates if the input string represents a valid number.
 *
 * @param input the string to validate.
 * @return true if the input is a valid number, false otherwise.
 */
public boolean validateNumber(String input);

/**
 * Validates if the input string represents a valid single character.
 *
 * @param input the string to validate.
 * @return true if the input is a valid single character, false otherwise.
 */
public boolean validateCharacter(String input);

/**
 * Validates if the input string represents a valid double.
 *
 * @param input the string to validate.
 * @return true if the input is a valid double, false otherwise.
 */
public boolean validateDouble(String input);
```

This set of methods are needed for the earlier method; these methods are the validation methods for all the other methods which were shown. These are needed so the user cannot do invalid inputs. If the program requires a double input a String will not be accepted unless it is a double. All inputs are Strings initially and only once they are valid then they get casted into another data type.

User Options Interface.

Before I show the implementation of these methods, I will show the second interface for the options which the user is given. These options will make more sense once the user can edit the race details in-between races. This will be later in the document. The methods in the options interface are all needed to change race details so the implementation will be explained for each of them one by one later.

So, the options interface is shown below.

```java
interface UserOptionInterface {

    /**
     * Prompts the user with a yes/no question and validates the response.
     *
     * @param statement the prompt message to display to the user.
     * @return true if the user selects "yes" (1), false if the user selects "no" (2).
     */
    public boolean askYesNo(String statement);

    /**
     * Validates if the input is a valid yes/no response.
     *
     * @param input the user's input to validate.
     * @return true if the input is "1" or "0", false otherwise.
     */
    public boolean yesNo(String input);

    /**
     * Prompts the user to choose the length of the race track and validates the input.
     * The track length must be within a valid range (e.g., 25 to 100 units).
     *
     * @param statement the prompt message to display to the user.
     * @return the validated track length chosen by the user.
     */
    public int chooseTrackLength(String statement);

    /**
     * Prompts the user to choose the number of lanes for the race and validates the input.
     * The number of lanes must be within a valid range (e.g., 2 to 8 lanes).
     *
     * @param statement the prompt message to display to the user.
     * @return the validated number of lanes chosen by the user.
     */
    public int chooseNumberOfLanes(String statement);
```

```java
/**
 * Prompts the user to choose any of the horses.
 * Should be used for picking a specific horse.
 *
 * @param statement the prompt message to display to the user.
 * @return the validated horse lane number from the user.
 */
public int pickAnyHorse(String statement);

/**
 * Prompts the user to choose the number of horses for the race and validates the input.
 * The number of horses must be within a valid range (e.g., 2 to 8 horses).
 *
 * @param statement the prompt message to display to the user.
 * @return the validated number of horses chosen by the user.
 */
public int chooseNumberOfHorses(String statement);

/**
 * Prompts the user to choose the number of horses given a specific number of lanes and validates the input.
 * The number of horses must not exceed the number of lanes.
 *
 * @param statement the prompt message to display to the user.
 * @param numberOfLanes the number of lanes available for the race.
 * @return the validated number of horses chosen by the user.
 */
public int chooseNumberOfHorsesGivenNumberOfLanes(String statement, int numberOfLanes);

/**
 * Prompts the user to pick a lane from the available lanes and validates the input.
 * The lane number must be within the range of available lanes (1 to max).
 *
 * @param statement the prompt message to display to the user.
 * @param max the maximum number of lanes available.
 * @return the validated lane number chosen by the user.
 */
public int pickOneOfTheLanes(String statement, int max);
```

Once these methods get implemented changes can be made to the race during run time and the user can be asked to play again and simulate race after race without the need to close the program.

User Input Class and User Input Validation.

Here is the first method which I showed earlier. This is the implementation of it. A statement is passed to the method this is what the user will read before putting in the input. Then the user will input a string as an answer, and this will be confirmed while the input is invalid the user will be asked the question again until they give a valid input. In this case I was asking for an Integer input so it must be a whole number. Once I know that it is a valid input I can return the integer. The validation method implementation will also be shown below.

```java
public class UserInput implements UserOptionInterface, UserInputInterface {

    /**
     * Prompts the user for a number input and validates it.
     *
     * @param statement the prompt message to display to the user.
     * @return the validated integer input from the user.
     */
    @Override
    public int inputNumber(String statement) {
        String input = inputString(statement);
        boolean valid = validateNumber(input);
        while (!valid) {
            input = inputString(statement);
            valid = validateNumber(input);
        }
        return Integer.parseInt(input);
    }
```

```java
/**
 * Validates if the input string is a valid number.
 *
 * @param input the string to validate.
 * @return true if the input is a valid number, false otherwise.
 */
@Override
public boolean validateNumber(String input) {
    if (input.length() == 0) return false;
    for (int i = 0; i < input.length(); i++) {
        char x = input.charAt(i);
        if (x < '0' || x > '9') return false;
    }
    return true;
}
```

If the length of the input is equal to 0 then it is invalid return false, then we must check that each character in the input is a number to force the input to be a number. If we find any character to no be a number, then we return false. If we pass all these checks then we know we have a valid input and we return true and move on.

```java
/**
 * Prompts the user for a string input.
 *
 * @param statement the prompt message to display to the user.
 * @return the string input from the user.
 */
@Override
public String inputString(String statement) {
    @SuppressWarnings("resource") // Resource Leak: scanner never closed
    Scanner scanner = new Scanner(System.in);
    System.out.print(statement);
    return scanner.nextLine();
}
```

The method above is the method which does all the input. All inputs are taken in as string for validation purposes. This was the method used in the input Number method as well above. And this will come up in all other input methods.

Next let's look at inputting Doubles.

```java
/**
 * Prompts the user for a number input and validates it as a double.
 *
 * @param statement the prompt message to display to the user.
 * @return the validated double input from the user.
 */
@Override
public double inputDouble(String statement) {
    String input = inputString(statement);
    boolean valid = validateDouble(input);
    while (!valid) {
        input = inputString("Invalid input.\n"+statement+": ");
        valid = validateDouble(input);
    }
    return Double.parseDouble(input);
}

/**
 * Validates whether the given string can be parsed as a double.
 *
 * @param input the string to validate.
 * @return true if the input is a valid double, false otherwise.
 */
@SuppressWarnings("UnnecessaryTemporaryOnConversionFromString")
@Override
public boolean validateDouble(String input) {
    if (input == null || input.trim().isEmpty()) {
        return false;
    }
    try {
        Double.parseDouble(input.trim());
        return true;
    } catch (NumberFormatException e) {
        return false;
    }
}
```

This is remarkably like the integer checks I did earlier, but I used to try and exceptions. I had to do this because a double as an input can have a character inside of it. So, 1.0 is a valid double but it will be an invalid integer because of the '.' inside of the number. However, the logic is remarkably similar, so I have put both the input and validation in the same screen shot above.

The only other input we have not covered yet is character input. This is needed for when the user inputs the character of the horse in the program. This is an extremely easy method to implement, and it is shown below.

```java
/**
 * Prompts the user for a character input and validates it.
 *
 * @param statement the prompt message to display to the user.
 * @return the validated character input from the user.
 */
@Override
public char inputCharacter(String statement) {
    String input = inputString(statement);
    boolean valid = validateCharacter(input);
    while (!valid) {
        input = inputString(statement);
        valid = validateCharacter(input);
    }
    return input.charAt(index:0);
}

/**
 * Validates if the input string is a valid character.
 *
 * @param input the string to validate.
 * @return true if the input is a valid character, false otherwise.
 */
@Override
public boolean validateCharacter(String input) {
    return (input.length() != 1);
}
```

We once again have the same method lay outs before but this time we are asking for a character, and we are checking for a character. We check for a character by asking for a string input and checking it to make sure it only has a length of 1 showing that it is a character. Then even if this validation method didn't catch the error, then we just return the index 0 of the String to guarantee that the input will be a single character.

Sometimes in the program I will also need to ask the user yes/no questions. Such as: Would you like to continue simulating races? In cases like this where the answer is yes or no or true or false. I will implement a special method to handle these user options.

This is shown below.

```java
/**
 * Prompts the user for a yes/no input and validates it.
 *
 * @param statement the prompt message to display to the user.
 * @return true if the user input is "1" (yes), false if "0" (no).
 */
@Override
public boolean askYesNo(String statement) {
    String input = inputString(statement);
    boolean valid = yesNo(input);
    while (!valid) {
        input = inputString(statement);
        valid = yesNo(input);
    }
    return (input.equals(anObject:"1"));
}

/**
 * Validates if the input is a valid yes/no response.
 *
 * @param input the string to validate.
 * @return true if the input is "1" or "0", false otherwise.
 */
@Override
public boolean yesNo(String input) {
    return (input.equals(anObject:"1") || input.equals(anObject:"0"));
}
```

Here I am asking a yes or no question and asking it until I receive a valid answer with 1 or no 0. Here yes will be equal to 1 because this will return true else if will be false if the input is 0.

The validation of 1s and 0s is done by the yes, no method below the ask Yes, No method.

To allow the user to change the length of the race I have implemented this method which ask for a number from 25 to 100. 25 units is the shortest possible race allowed, and 100 units is the longest race allowed by the program. This uses the input number method previously shown and it is confirmed like all other input in the program. Then the valid race length will be returned.

```
/**
 * Prompts the user to choose a track length and validates it.
 *
 * @param statement the prompt message to display to the user.
 * @return the validated track length input from the user.
 */
@Override
public int chooseTrackLength(String statement) {
    int input = inputNumber(statement);
    while (input < 25 || input > 100) {
        input = inputNumber(statement);
    }
    return input;
}
```

The user will also be able to remove horses from the simulation between races, so I need to implement a method which allows the user to pick out a horse.

```
/**
 * Prompts the user to choose a horse
 *
 * @param statement the prompt message to display to the user.
 * @return the validated horse number
 */
@Override
public int pickAnyHorse(String statement) {
    int input = inputNumber(statement);
    while (input < 1  || input > 8) {
        System.out.println(x:"Invalid index for lane.");
        input = inputNumber(statement);
    }
    return input;
}
```

This makes them pick a lane, since the number of lanes cannot be greater than 8 or less than 1 the input is checked to keep all data in the program correct.

```java
/**
 * Prompts the user to pick a lane and validates it.
 *
 * @param statement the prompt message to display to the user.
 * @param max the maximum number of lanes currently used.
 * @return the validated lane number input from the user.
 */
@Override
public int pickOneOfTheLanes(String statement, int max) {
    int input = inputNumber(statement);
    while (input < 1  || input > max) {
        System.out.println(x:"Invalid index for lane.");
        input = inputNumber(statement);
    }
    return input;
}
```

The same way the user will be able to add and remove horses they will soon be able to remove and add lanes. For this they need to be able to pick a lane, and the method allows exactly this. Since not all 8 of the possible lanes may be used the max variable stores the current max number of lanes in the race.

```java
/**
 * Prompts the user to choose the number of lanes and validates it.
 *
 * @param statement the prompt message to display to the user.
 * @return the validated number of lanes input from the user.
 */
@Override
public int chooseNumberOfLanes(String statement) {
    System.out.println(x:"\n\nNumber of lanes must be greater than 2 to start race simulation.");
    int input = inputNumber(statement);
    while (input < 2 || input > 8) {
        System.out.println(x:"Number of lanes must be greater than 2 to start race simulation.");
        input = inputNumber(statement);
    }
    return input;
}
```

And this above is the method used at the very start of the program execution which allows the user to choose the number of lanes which they want to use in the race. To start the race there must be at least 2 lanes so that there can be at least 2 horses to race.

Now that the user can choose the number of lanes they want, they should be able to choose the number of horses they want, but these 2 variables depend on each other. This is because the number of horses cannot be greater than the number of lanes. So, this next method ensures a valid number of horses is chosen depending on the number of lanes which was chosen.

```java
/**
 * Prompts the user to choose the number of horses given a specific number of lanes and validates it.
 *
 * @param statement the prompt message to display to the user.
 * @param numberOfLanes the number of lanes available.
 * @return the validated number of horses input from the user.
 */
@Override
public int chooseNumberOfHorsesGivenNumberOfLanes(String statement, int numberOfLanes) {
    int input = inputNumber(statement);
    while (input < 2 || input > numberOfLanes) {
        System.out.println(x:"Number of horses must be greater than 2 and less than or equal to the number of lanes.");
        input = inputNumber(statement);
    }
    return input;
}
```

So here is an example. If the number of lanes chosen is 3 then we can have either 3 horses and use all lanes or we can have 2 horses which is the smallest amount to start the race. Number of horses must be less than or equal to the number of lanes. This method ensures that.

This is the end of the user input and user options. Now this needs to be implemented in the race class to allow the user to edit the race simulation between races.

```java
// Between Races or Quit program
finishedSimulation = askYesNo(statement:"\nSTOP SIMULATION:  yes [1], no [0]: ");
if (!finishedSimulation) {
    finishedRace = false;
    if (askYesNo(statement:"\nWould you like to make changes to the next simulation yes [1], no [0]: ")) {
        changeRaceDetails();
    }
```

The code shown above is the section of code which executes after the race has finished. The user is asked if they want to stop the simulation if they answer yes then the program will end. Otherwise, they will be asked if they want to make changes to the simulation. And this is what I have been working towards. So, the change Race Details method will be shown below.

Customising the simulation between races.

```
/**
 * Allows the user to change race details such as removing or adding lanes, removing or adding horses, and changing the race length.
*/
private void changeRaceDetails() {
    if (askYesNo(statement:"\n\nWould you like to Remove any lanes yes [1], no [0]: ")) {
        removeLanes();
        addHorsesToLanes();
    }
    if (askYesNo(statement:"\n\nWould you like to Add any lanes yes [1], no [0]: ")) {
        addLanes();
        addHorsesToLanes();
    }
    if (askYesNo(statement:"\n\nWould you like to Remove any horses yes [1], no [0]: ")) {
        removeHorses();
        addHorsesToLanes();
    }
    if (askYesNo(statement:"\n\nWould you like to Add any horses yes [1], no [0]: ")) {
        addHorses();
        addHorsesToLanes();
    }
    if (askYesNo(statement:"\n\nWould you like to change the length of the race yes [1], no [0]: ")) {
        raceLength = chooseTrackLength(statement:"Length of Race [25m - 100m]: ");
    }
}
```

Currently the options are to remove lanes and to add lanes and to remove horses and to add horses and to change the length of the race.

Let's break this method down into pieces and look at them separately.

Removing lanes.

Firstly, let's look at removing lanes from the race. Removing a lane is quite simple because it doesn't matter if the lane has a horse on it or not.

```
/**
 * Removes lanes from the race.
 */
private void removeLanes() {
    boolean done = false;
    while (horses.size() > 2 && !done) {
        System.out.println("\nThere are currently "+horses.size()+" lanes.");
        System.out.println(x:"The number of lanes cannot be less than 2");
        int input = pickOneOfTheLanes(statement:"Enter the lane number you want to remove: ", horses.size());
        if (input-1 > horses.size()) {
            System.out.println(x:"Invalid Choice of Lane");
        } else {
            if (horses.get(input-1) != null) {
                Horse.horseCounter--;
            }
            horses.remove(input-1);
            if (horses.size() == 2) {
                System.out.println(x:"\nThe number of lanes is now 2, cannot remove more.");
                return;
            }
            done = askYesNo(statement:"Stop removing Lanes yes [1], no [0]");
        }
    }
}
```

For this method we need to set a Boolean flag to check if the user is finished removing the

lanes which they want to remove. They can remove lanes, but they cannot remove lanes once they are down to the last two lanes. The lane which they choose also needs to be confirmed they cannot remove a lane which does not exist. If the lane which they remove happens to have a horse on it then we need to decrease the horse counter and remove the horse from the Array List. This uses one of the methods which I have shown earlier from the user options interface so that the user selects a valid lane which is allowed to be removed. This method also does not return anything as this is just a side effect. Then we ask one more time at the end if they want to remove more lanes or not. If they don't want to remove more lanes, then the 'done' flag is set to true, and we return.

Removing lanes also effects the lane number of the remaining horses and this is also the same if we add lanes, so I have also created a method which adds the horses back to their correct lanes after catching this error during testing.

Since removing lanes can affect the number of horses, we need to check if the lane has a horse on it, if it did then we need to decrease the number of horses by 1.

Adding horses back into correct lanes.

So, the method which puts the horses back into their correct lane is shown below.

```
/**
 * Adds horses to their respective lanes based on their position in the list.
 */
private void addHorsesToLanes() {
    for (int i = 0; i < horses.size(); i++) {
        if (horses.get(i) == null) continue;
        horses.get(i).setLaneNumber(i+1);
    }
}
```

The method above just goes through all the horses which are still in the simulation and add them to their correct lane based on their position in the horses Array List.

This is an amazingly simple method and is needed after every change made to the lanes so that the correct lane numbers are shown to the user.

Adding lanes.

Next, I will develop adding lanes to the race. The code will be below.

```java
/**
 * Adds lanes to the race.
 */
private void addLanes() {
    horses.add(e:null);
    boolean done = false;
    while (horses.size() <= 8 && !done) {
        System.out.println("\nThere are currently "+horses.size()+" lanes.");
        done = askYesNo(statement:"Would you like to add a lane yes [1], no [0]");
        if (done) {
            horses.add(e:null);
            done = false;
        } else {
            done = true;
        }
    }
}
```

I have decided to add new lanes always to the back of the Array List. This is mainly for simplicity, and it avoids which I may face later.

While the number of lanes is less than 8, we can add lanes, but once we have 8 lanes, we have reached the largest amount, and we cannot continue. Otherwise, lane adding stops whenever the user decides to stop adding lanes. After this method is done the earlier horses should be unaffected because the lanes have been added to the back.

This basically means that if there are 2 lanes and the user adds 2 more lanes then their will now be 4 lanes and the lanes 1 and 2 will not be affected.

After all this the user is now able to edit the number of lanes between simulations. But this is also useless as of now if they cannot also add and remove horses to the new lanes.

And this is what will be implemented next.

Removing horses.

I will now develop removing horses and the code will be below.

```java
/**
 * Removes horses from the race.
 */
private void removeHorses() {
    boolean done = false;
    while (horseLanes() >= 2  && !done) {
        showFullLanes();
        int input = pickOneOfTheLanes(statement:"\nEnter the lane number of the horse you want to remove: ", horses.size());
        if (input-1 > horses.size() || horses.get(input-1) == null) {
            System.out.println(x:"Invalid Choice of Lane.");
        } else {
            horses.set(input-1, element:null);
            Horse.horseCounter--;
            done = askYesNo(statement:"\nStop removing horses yes [1], no [0]: ");
        }
    }
}
```

To remove horses from the race we first need to tell the user which horse is in which lane. This is done by the method called on the second line of the remove Horse method.

I will go over that method after explaining this one.

To remove a horse, we need the user to pick the lane which holds the horse which the user wants to remove. This requires input validation just like all other input. It must be valid. Once we have a valid input then we set the horse at that lane null and this will make that lane an empty lane effectively removing the horse, which was the aim of this method.

The user can remove as many horses as they want but the number of horses cannot go below 2. Also, since we are now removing horses the horse count variable needs to be changed every time a horse is removed here as well.

Show full lanes.

```
/**
 * Displays lanes that are occupied by horses.
 */
private void showFullLanes() {
    System.out.println(x:"\nLanes with Horses: ");
    for (int i = 0; i < horses.size(); i++) {
        if (horses.get(i) == null) continue;
        int fullLane = i+1;
        System.out.println("This lane has a horse: "+fullLane);
    }
}
```

The method above shows the lane number of the full lanes. These are the lanes which have horses so the user can remove them if they choose to do so.

This method to remove horses also needs to be able to check how many horses there are in the race so I have written a method which is in the while loop condition check to count if any horses could be removed or if the user should be able to remove any more.

The method is shown below.

```
/**
 * Counts the number of lanes with horses.
 *
 * @return the number of lanes with horses.
 */
private int horseLanes() {
    int count = 0;
    for (Horse horse : horses) {
        if (horse != null) count++;
    }
    return count;
}
```

This method returns the exact count of horses in the race.

If this returns a number less than or equal to 2 then the user cannot remove more horses.

Add horses.

Now that I can remove horses, I need to be able to add horses this will be shown below.

```
/**
 * Adds horses to the race.
 */
private void addHorses() {
    boolean done = false;
    while (!fullLanes() && !done) {
        showEmptyLanes();
        int input = pickOneOfTheLanes(statement:"\nEnter the lane number you want to add a horse to: ", horses.size());
        if (input-1 > horses.size() || horses.get(input-1) != null) {
            System.out.println(x:"Invalid Lane or Taken Lane.");
        } else {
            horses.set(input-1, createHorse(input));
            Horse.horseCounter++;
            done = askYesNo(statement:"Stop adding horses yes [1], no [0]: ");
        }
    }
}
```

To be able to add horses to the simulation the user needs to be able to know which lanes are empty, they need to know which lanes are free so they can add horses to.

So, I have also written a method to show empty lanes. This method is shown below.

Show empty lanes.

```
/**
 * Displays lanes that are empty.
 */
private void showEmptyLanes() {
    System.out.println(x:"\nEmpty Lanes: ");
    for (int i = 0; i < horses.size(); i++) {
        if (horses.get(i) != null) continue;
        int emptyLane = i+1;
        System.out.println("Empty Lane: "+emptyLane);
    }
}
```

If the horse at a certain index is null in the Array List, then we know that it is an empty lane so we can show this to the user so they know which lanes they can choose to add horses to.

Then we can confirm their input and it if it's valid then to add a horse to the race we can reuse the create horse method to allow the user to type in the exact horse they want. Then we increase the number of horses by 1 and an ask if they want to add more if there are more empty lanes. We also need a way to check if all lanes are full. So, I have written a method for this as well. This is used in the while loop condition in the method to add horses. This method is shown below.

Check if all lanes are full.

```
/**
 * Checks if all lanes are occupied by horses.
 *
 * @return true if all lanes are occupied, false otherwise.
 */
private boolean fullLanes() {
    for (Horse horse : horses) {
        if (horse == null) return false;
    }
    System.out.println(x:"\nAll lanes taken");
    return true;
}
```

This method return true of all the lanes are full and it return false if it finds an empty lane. This method plays a vital role in adding horses between races.

```
/**
 * Allows the user to change race details such as removing or adding lanes, removing or adding horses, and changing the race length.
 */
private void changeRaceDetails() {
    if (askYesNo(statement:"\n\nWould you like to Remove any lanes yes [1], no [0]: ")) {
        removeLanes();
        addHorsesToLanes();
    }
    if (askYesNo(statement:"\n\nWould you like to Add any lanes yes [1], no [0]: ")) {
        addLanes();
        addHorsesToLanes();
    }
    if (askYesNo(statement:"\n\nWould you like to Remove any horses yes [1], no [0]: ")) {
        removeHorses();
        addHorsesToLanes();
    }
    if (askYesNo(statement:"\n\nWould you like to Add any horses yes [1], no [0]: ")) {
        addHorses();
        addHorsesToLanes();
    }
    if (askYesNo(statement:"\n\nWould you like to change the length of the race yes [1], no [0]: ")) {
        raceLength = chooseTrackLength(statement:"Length of Race [25m - 100m]: ");
    }
}
```

Now I have explained all methods in this method which remove / add lanes and remove / add horses. The last if statement is there to ask if the user wants to change the length of the race, but that one has already been covered earlier in this document because that method is being reused. To change the length of the race we use the same method we set it with at the start of the program.

Horse success variables.

The next feature I decided to add was a race counter, win counter and win-loss ratio for the horses. When a new horse is created then we will set the number of races and number of

wins and the win ratio all to 0. This will slightly change the constructor method for the horses and add 3 more private attributes which all require getter and setter methods. These changes are all simple. I am deciding to add these changes because I am planning to save horse details such as name, symbol, confidence, win-loss ratio, win count and race count within a file system. And this is what I will be working on next.

To add a race count, win count and win rate variable I need to change the constructor. So, now the constructor for the Horse class looks like this.

Win-rate, total races, total races.

```
/**
 * Constructor for objects of class Horse.
 * Initializes the horse with a symbol, name, confidence rating, and lane number.
 * The distance traveled is set to 0, and the horse starts as not fallen.
 * Confidence is clamped between 0.0 and 1.0.
 *
 * @param horseName String representation of the horse's unique identifier
 * @param horseConfidence double representation of confidence value (0.0-1.0)
 * @param horseSymbol character containing single character visual representation
 * @param totalWins integer representation of lifetime victory count
 * @param totalRaces integer representation of lifetime race participation
 * @param winRate double representation of win percentage (0.0-1.0)
 * @param lane integer representing the lane number of the horse
 */
public Horse(String horseName, double horseConfidence, char horseSymbol, int totalWins, int totalRaces, double winRate, int laneNumber) {
    this.horseName = horseName;
    // Ensure confidence is within the valid range [0.0, 1.0]
    if (horseConfidence > 1.0) {
        this.horseConfidence = 1.0;
    } else if (horseConfidence < 0.0) {
        this.horseConfidence = 0.0;
    } else {
        this.horseConfidence = horseConfidence;
    }
    this.horseSymbol = horseSymbol;
    this.totalWins = totalWins;
    this.totalRaces = totalRaces;
    if (this.totalRaces == 0) {
        this.winRate = 0;
    } else {
        this.winRate = totalWins / totalRaces;
    }
    this.laneNumber = laneNumber;

    this.horseDistance = 0;
    this.horseFallen = false;
    horseCounter++;
}
```

The win rate variable needs to be checked by an if statement because we cannot divide by 0 of the total number of races is 0.

With these new variables we need to edit these whenever a horse has run in a race and change it if a horse has won the race. And it doesn't matter if the horse has won or lost the new win rate needs to be calculated. That will be done at once after I have finished creating the getter and setter methods for these new private attributes.

Getters and setters.

```java
/**
 * Retrieves the cumulative count of races won by this horse.
 *
 * @return Total wins as a non-negative integer
 */
public int getTotalWins() {
    return this.totalWins;
}

/**
 * Updates the horse's lifetime win count.
 *
 * @param totalWins New win count (must be non-negative)
 * @throws IllegalArgumentException if negative value is provided
 */
public void setTotalWins(int totalWins) {
    if (totalWins < 0) {
        throw new IllegalArgumentException(s:"Win count cannot be negative");
    }
    this.totalWins = totalWins;
}
```

The getter and setter methods for the total number of wins are shown above and the total number of wins cannot be set to a negative number. And the get method just returns the value of the total wins attribute.

The same is done for the total number of races attribute. These are shown below. However, the total races can never be less than the number of wins and it also cannot be negative so there is an extra restriction. These restrictions must be put in place to support the safety of the data.

```java
/**
 * Retrieves the cumulative count of races participated in by this horse.
 *
 * @return Total races as a non-negative integer
 */
public int getTotalRaces() {
    return this.totalRaces;
}

/**
 * Updates the horse's lifetime race participation count.
 *
 * @param totalRaces New race count (must be non-negative and ≥ totalWins)
 * @throws IllegalArgumentException if invalid count is provided
 */
public void setTotalRaces(int totalRaces) {
    if (totalRaces < 0) {
        throw new IllegalArgumentException(s:"Race count cannot be negative");
    }
    if (totalRaces < this.totalWins) {
        throw new IllegalArgumentException(s:"Race count cannot be less than win count");
    }
    this.totalRaces = totalRaces;
}
```

Then I needed to do the same to the win rate variable. This is also shown below.

Here in the setter, we need to check that both the total win and the total races are both nonnegative numbers and we also need to make sure that the number if wins is less than or equal to the number of races otherwise, we throw a new illegal argument exception. If we do not throw, then we calculate and set the new win rate. It is set to 0 of the horse has not had any races yet.

```java
/**
 * Retrieves the current win rate percentage.
 *
 * @return Win rate as a decimal between 0.0 (0%) and 1.0 (100%)
 */
public double getWinRate() {
    return this.winRate;
}

/**
 * Calculates and updates the horse's win rate based on current statistics.
 * Win rate is computed as wins divided by total races, with protection against
 * division by zero (returns 0.0 when no races completed).
 *
 * @param wins Number of wins (must be <= total races and non-negative)
 * @param total Total races (must be >= wins and non-negative)
 */
public void setWinRate(double wins, double total) {
    if (wins < 0 || total < 0) {
        throw new IllegalArgumentException(s:"Values cannot be negative");
    }
    if (wins > total) {
        throw new IllegalArgumentException(s:"Wins cannot exceed total races");
    }
    this.winRate = (total == 0) ? 0.0 : wins / total;
}
```

Now that these attributes can be used and accessed, I need to make sure that I change them when a horse runs in a race and when it wins a race.

Update horse success variables after the race.

```
/**
 * Displays the winner of the race and adjusts the confidence of the horses.
 * And adjust the total races, total wins, and win rate of the horse
 */
private void showWinner() throws IOException {
    for (Horse horse : horses) {
        if (horse == null) continue;
        horse.setTotalRaces(horse.getTotalRaces()+1);
        if (raceWonBy(horse)) {
            System.out.println("\n\n"+horse.getName()+" has won the Race!");
            horse.setConfidence(horse.getConfidence()*1.1);
            horse.setTotalWins(horse.getTotalWins()+1);
        } else {
            horse.setConfidence(horse.getConfidence()*0.9);
        }
        horse.setWinRate(horse.getTotalWins(), horse.getTotalRaces());
    }
}
```

This method has been shown before but it is reused to change the total race count and win rate and win count of the horses.

We increase the total race count of all horses; the win count of the winning horse is incremented by one and a new win rate is calculated for all horses. That is the end of this feature.

To put this to use I will now start file handling where I will be able to store all the details about every horse which the user decides to save. The data I would like to be able to save includes the horse's name, horse character, horse confidence, race count, win count, and win ratio. This will allow me to be able to develop a feature which lets me allow the user to be able to reuse saved horses and add horses from files into the simulation.

To ensure that there are no repeated horses in the file I want the names of the horses to be unique. The simplest way to implement unique names is to have an Array List of type String which will hold the names of all the horses and this will be updated at the same time as the horses Array List and this will allow for unique names and it will mean that horses with the same name cannot over write other horses.

Unique horse names.

```
static ArrayList<Horse> horses = new ArrayList<>();
static ArrayList<String> uniqueHorseNames = new ArrayList<>();
```

So now I have 2 Array Lists. I won't show all the instances where these arrays are changed but this now ensures that unique names are enforced.

```
/**
 * Names must be unique
 * check if name is taken
 *
 */
private boolean usedName(String name) {
    for (int i = 0; i < uniqueHorseNames.size(); i++) {
        if (name.equals(uniqueHorseNames.get(i))) {
            System.out.println(x:"\nHorse name taken. Choose another name.");
            return true;
        }
    }
    return false;
}
```

When a new name is entered it is always checked against the names we already have. If there are any names which are the same the user will be told to enter a new name. If the name is already taken by a horse, then it will be flagged and true will be returned by the method. Otherwise, it will be false, and the name entered is fine.

File Handling. (Saving Horse Data)

This is now the beginning of the file handling section.

This is the start of the file handling, this is a comma separated value system, and the expected number of columns is 6 as said earlier.

Name, Confidence, Character, Total Wins, Total Races, Win Rate

```
public class HorseDetailsFileHandling {
    private static final String FILE_NAME = "SavingHorseDetails.csv";
    private static final int EXPECTED_COLUMNS = 6;
```

Now to add to add horse to the file we need all the details of the horse to be passed to the write method and write this into the file.

Saving horse details.

```java
/**
 * Saves horse details to the CSV file in append mode after validating parameters
 * and ensuring the horse name doesn't already exist in the file. If the horse
 * name exists, the method returns without making any changes.
 *
 * @param name The name of the horse (cannot be null or empty)
 * @param confidence The confidence score (0.0-1.0 scale)
 * @param character The character symbol representing the horse
 * @param win Number of wins (must be >= 0)
 * @param total Total races (must be >= wins)
 * @param winRate Win percentage (0.0-1.0 scale)
 * @throws IOException If an I/O error occurs
 * @throws IllegalArgumentException If any parameter is invalid
 */
public static void saveHorseDetails(String name, double confidence, char character, int win, int total, double winRate) throws IOException {
    // Parameter validation
    if (name == null || name.trim().isEmpty()) {
        throw new IllegalArgumentException(s:"Horse name cannot be null or empty");
    }
    if (confidence < 0.0 || confidence > 1.0) {
        throw new IllegalArgumentException(s:"Confidence must be between 0.0 and 1.0");
    }
    if (win < 0 || total < win) {
        throw new IllegalArgumentException(s:"Invalid win/total race counts");
    }
    if (winRate < 0.0 || winRate > 1.0) {
        throw new IllegalArgumentException(s:"Win rate must be between 0.0 and 1.0");
    }

    // Check if horse name already exists
    File file = new File(FILE_NAME);
    if (file.exists()) {
        try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(regex:",");
                if (parts.length > 0 && parts[0].equalsIgnoreCase(name)) {
                    return; // Horse already exists - silent return
                }
            }
        }
    }

    // Save the new horse
    try (PrintWriter horseFile = new PrintWriter(new FileWriter(file, append:true))) {
        horseFile.printf(format:"%s,%.2f,%c,%d,%d,%.2f%n",name, confidence, character, win, total, winRate);
    }
}
```

This is quite a large method, but I will explain. We give this method all the important attributes of the horse and make sure they are all valid otherwise we throw an Illegal Argument Exception. Then if the file already exists then we check if the name of the horse is already in the file or not. Since the name are unique if the name is already in the file, then we can just return, the horse does not need to be saved twice. Otherwise, we format a String and write everything to the file all at once. This method des not return a value as it is a side effect.

- **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).`

Throughout the development of the program, I do not think the Constructor will change any more. All the attributes which are required have now been added and they have all been correctly defined and implemented as needed, and all of them are in the constructor.

CSV file.

The file now looks something like this.

```
Name,Confidence,Character,Total Wins,Total Races,Win Rate
Horse 1,0.23,1,0,1,0.00
Horse 2,0.23,2,0,1,0.00
Horse 3,0.23,3,0,1,0.00
Horse 4,0.23,4,0,1,0.00
Horse 5,0.23,5,0,1,0.00
Horse 6,0.28,6,1,1,1.00
Horse 7,0.23,7,0,1,0.00
Horse 8,0.23,8,0,1,0.00
```

Making the CSV into a table for user.

This shows the insides of the comma separated value file after 8 horses ran a race. Here we can deduce that the horse called ‘Horse 6’ found in row 6 of the file has won the race as it has a win ratio of 1 and a higher confidence, while the other horses have a lower confidence and a win rate of 0.

This file however must be visible to the user, so I need to develop a metho which formats the table in a user-friendly way and displays all data. This I will be able to do by reading the table into a list and then print it out in a nice table format.

The code which I will develop will be shown below.

```java
/**
 * Reads all horse details from the CSV file.
 *
 * @return List of String arrays where each array represents a horse record
 * @throws IOException If an I/O error occurs while reading
 */
public static List<String[]> readCSV() throws IOException {
    List<String[]> data = new ArrayList<>();
    File file = new File(FILE_NAME);

    if (!file.exists() || file.length() == 0) {
        return data;
    }

    try (BufferedReader br = new BufferedReader(new FileReader(file))) {
        String line;
        while ((line = br.readLine()) != null) {
            String[] row = line.split(regex:",", -1); // Keep empty values
            data.add(validateDataRow(row));
        }
    }
    return data;
}
```

This is the code to read the CSV

The confirm data row method is used to check for invalid rows within the data which do not meet the table requirements of the 6 columns. This is the method which does this check, and it is shown below.

```java
/**
 * Validates a data row to ensure it has the correct number of columns.
 *
 * @param row The data row to validate
 * @return The validated row array
 * @throws IllegalArgumentException If row has incorrect number of columns
 */
private static String[] validateDataRow(String[] row) {
    if (row.length != EXPECTED_COLUMNS) {
        throw new IllegalArgumentException(
            String.format(format:"Invalid data format. Expected %d columns, found %d",
            EXPECTED_COLUMNS, row.length));
    }
    return row;
}
```

Now that only valid rows are returned, we can enforce a table format.

```java
/**
 * Displays all horse details in a formatted table with borders.
 * provides clear error messages if error happens
 */
public static void printFormattedTable() {
    try {
        List<String[]> data = readCSV();

        if (data.isEmpty()) {
            System.out.println(x:"No horse data available.");
            return;
        }

        // Verify all rows have expected columns
        for (String[] row : data) {
            if (row.length != EXPECTED_COLUMNS) {
                System.err.println("Error: Malformed data found. Expected " +
                                   EXPECTED_COLUMNS + " columns but found " + row.length);
                return;
            }
        }

        // Calculate column widths
        int[] colWidths = new int[EXPECTED_COLUMNS];
        for (String[] row : data) {
            for (int i = 0; i < EXPECTED_COLUMNS; i++) {
                colWidths[i] = Math.max(colWidths[i], row[i].length());
            }
        }

        // Print table
        printSeparatorLine(colWidths);
        printTableRow(data.get(index:0), colWidths); // Header row
        printSeparatorLine(colWidths);

        for (int i = 1; i < data.size(); i++) {
            printTableRow(data.get(i), colWidths);
            printSeparatorLine(colWidths);
        }
    } catch (IOException e) {
        System.err.println("Error reading horse data: " + e.getMessage());
    }
}
```

This method checks for an empty file, then checks for invalid rows, then calculates the length of the columns needed. Then it will use helper methods which will later be shown to print out the table rows followed by the separator lines. This entire process must be wrapped inside of a try and catch statement however just in case we meet any file reading errors.

The helper methods will be shown below.

To print out the separator lines I have written this method.

```java
/**
 * Prints a separator line for the table.
 *
 * @param colWidths Array of column widths
 */
private static void printSeparatorLine(int[] colWidths) {
    System.out.print(s:"+");
    for (int width : colWidths) {
        System.out.print("-".repeat(width + 2) + "+");
    }
    System.out.println();
}
```

To print these lines in the file table the process is like printing the track in the horse racing. For a certain amount of width, we must print a certain number of characters which will be the separator lines in the table.

Then to print the actual lines with data in them this is the helper method.

```java
/**
 * Prints a formatted table row with proper alignment.
 *
 * @param row The data row to print
 * @param colWidths Array of column widths
 */
private static void printTableRow(String[] row, int[] colWidths) {
    System.out.print(s:"|");
    for (int i = 0; i < row.length; i++) {
        String value = row[i].trim();
        if (value.matches(regex:"[A-Za-z0-9]+")) { // Alphanumeric value
            System.out.printf(" %" + colWidths[i] + "s |", value);
        } else { // String value
            System.out.printf(" %-" + colWidths[i] + "s |", value);
        }
    }
    System.out.println();
}
```

For all the data in the line we just print it out, here I chose to use a regex for the horse characters since they can be any value from A to Z or a-z or 0-9.

Putting all these methods together for displaying the table allows me to be able to make this table inside of the terminal for the user to see.

Horse Details Table.

```
Saved Horses:
+---------+------------+-----------+------------+-------------+----------+
|    Name | Confidence | Character | Total Wins | Total Races | Win Rate |
+---------+------------+-----------+------------+-------------+----------+
| Horse 1 | 0.23       |         1 |          0 |           1 | 0.00     |
+---------+------------+-----------+------------+-------------+----------+
| Horse 2 | 0.23       |         2 |          0 |           1 | 0.00     |
+---------+------------+-----------+------------+-------------+----------+
| Horse 3 | 0.23       |         3 |          0 |           1 | 0.00     |
+---------+------------+-----------+------------+-------------+----------+
| Horse 4 | 0.23       |         4 |          0 |           1 | 0.00     |
+---------+------------+-----------+------------+-------------+----------+
| Horse 5 | 0.23       |         5 |          0 |           1 | 0.00     |
+---------+------------+-----------+------------+-------------+----------+
| Horse 6 | 0.28       |         6 |          1 |           1 | 1.00     |
+---------+------------+-----------+------------+-------------+----------+
| Horse 7 | 0.23       |         7 |          0 |           1 | 0.00     |
+---------+------------+-----------+------------+-------------+----------+
| Horse 8 | 0.23       |         8 |          0 |           1 | 0.00     |
+---------+------------+-----------+------------+-------------+----------+
```

And this is what the table looked like in the file:

```
Name,Confidence,Character,Total Wins,Total Races,Win Rate
Horse 1,0.23,1,0,1,0.00
Horse 2,0.23,2,0,1,0.00
Horse 3,0.23,3,0,1,0.00
Horse 4,0.23,4,0,1,0.00
Horse 5,0.23,5,0,1,0.00
Horse 6,0.28,6,1,1,1.00
Horse 7,0.23,7,0,1,0.00
Horse 8,0.23,8,0,1,0.00
```

At the end of every race the user will now be asked if they want to save a horse into the file system.

To be able to save these horses the code has not been showed yet, but here it is.

Ask User if they want to save a horse.

```
/*
 * Save a horse details
 * User choice which one to save
 */
private void saveHorse() throws IOException {
    showHorseDetails();
    int pickedHorseIndex = pickAnyHorse(statement:"\nEnter the lane number of the horse you want to save: ") -1;
    while (horses.get(pickedHorseIndex) == null) {
        System.out.println(x:"\nLane is empty. ");
        pickedHorseIndex = pickAnyHorse(statement:"\nEnter the lane number of the horse you want to save: ") -1;
    }
    Horse horse = horses.get(pickedHorseIndex);
    HorseDetailsFileHandling.saveHorseDetails(horse.getName(), horse.getConfidence(), horse.getSymbol(), horse.getTotalWins(), horse.getTotalRaces(), horse.getWinRate());
}
```

This method has a helper method which shows the details of all the horses inside of the race. This will be shown later. Then the user is asked to enter the lane number of the horse

they want to save, and all the details of the horse will be passed to the write method which was shown at the start of the file handling section.

Now to call the method which prints the details of each horse is shown below.

Showing the horses in the race they can choose to save.

```
/**
 * Shows the Details of each horse in the race
 */
private void showHorseDetails() {
    for (Horse horse : horses) {
        if (horse == null) continue;
        System.out.println("Lane: "+horse.getLaneNumber()+", "+horse.getName()+" with confidence "+horse.getConfidence());
    }
}
```

Now that the user can save horses inside of a file and they can view the horses inside of the file my next step is to be able to let the user use these saved horses inside of a race. To be able to do this they need to be able to read the file once again so I can reuse the same method to print the formatted table and then the user needs to pick the row number of the horse they would like to use in the race.

The code which I will develop will be shown below.

Firstly, to be able to read a horse from the file, the file cannot be empty, so I need to check the file to make sure it has more than 1-line apart from the header. Then I need a method to return a String array full of the data. I'm going to start with the line counter.

```
/**
 * Counts the number of records in the horse details file.
 *
 * @return Number of records (0 if file doesn't exist or is empty, or no horses saved)
 * @throws IOException If an I/O error occurs while reading
 */
public static int countFileLines() throws IOException {
    File file = new File(FILE_NAME);
    if (!file.exists() || file.length() == 0) {
        return 0;
    }

    int lineNumber = -1; // Start at -1 to ignore initiall line.
    try (BufferedReader horseFile = new BufferedReader(new FileReader(file))) {
        while (horseFile.readLine() != null) {
            lineNumber++;
        }
    }
    return lineNumber;
}
```

This method uses a simple while loop to count every line if there is a line to read. The line counter starts from –1 so that counting the header in the file makes it go to 0. and 0 will also be returned if the file does not exist or if the length of the file is 0. Otherwise, we enter the try and except block to count the lines.

And the line number is returned.

Then I need a method to return a specific horse from the file. This will be shown below.

Retrieving Horses from the File.

I want to return specific horses to the user based on which row in the file they want so I can write a method which iterates up to that line and returns that line of the file.

```
/**
 * Retrieves horse details from a specific line in the file.
 *
 * @param lineNumber The 1-based line number to retrieve
 * @return String array containing horse details, or null if line doesn't exist
 * validation happens before method call.
 * (invalid line number will not be passed to this method)
 * @throws IOException If an I/O error occurs
 * @throws IllegalArgumentException If lineNumber is invalid
 */
public static String[] getHorseDetails(int lineNumber) throws IOException {
    if (lineNumber < 1) {
        throw new IllegalArgumentException(s:"Line number must be positive");
    }

    File file = new File(FILE_NAME);
    if (!file.exists()) {
        return null;
    }

    try (BufferedReader horseFile = new BufferedReader(new FileReader(file))) {
        String line;
        int currentLine = 0;
        while ((line = horseFile.readLine()) != null) {
            if (currentLine++ == lineNumber) {
                return validateDataRow(line.split(regex:",", -1));
            }
        }
    }
    return null;
}
```

This will also use a method which I have shown earlier, this confirms the data in the row then returns it.

Now that I can return lines from the file, I can return them back into the race Class and add the horse to the race.

```
/**
 * Ask the user if they want to use a horse from the file
 *
 * @throws IOException I/O Error
 */
private void chooseSavedHorse() throws IOException {
    boolean answer = askYesNo(statement:"\nWould you like to use a previously saved horse yes [1], no [0]: ");
    if (answer) {
        if (HorseDetailsFileHandling.countFileLines() == 0) {
            System.out.println(x:"\nThere are curently no saved horses...");
            return;
        }
        showHorseDetailsFromFile();
        askUserToChooseHorseFromFile();
    }
}
```

To check if the user would like to use a horse from the file, I ask them and if they say yes, true will be returned (This method has been shown before) and then we check if there are any horses in the file. If these are horses in the file, then we output the formatted table to the user to see all the horses which they must choose from.

This is the method which outputs the formatted table previously shown.

```
/**
 * Print the formated table for all the horses
 * which are saved in files
 */
private void showHorseDetailsFromFile() {
    System.out.println(x:"\n\n\nSaved Horses: ");
    HorseDetailsFileHandling.printFormattedTable();
}
```

Then in the next method which we call I ask them to enter the row number of the horse which they want and then create a horse with the returned details.

```java
/**
 * Ask for a line in the file
 * Create a horse with those details
 *
 * @throws IOException I/O Error
 */
private void askUserToChooseHorseFromFile() throws IOException {
    int numberOfSavedHorses = HorseDetailsFileHandling.countFileLines();
    if (numberOfSavedHorses >=1 ) {
        int input = inputNumber(statement:"\nEnter the row number of the horse you want to use: ");
        if (input <= 0 || input > numberOfSavedHorses) {
            System.out.println(x:"\nInvalid choice of file rows.");
            input = inputNumber(statement:"\nEnter the row number of the horse you want to use: ");
        }
        String[] horseDetails = HorseDetailsFileHandling.getHorseDetails(input);
        Horse horse = new Horse(horseDetails[0], horseDetails[1], horseDetails[2], horseDetails[3], horseDetails[4], horseDetails[5], Horse.horseCounter+1);
        horses.add(Horse.horseCounter-1, horse);
        uniqueHorseNames.add(Horse.horseCounter-1, horse.getName());
        System.out.println("\nThis horse will be added to lane "+Horse.horseCounter);
    }
}
```

This will ask for a valid row number from the file and then use that number to return a String array of all the horse details. Since we are reading the data from the file all the data is String, so I have had to create a new constructor for horses which are being created with String data types. And this method will also check the horse has a unique name and add it into the next free lane number.

Now that the file horses can be read the next step is to create the constructor which accepts the String inputs.

Horse Constructor made for String input from file.

```java
/**
 * Constructs a Horse object from file-derived string data.
 * Constructor for objects of class Horse.
 * Initializes the horse with a symbol, name, confidence rating, and lane number.
 * The distance traveled is set to 0, and the horse starts as not fallen.
 * Confidence is clamped between 0.0 and 1.0.
 *
 * @param horseName String representation of the horse's unique identifier
 * @param horseConfidence String representation of confidence value (0.0-1.0)
 * @param horseSymbol String containing single character visual representation
 * @param totalWins String representation of lifetime victory count
 * @param totalRaces String representation of lifetime race participation
 * @param winRate String representation of win percentage (0.0-1.0)
 * @param lane integer representing the lane number of the horse
 */
public Horse(String horseName, String horseConfidence, String horseSymbol, String totalWins, String totalRaces, String winRate, int lane) {
    this.horseName = horseName;
    // Ensure confidence is within the valid range [0.0, 1.0]
    if (Double.parseDouble(horseConfidence) > 1.0) {
        this.horseConfidence = 1.0;
    } else if (Double.parseDouble(horseConfidence) < 0.0) {
        this.horseConfidence = 0.0;
    } else {
        this.horseConfidence = Double.parseDouble(horseConfidence);
    }
    this.horseSymbol = horseSymbol.charAt(index:0);
    this.totalWins = Integer.parseInt(totalWins);
    this.totalRaces = Integer.parseInt(totalRaces);
    if (this.totalRaces == 0) {
        this.winRate = 0;
    } else {
        this.winRate = Double.parseDouble(winRate);
    }
    this.laneNumber = lane;

    this.horseDistance = 0;
    this.horseFallen = false;
    horseCounter++;
}
```

This new constructor is identical to the earlier one, but this is only used for horses which are coming from files. While testing the files I noticed that I need to change the details in the file for all horses which may be saved in the file already. So, if a horse is in the file and it just won a race then the race count and total win count and win ratio need to be updated in the file system. So, I also need to write an update method which is called at the end of every race and checks if any horse in the race is already in the file or not. If it's then the added details need to be written to the file, and the old ones need to be removed.

Updating Horses in the File After Race.

```
/**
 * Update the saved horse details after every race
 * Finds horse based on name (name is unique) and updates all of its details
 * @param horse The horse which will be updated in the file... IF it is in the file already.
 */
public static void updateHorseInFile(Horse horse) throws IOException {
    List<String> lines = new ArrayList<>();
    try (BufferedReader reader = new BufferedReader(new FileReader(FILE_NAME))) {
        String line;
        // Read header first
        String header = reader.readLine();
        if (header != null) {
            lines.add(header);
        }

        while ((line = reader.readLine()) != null) {
            String[] parts = line.split(regex:",");
            if (parts.length >= 6 && parts[0].equalsIgnoreCase(horse.getName())) {
                // Found the horse - replace with updated details
                String updatedLine = String.format(format:"%s,%.2f,%s,%d,%d,%.2f",
                    horse.getName(),
                    horse.getConfidence(),
                    horse.getSymbol(),
                    horse.getTotalWins(),
                    horse.getTotalRaces(),
                    horse.getWinRate());
                lines.add(updatedLine);
            } else {
                lines.add(line);
            }
        }
    }

    // Write all lines back to the file
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE_NAME))) {
        for (String fileLine : lines) {
            writer.write(fileLine);
            writer.newLine();
        }
    } catch (IOException e) {
        System.err.println("Error writing to file: " + e.getMessage());
    }
}
```

For all horses at the end of the race this method is called, then we check if the horse is in the file or not. First, I read the whole file to save it then we find the horse which needs to be updated and add its new line then I rewrite all other lines back into the file with the new updated line. This method is called at the end of every race to keep all details up to date and correct.

This is the end of the file handling section. Now I can save horses into a comma separated value file after the end of every race and then let the user to add horses from the file into the race.

Random Horses.

The next feature I decided to implement into the horse racing simulator is to make random horses. To make a random horse all I need is random names and random symbols to stand for each horse. The lane number which these horses can be added to is just the next available spot in the track.

First, I decided to code the method which combines prefixes and suffixes to form a name and returns it. This process is random to minimize the chance of the same name occurring twice.

```
/**
 * Generates a random horse name by combining prefixes and suffixes
 * @return A randomly generated horse name
 */
private String generateRandomHorseName() {
    String[] prefixes = {
        "Thunder", "Midnight", "Shadow", "Lightning", "Silver",
        "Golden", "Diamond", "Black", "White", "Red",
        "Wild", "Crazy", "Majestic", "Royal", "Brave",
        "Flying", "Dancing", "Galloping", "Mystic", "Spirit"
    };

    String[] suffixes = {
        "hoof", "mane", "tail", "storm", "fire",
        "wind", "blaze", "dancer", "chaser", "runner",
        "prince", "king", "queen", "star", "moon",
        "sun", "dream", "whisper", "shadow", "flash"
    };

    Random random = new Random();
    String prefix = prefixes[random.nextInt(prefixes.length)];
    String suffix = suffixes[random.nextInt(suffixes.length)];

    return prefix + " " + suffix;
}
```

Then I needed to write a method which gives the horse a random character. Horse characters can be any alphanumeric value from A-Z or a-z or 0-9.

This was also an extremely easy method to implement.

```java
/**
 * Generates a random alphanumeric character (A-Z, a-z, 0-9)
 * @return A random character
 */
private char generateRandomAlphanumericChar() {
    Random random = new Random();
    int choice = random.nextInt(bound:3); // 0, 1, 2

    return (char) (switch (choice) {
        case 0 -> random.nextInt(bound:26) + 65;
        case 1 -> random.nextInt(bound:26) + 97;
        case 2 -> random.nextInt(bound:10) + 48;
        default -> 'X';
    });
}
```

Now that I have 2 methods for random names and random characters, I need to ask the user if they even want random horses, and if they do, how many do they want.

```java
/**
 * Ask the user if they want random horses
 * Ask how many they want
 * add as many of those horses as possible considering how many empty lanes there are
 * if all lanes fill up and all horses cannot be addded then tell the user.
 */
private void askUserToGenerateRandomHorse() {
    boolean input = askYesNo(statement:"\nWould you like to add a randomly generated horse into the race? yes [1] no [0]: ");
    if (!input) return;

    int numberOfRandomHorses = inputNumber(statement:"\nHow many Random Horse would you like [0 - 8]: ");
    if (numberOfRandomHorses == 0) return;

    while (numberOfRandomHorses < 0 || numberOfRandomHorses > 8) {
        System.out.println(x:"\nInvalid choice. Try again.");
        numberOfRandomHorses = inputNumber(statement:"\nHow many Random Horse would you like [0 - 8]: ");
    }

    int added = 0;
    while (added < numberOfRandomHorses && horses.size() < 8) {
        Horse horse = generateRandomHorse();
        horses.add(Horse.horseCounter-1, horse);
        uniqueHorseNames.add(Horse.horseCounter-1, horse.getName());
        added++;
    }
    if (horses.size() == 8 && added != numberOfRandomHorses) {
        System.out.println(x:"\nAll random horses could not be added,\n8 lane limit reached.");
        System.out.println(numberOfRandomHorses-added+" random horses has not been addded.");
    }
}
```

The method above asks the user if they want random horse and if they do want random horses they can enter how many they want. Then all horses will be generated using a method which will be shown next, and they will be added to the horses Array List just like

all other horses. If we run out of lanes, then we need to tell the user we were not able to add all their horses and tell them that the lanes are full and how many we were not able to add. Then we return.

To generate the horse's the method is shown below.

```
/**
 * Generates a random horse with a randomly generated name and character
 * @return A new Horse object with random attributes
 */
private Horse generateRandomHorse() {
    String name = generateRandomHorseName();
    while (usedName(name)) {
        name = generateRandomHorseName();
    }
    char character = generateRandomAlphanumericChar();
    Horse randomHorse = new Horse(name, horseConfidence:0.25, character, totalWins:0, totalRaces:0, winRat…0.0, Horse.horseCounter+1);
    System.out.println(x:"\n\nRandom Horse: ");
    System.out.println("Horse Name: "+randomHorse.getName()+",\nHorse Symbol: "+randomHorse.getSymbol());

    return randomHorse;
}
```

First we generate a random name then a random character and create a horse with these details and then we show the horse details to the user to let them know the details about the random horse and then we return the horse to be added to the horses Array List.

Also I have had to go back and make changes to this method during testing and make sure the random names are also unique, there is a low chance of 2 same names being generated so in that case we need to generate a new name hence why I had to wrap that in a while loop after the name generation.

With these methods I am now able to generate random horses, and I will show this through the file table format.

Testing Random Horses.

Further testing on random horses:

```
How many Random Horse would you like [0 - 8]: 8


Random Horse:
Horse Name: Shadow moon,
Horse Symbol: R


Random Horse:
Horse Name: Lightning dream,
Horse Symbol: z


Random Horse:
Horse Name: Flying runner,
Horse Symbol: Z


Random Horse:
Horse Name: Thunder hoof,
Horse Symbol: E


Random Horse:
Horse Name: Majestic shadow,
Horse Symbol: 9


Random Horse:
Horse Name: Galloping hoof,
Horse Symbol: t


Random Horse:
Horse Name: Royal wind,
Horse Symbol: b


Random Horse:
Horse Name: Shadow tail,
Horse Symbol: F
```

Saving them to a file after the race:

```
Saved Horses:
+-----------------+------------+-----------+------------+-------------+----------+
|            Name | Confidence | Character | Total Wins | Total Races | Win Rate |
+-----------------+------------+-----------+------------+-------------+----------+
| Shadow moon     | 0.12       |         R |          0 |           1 | 0.00     |
+-----------------+------------+-----------+------------+-------------+----------+
| Lightning dream | 0.12       |         z |          0 |           1 | 0.00     |
+-----------------+------------+-----------+------------+-------------+----------+
| Flying runner   | 0.12       |         Z |          0 |           1 | 0.00     |
+-----------------+------------+-----------+------------+-------------+----------+
| Thunder hoof    | 0.14       |         E |          1 |           1 | 1.00     |
+-----------------+------------+-----------+------------+-------------+----------+
| Majestic shadow | 0.12       |         9 |          0 |           1 | 0.00     |
+-----------------+------------+-----------+------------+-------------+----------+
| Galloping hoof  | 0.12       |         t |          0 |           1 | 0.00     |
+-----------------+------------+-----------+------------+-------------+----------+
| Royal wind      | 0.12       |         b |          0 |           1 | 0.00     |
+-----------------+------------+-----------+------------+-------------+----------+
| Shadow tail     | 0.12       |         F |          0 |           1 | 0.00     |
+-----------------+------------+-----------+------------+-------------+----------+
```

The test is very clearly passed as the random horses were flawlessly generated and added to their lanes, and the file was successfully updated.

Weather Conditions.

The next feature which I would like to develop is weather on the racetrack. For example, if it is sunny then the horses will be confident, and the horse confidence will not change, However, if it is raining then the horse confidence will be lowered, and this will make horses run slower throughout the race. I would like to have race conditions such as sunny, raining, wet and snow and each of them will affect the horse's confidence a unique amount. At the start of every race a random weather condition will be generated, and the horse confidence will be updated accordingly.

This should be another nice feature to implement into the simulation.

Firstly, I will need an array of weather conditions and have a random number generator pick out an index. This will now be the weather during the race. In a second array I will have at the same index the change that this weather condition makes to the confidence of the horses as decimal multiplier. If it does not change the confidence, then it will just be 1, The closer the multiplier is to 1 the less effect it will have on the confidence of each horse.

If the weather condition chosen happens to be the current weather condition, then we do not make any changes. And move on.

This method I have just described I have shown below.

```java
/**
 * weather during race race conditions affects horse' confidence
 */
private void raceConditions() {
    Random random = new Random();

    String[] weatherConditions = {"Raining", "Wet", "Sunny", "Snow"};
    double[] weatherConditionsEffect = {0.75, 0.85, 1, 0.50};

    int chosen = random.nextInt(weatherConditions.length);
    String newWeatherCondition = weatherConditions[chosen];
    double change = weatherConditionsEffect[chosen];

    if (raceCondition.equals(newWeatherCondition)) {
        System.out.println("\n\nWeather conditions have not changed. ("+raceCondition+")");
        return;
    }

    changeHorseConfidenceDueToConditions(newWeatherCondition, change);
}
```

Now that we can have weather this method assumes we have a race Condition variable inside of the program already so I have had to add one to the race class, I have assumed it to be sunny at the start, but it may change before the first race even happens.

```java
private String raceCondition = "Sunny";
```

Then if the weather does end up changing then we need to be able to tell the user the effect this will have on the horses and then change the confidence of each horse according to the weather.

Weather affects confidence.

```
/**
 *
 * Change horse confidence
 * and output information about the weather
 * @param weather new weather condition
 * @param change the scale factor by which the horse confidence will be
 */
private void changeHorseConfidenceDueToConditions(String weather, double change) {
    switch (weather) {
        case "Raining" -> System.out.println(x:"\nHorse confidence is decreased by 25%, due to rain.");
        case "Wet"     -> System.out.println(x:"\nHorse confidence is decreased by 15%, due to a wet track.");
        case "Sunny"   -> System.out.println(x:"\nHorse confidence is not affected, it is sunny.");
        case "Snow"    -> System.out.println(x:"\nHorse confidence is decreased by 50% due to snow.");
    }
    for (Horse horse : horses) {
        if (horse == null) continue;
        horse.setConfidence(horse.getConfidence()*change);
    }
    raceCondition = weather;
}
```

Then we need to change the weather variable accordingly as well. Now at the start of each race it will display the weather and how it will affect the confidence of each horse.

A couple example runs are shown below:

```
Weather conditions have not changed. (Sunny)
```

```
Horse confidence is decreased by 50% due to snow.
```

Betting System.

The next feature which I have decided to start implementing is a betting system. The user will start the program and have £100 to place bets on races but only if they would like to, this is an optional feature. I am planning to make the amount of money the user able to win proportional to the horse confidence and the horse win rate. So, a horse with a higher confidence has a higher chance of falling but it will be running faster so the output should be higher amount of money which the user can win. I decided to start this with a class called Betting System. This code which I will write will be shown below.

Betting Class.

```java
/**
 * Class to handle all of the bets
 * @author Peter Bojthe
 * @version 18/04/25
 */
public class BettingSystem {
    // Balance initially set to 100.
    public static double balance = 100.0;

    /**
     * Calculate the possible winnings of a horse
     * @param horse horse we are calculating the winnings of
     * @param bet the amount of money placed on the race by the user
     * @return return the possible winnings by the hore
     */
    public static double calculateWinnings(Horse horse, double bet) {
        return ((horse.getConfidence()+horse.getWinRate()+1.1)*bet);
    }

    /**
     * add to balance
     * @param wonMoney money to add to balance
     */
    public static void addWinnings(double wonMoney) {
        balance = balance + wonMoney;
    }

    /**
     * decrease balance
     * @param lostMoney amount to decrease balcance by
     */
    public static void removeLoss(double lostMoney) {
        balance = balance - lostMoney;
    }
}
```

This method will have a public static variable of type double which will be the balance of the user and the total amount of money which they currently have. The calculate winnings method will calculate how much money the user can win if they place a bet with a certain amount of money on a specific horse. This will be calculated for all horses to ensure the user is informed of all possible horses they can bet on for the race. The next method add winnings adds all the money the user won to the balance. The next method called remove loss removes the money from the account when the user places a bet.

These methods I think is all I need to control the flow of money around the program; the user input and simulation flow will be handled by other methods.

Before I start work on other methods, I noticed I will need more attributes related to the horse. I need to be able to keep track of how much money a certain horse will be able to earn the user if they win so I will require a variable to store the number of the possible winnings. And then I also noticed I need to keep track of which horse the user has placed a bet on. I think the simplest way to do this is to have a private attribute which is a Boolean which is only set to true if the user has placed a bet on the horse. These methods will be initially set to 0.0 and false respectively and here are the setters and getters for them.

Betting attributes getters and setters.

```java
/**
 * Retrieves the current possible winnings of a horse
 * @return double The amount of money a horse could win
 */
public double getWinnings() {
    return this.winnings;
}

/**
 * Set the winnings of a horse to 2 decimal places
 * @param winnings type double number we are settign winnings to
 */
public void setWinnings(double winnings) {
    this.winnings = Math.round(winnings * 100.0) / 100.0;
}

/**
 * Retrieves the boolean flag which determines
 * if the user has placed a bet on this horse
 * @return boolean which checks if the horse has had a bet placed on it
 */
public boolean isBetPlacedOn() {
    return this.betPlacedOn;
}

/**
 * Set the variable to true or false
 * @param betPlacedOn type boolean which determines if a horse has
 * had a bet placed on it
 */
public void setBetPlacedOn(boolean betPlacedOn) {
    this.betPlacedOn = betPlacedOn;
}
```

4

More horse attributes for betting.

The new getter and setter methods for these variables are shown above and they have also been added to the constructor, so the constructor now has these 2 new lines.

Attributes.

```
this.betPlacedOn = false;
this.winnings = 0.0;
```

These two lines have been added to both constructors now for both the horses which are read from the file and the horses which the user decided to create.

Getters and Setters.

```
/**
 * Retrieves the current possible winnings of a horse
 * @return double The amount of money a horse could win
 */
public double getWinnings() {
    return this.winnings;
}

/**
 * Set the winnings of a horse to 2 decimal places
 * @param winnings type double number we are settign winnings to
 */
public void setWinnings(double winnings) {
    this.winnings = Math.round(winnings * 100.0) / 100.0;
}

/**
 * Retrieves the boolean flag which determines
 * if the user has placed a bet on this horse
 * @return boolean which checks if the horse has had a bet placed on it
 */
public boolean isBetPlacedOn() {
    return this.betPlacedOn;
}

/**
 * Set the variable to true or false
 * @param betPlacedOn type boolean which determines if a horse has
 * had a bet placed on it
 */
public void setBetPlacedOn(boolean betPlacedOn) {
    this.betPlacedOn = betPlacedOn;
}
```

Ask user to place bets.

Now with these variables and the Betting System class I can now create the methods which take care of the user placing bets.

```
/**
 * Ask the user if they want to place a bet
 */
private void askToPlaceBet() {
    if (askYesNo(statement:"\n\nWould you like to bet on a horse yes [1] no [0]: ")) {
        checkWinnings();
    }
}
```

The first step is to find out if the user even wants to place a bet. If they answer yes, then I call a method which ask them how much of their total money they want to place on this race. This obviously neds to be check as it cannot be negative and it cannot be greater than the total balance and it is £0 then I can just return.

```
/**
 * calculate how much money the user would win
 * if their choice of horse wins
 *
 */
private void checkWinnings() {
    System.out.println("The balance: "+BettingSystem.balance);
    double usersBet = placeBet(statement:"How much money are you putting on this race: ");
    if (usersBet == 0.0) return;
    for (Horse horse : horses) {
        if (horse == null) continue;
        horse.setWinnings(BettingSystem.calculateWinnings(horse, usersBet));
        System.out.println("If "+horse.getName()+" wins then the payout will be £"+horse.getWinnings());
    }
    chooseHorseToPlaceBetOn();
}
```

I now ask the user to place their bet and for each horse I calculate how much that horse would win the user if it were to win the race. And I set this value to the horse's attribute which holds this piece of data. Once this has been done for all horses, I will call another method which lets the user choose the horse they want to place the bet on.

```java
/**
 * this method ask for the amount the user want to spend on the race and
 * returns this value
 * @param statement the prompt given to the user
 * @return the amount of money put on the race
 */
@Override
public double placeBet(String statement) {
    if (BettingSystem.balance == 0.0) {
        System.out.println(x:"Balance is £0. Cannot place Bets.");
        return 0.0;
    }

    double bet = inputDouble(statement);
    while (bet <= 0.0 || bet > BettingSystem.balance) {
        if (bet <= 0.0) System.out.println(x:"Bet cannot be less than £0.");
        if (bet > BettingSystem.balance) System.out.println(x:"Not eneough money to place bet.");
        System.out.println(x:"Try again.");
        bet = inputDouble(statement);
    }
    System.out.println(x:"Bet has been placed.");
    BettingSystem.removeLoss(bet);
    System.out.println("Current Balance: "+BettingSystem.balance+"\n");
    return bet;
}
```

To place a bet on the race I have written this method which is inside of the User Input class and is inside of the User Input Interface This method is what returns the validated amount the user is betting on the race. This value is then returned to the race class, and we decrease the amount of money in the balance by the amount of money they have put on the race.

```java
/**
 * User must enter the name of a horse
 * If they have said they want to place a bet
 * Name must exist within the horses currently racing
 */
private void chooseHorseToPlaceBetOn() {
    String name = inputString(statement:"Enter the name of the horse you want to place the bet on: ");
    while (true) {
        for (Horse horse : horses) {
            if (horse == null) continue;
            if (horse.getName().equals(name)) {
                horse.setBetPlacedOn(betPlacedOn:true);
                return;
            }
        }
        System.out.println(x:"Invalid Name.");
        name = inputString(statement:"Enter the name of the horse you want to place the bet on: ");
    }
}
```

In this method above the user can now pick a horse based on the name of the horse and then the variable which controls if a horse has had a bet placed on it is set to true and that horse is now marked inside of the program so we have something to check for when the race is over.

Update to show winner method (to show bets and update balance).

```
/**
 * Displays the winner of the race and adjusts the confidence of the horses.
 * And adjust the total races, total wins, and win rate of the horse
 * Updates the user balance if they have placed a bet on the race
 */
private void showWinner() throws IOException {
    for (Horse horse : horses) {
        if (horse == null) continue;
        horse.setTotalRaces(horse.getTotalRaces()+1);
        if (raceWonBy(horse)) {
            System.out.println("\n\n"+horse.getName()+" has won the Race!");
            horse.setConfidence(horse.getConfidence()*1.1);
            horse.setTotalWins(horse.getTotalWins()+1);
            if (horse.isBetPlacedOn()) {
                BettingSystem.balance = horse.getWinnings();
                System.out.println("Current balance: "+BettingSystem.balance);
            }
        } else {
            horse.setConfidence(horse.getConfidence()*0.9);
            if (horse.isBetPlacedOn()) {
                System.out.println(x:"The horse you placed the bet on has lost");
                System.out.println("Current balance: "+BettingSystem.balance);
            }
        }
        horse.setWinRate(horse.getTotalWins(), horse.getTotalRaces());
        HorseDetailsFileHandling.updateHorseInFile(horse);
    }
}
```

The method above has been edited quite a few times during the development of this program, and it has been shown before in this document but now we also check which horse has had the bet placed on it and if it was the winning horse then we add the winnings to the user balance otherwise we tell them that their horse has not won and we show them their balance with the money removed.

Now these attributes which I have added to the horse race also need to be reset between races and this is an issue which I caught during testing. Not resetting these values means that the betting system does not work for consecutive races.

Remove bets once race is over.

So here is the method I have written to reset these variables between races.

```
/**
 * Reset all the variables related to the
 * Betting System
 */
private void removeAllBets() {
    for (Horse horse : horses) {
        if (horse == null) continue;
        horse.setBetPlacedOn(betPlacedOn:false);
        horse.setWinnings(winnings:0.0);
    }
}
```

All this method needs to do is remove the all the bets placed on the horses and reset their winnings back to £0.

The amount of money that the user can win depends on the horse confidence and the horse confidence can be affected by many things such as the weather and how many races it has won or lost.

Timing the race.

The very last feature which I have decided to add is a timer to time how long it took the quickest horse to finish the race. This is a very quick feature to add because I need to surround the race while loop by timers which start when the race starts and finish when the race loop finishes, then I can output to the user the amount of time it took to complete the race.

Race timer.

```
// This is the Race
long startTime = System.nanoTime(); // Start of race timer
while (!finishedRace) {
    moveAllHorses();
    printRace();
    finishedRace = raceFinished();

    // Wait bwtween horse moves
    try {
        TimeUnit.MILLISECONDS.sleep(timeout:100);
    } catch (InterruptedException e) {}
} // END RACE

long endTime = System.nanoTime(); // End of race timer
long raceDurationNano = endTime - startTime;
double raceDurationSeconds = raceDurationNano / 1_000_000_000.0;
raceDurationSeconds = Math.round(raceDurationSeconds*100.0)/100.0;
System.out.println("The race was completed in "+raceDurationSeconds+" seconds.");
```

And this is now how the while loop which controls the race looks. First, we start the timer right at the top then we start the race almost simultaneously. Then I can run the race, and this will stop either when a horse has finished a race or if all the horses have fell. Then this will end the while loop, and the end Time variable will end the timer. I can now find the race duration and convert this to seconds and round it to 2 decimal places just like all other decimal place numbers within the program. Then I can output this number to the user and show them how long it took to finish the race.

Now that this feature has been added this is the last version of the start Race method which controls the whole simulation.

While loop which controls the race.

```java
/**
 * Starts the race simulation.
 * Horses are brought to the start and repeatedly moved forward until the race is finished.
 * @throws IOException if there is an input/output (File Handling) error during the race simulation.
 */
public void startRace() throws IOException {
    boolean finishedSimulation = false;
    boolean finishedRace = false;

    chooseSavedHorse();
    askUserToGenerateRandomHorse();
    raceLength = chooseTrackLength(statement:"\nChoose the length of race [25m - 100m]: ");
    // Only Create Horses if 2 or more horses have not yet bee chosen
    if (horses.size() <= 1) {
        createHorses();
    }

    raceConditions();
    askToPlaceBet();
    resetHorsesPosition();
    while (!finishedSimulation) {
        // Before Race;
        try {
            System.out.print(s:"\nRace will start soon...");
            TimeUnit.MILLISECONDS.sleep(timeout:2000);
        } catch (InterruptedException e) {}

        // This is the Race
        long startTime = System.nanoTime(); // Start of race timer
        while (!finishedRace) {
            moveAllHorses();
            printRace();
            finishedRace = raceFinished();

            // Wait bwtween horse moves
            try {
                TimeUnit.MILLISECONDS.sleep(timeout:100);
            } catch (InterruptedException e) {}
        } // END RACE

        long endTime = System.nanoTime(); // End of race timer
        long raceDurationNano = endTime - startTime;
        double raceDurationSeconds = raceDurationNano / 1_000_000_000.0;
        raceDurationSeconds = Math.round(raceDurationSeconds*100.0)/100.0;
        System.out.println("The race was completed in "+raceDurationSeconds+" seconds.");

        // End of race PROCEDURES
        showWinner();
        resetHorsesPosition();
        showRaceDetails();
        removeAllBets();

        // Between Races or Quit program
        finishedSimulation = askYesNo(statement:"\nSTOP SIMULATION:  yes [1], no [0]: ");
        if (!finishedSimulation) {
            finishedRace = false;
            if (askYesNo(statement:"\nWould you like to make changes to the next simulation yes [1], no [0]: ")) {
                changeRaceDetails();
            }
            System.out.println(x:"\n\nA new race will begin...");
            raceConditions();
            askToPlaceBet();
        }
    }
}
```

This is now the end of the development of the horse Racing Simulator (Part I).

Final Testing.

Here I will run the finalised program many times and check all features to make sure they work as they must.

Displaying the winner.

To test this, I will start a race and check if the winning horses name has been correctly shown to the user.

```
==========================
|                        T| Lane: 1, TEST HORSE (Current Confidence 0.25)
|                       E | Lane: 2, TEST HORSE 2 (Current Confidence 0.25)
==========================
The race was completed in 13.63 seconds.


TEST HORSE has won the Race!
```

And as we can see above the horse with name 'TEST HORSE' has won the race and the name has been successfully shown to the user.

Selecting track length.

Starting a 30m long race.

```
Choose the length of race [25m - 100m]: 30
```

```
================================
|  V                           | Lane: 1, Brave blaze (Current Confidence 0.13)
|8                             | Lane: 2, Red prince (Current Confidence 0.13)
|h                             | Lane: 3, Spirit mane (Current Confidence 0.13)
|  d                           | Lane: 4, Shadow dream (Current Confidence 0.13)
| M                            | Lane: 5, Mystic tail (Current Confidence 0.13)
|Y                             | Lane: 6, Royal prince (Current Confidence 0.13)
| 2                            | Lane: 7, Flying storm (Current Confidence 0.13)
| 0                            | Lane: 8, Silver runner (Current Confidence 0.13)
================================
```

Starting a 100m long race.

```
Choose the length of race [25m - 100m]: 100
```

```
================================================================================================
| k                                                                                            | Lane: 1, Dancing flash (Current Confidence 0.19)
|K                                                                                             | Lane: 2, Galloping tail (Current Confidence 0.19)
|  9                                                                                           | Lane: 3, Thunder tail (Current Confidence 0.19)
| 6                                                                                            | Lane: 4, White moon (Current Confidence 0.19)
| h                                                                                            | Lane: 5, Black mane (Current Confidence 0.19)
| z                                                                                            | Lane: 6, Brave tail (Current Confidence 0.19)
|  7                                                                                           | Lane: 7, Shadow sun (Current Confidence 0.19)
|  r                                                                                           | Lane: 8, Galloping blaze (Current Confidence 0.19)
================================================================================================
```

This is a successful test.

Testing Confidence and Confidence Bounds.

The user does not set the confidence level of the horse in the program, it is set to 0.25 at the creation of the horses and then it is affected by weather and race performance, so to test that it does not go above or below of 1 and 0 I need to test it using another block of code and not user input.

The code and the result will be shown below:

```
System.out.println(x:"1) Testing the Constructor: ");
Horse TEST_1 = new Horse(horseName:"NAME1", horseConfidence:1.0, horseSymbol:'Q', total…0, 0, 0, 1);
Horse TEST_2 = new Horse(horseName:"NAME2", horseConfidence:1.01, horseSymbol:'W', total…0, 0, 0, 2);
Horse TEST_3 = new Horse(horseName:"NAME3", horseConfidence:0.0, horseSymbol:'E', total…0, 0, 0, 3);
Horse TEST_4 = new Horse(horseName:"NAME4", -0.01, horseSymbol:'R', totalWins:0, totalRaces:0, …0, 4);

System.out.println(TEST_1.getConfidence());
System.out.println(TEST_2.getConfidence());
System.out.println(TEST_3.getConfidence());
System.out.println(TEST_4.getConfidence());

System.out.println(x:"\n2) Testing setter method: ");
TEST_1.setConfidence(newConfidence:1.99);
TEST_2.setConfidence(-0.99);
TEST_3.setConfidence(newConfidence:0.123);
TEST_4.setConfidence(newConfidence:0.234);

System.out.println(TEST_1.getConfidence());
System.out.println(TEST_2.getConfidence());
System.out.println(TEST_3.getConfidence());
System.out.println(TEST_4.getConfidence());
```

This is the code above I used to test the constructor and the setter method to make sure the confidence can never go out of bounds from 0 to 1 and I also made sure that it is always rounded to 2 decimal places in the program.

The results are below.

```
1) Testing the Constructor:
1.0
1.0
0.0
0.0

2) Testing setter method:
1.0
0.0
0.12
0.23
```

And as we can see this test is successful. The confidence of the horse can never be below 0 and it can never be greater that 1, and it is always rounded to 2 decimal places in the program.

User adding their own horses.

```
Number of lanes must be greater than 2 to start race simulation.
How many lanes would you like [2 - 8]: 4
How many horses would you like: 4

Choose a lane to add this horse to [1 - 4]: 1

Horse Name: TEST 1
Horse Character: 1

Choose a lane to add this horse to [1 - 4]: 2

Horse Name: TEST 2
Horse Character: 2

Choose a lane to add this horse to [1 - 4]: 3

Horse Name: TEST 3
Horse Character: 3

Choose a lane to add this horse to [1 - 4]: 4

Horse Name: TEST 4
Horse Character: 4
```

4 test horses have been added to 4 lanes.

Result:

```
============================
|   ?                      | Lane: 1, TEST 1 (Current Confidence 0.25)
|               2          | Lane: 2, TEST 2 (Current Confidence 0.25)
|            ?             | Lane: 3, TEST 3 (Current Confidence 0.25)
|                         4| Lane: 4, TEST 4 (Current Confidence 0.25)
============================
The race was completed in 10.99 seconds.
```

This test is also successful.

Empty Lanes.

Empty lanes occur when the user decides to have more lanes than horses.

I will test this with 6 lanes and 3 horses; the expected result is to have 3 empty lanes.

```
Number of lanes must be greater than 2 to start race simulation.
How many lanes would you like [2 - 8]: 6
How many horses would you like: 3

Choose a lane to add this horse to [1 - 6]: 1

Horse Name: TEST 1
Horse Character: 1

Choose a lane to add this horse to [1 - 6]: 3

Horse Name: TEST 2
Horse Character: 2

Choose a lane to add this horse to [1 - 6]: 5

Horse Name: TEST 3
Horse Character: 3
```

```
============================
| 1                        | Lane: 1, TEST 1 (Current Confidence 0.13)
|                          | Empty Lane
|  2                       | Lane: 3, TEST 2 (Current Confidence 0.13)
|                          | Empty Lane
| 3                        | Lane: 5, TEST 3 (Current Confidence 0.13)
|                          | Empty Lane
============================
```

And this test is also successful.

Edge Cases for lane choice and horse choice.

The lane choice only allows lanes from 2 to 8, and horse choice is only from 2 to the number of lanes.

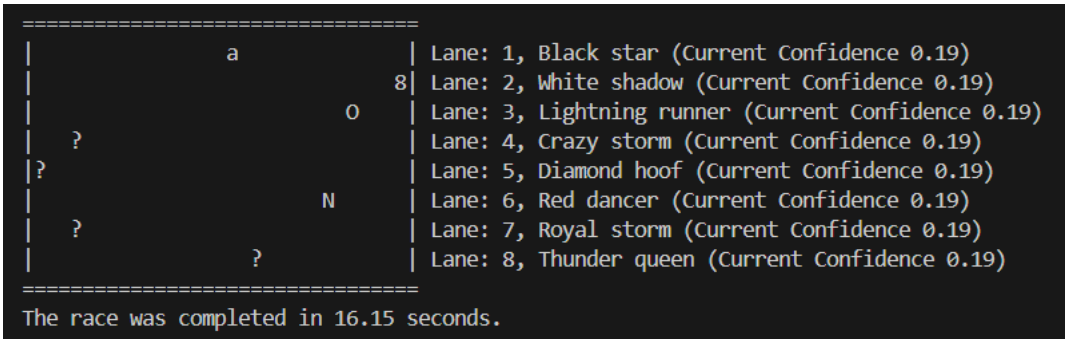
```
Number of lanes must be greater than 2 to start race simulation.
How many lanes would you like [2 - 8]: 1
Number of lanes must be greater than 2 to start race simulation.
How many lanes would you like [2 - 8]: 0
Number of lanes must be greater than 2 to start race simulation.
How many lanes would you like [2 - 8]: -1
How many lanes would you like [2 - 8]: 9
Number of lanes must be greater than 2 to start race simulation.
How many lanes would you like [2 - 8]: 10
Number of lanes must be greater than 2 to start race simulation.
How many lanes would you like [2 - 8]: 5
How many horses would you like: 0
Number of horses must be greater than 2 and less than or equal to the number of lanes.
How many horses would you like: -1
How many horses would you like: 1
Number of horses must be greater than 2 and less than or equal to the number of lanes.
How many horses would you like: 5
```

This has now been tested with the edge cases, and 1 change which I would like to make is that if the number of lanes chosen is greater than 8 tell them it must be less than 8 and not greater than 2. But this is subtle change and does not affect game logic.

Random horses added to race.

To test this, I will create 8 random horse and fill up all lanes.

```
How many Random Horse would you like [0 - 8]: 8


Random Horse:
Horse Name: Black star,
Horse Symbol: a


Random Horse:
Horse Name: White shadow,
Horse Symbol: 8


Random Horse:
Horse Name: Lightning runner,
Horse Symbol: O


Random Horse:
Horse Name: Crazy storm,
Horse Symbol: i


Random Horse:
Horse Name: Diamond hoof,
Horse Symbol: 8


Random Horse:
Horse Name: Red dancer,
Horse Symbol: N


Random Horse:
Horse Name: Royal storm,
Horse Symbol: K


Random Horse:
Horse Name: Thunder queen,
Horse Symbol: h
```



And this test is also successful as all lanes have been filled up with random horses.

Horses added from file to race.

To test this, I will add the 5th horse in the file to the race in the first lane and add a random horse in the 2nd lane to start the race.

```
Would you like to use a previously saved horse yes [1], no [0]: 1



Saved Horses:
+------------------+------------+-----------+------------+-------------+----------+
|             Name | Confidence | Character | Total Wins | Total Races | Win Rate |
+------------------+------------+-----------+------------+-------------+----------+
| Shadow moon      | 0.12       |         R |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Lightning dream  | 0.12       |         z |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Flying runner    | 0.02       |         Z |          0 |           6 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Thunder hoof     | 0.17       |         1 |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Majestic shadow  | 0.05       |         9 |          0 |           3 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Galloping hoof   | 0.21       |         8 |          1 |           1 | 1.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Royal wind       | 0.12       |         b |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Shadow tail      | 0.12       |         F |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Dancing fire     | 0.10       |         U |          0 |           5 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Silver tail      | 0.10       |         Q |          0 |           5 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Galloping whisper | 0.14       |         D |          2 |           5 | 0.40     |
+------------------+------------+-----------+------------+-------------+----------+
| Black king       | 0.12       |         6 |          1 |           5 | 0.20     |
+------------------+------------+-----------+------------+-------------+----------+
| Red chaser       | 0.12       |         0 |          1 |           5 | 0.20     |
+------------------+------------+-----------+------------+-------------+----------+
| Mystic queen     | 0.17       |         T |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Midnight whisper | 0.17       |         6 |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Golden king      | 0.12       |         H |          1 |           5 | 0.20     |
+------------------+------------+-----------+------------+-------------+----------+

Enter the row number of the horse you want to use: 5

This horse will be added to lane 1
```

The 5th horse in the file is majestic shadow.

```
================================
|9                             | Lane: 1, Majestic shadow (Current Confidence 0.04)
| q                            | Lane: 2, Diamond runner (Current Confidence 0.21)
================================
```

Since Majestic shadow has been added from the file system into the race in lane 1 this test has also been passed.

Horsed added from race to file.

To test this, I will create a race with 2 test horses and add both into the file between races.

```
Number of lanes must be greater than 2 to start race simulation.
How many lanes would you like [2 - 8]: 2
How many horses would you like: 2

Choose a lane to add this horse to [1 - 2]: 1

Horse Name: TEST 1 FILE
Horse Character: 1

Choose a lane to add this horse to [1 - 2]: 2

Horse Name: TEST 2 FILE
Horse Character: 2
```

Two horses running the race:

```
 ============================
 |              ?           | Lane: 1, TEST 1 FILE (Current Confidence 0.25)
 |                         2| Lane: 2, TEST 2 FILE (Current Confidence 0.25)
 ============================
 The race was completed in 12.92 seconds.
```

Both horses getting saved in the CSV:

```
Would you like to save any of the horse details yes [1], no [0]: 1
Lane: 1, TEST 1 FILE with confidence 0.23
Lane: 2, TEST 2 FILE with confidence 0.28

Enter the lane number of the horse you want to save: 2



Saved Horses:
+------------------+------------+-----------+------------+-------------+----------+
|             Name | Confidence | Character | Total Wins | Total Races | Win Rate |
+------------------+------------+-----------+------------+-------------+----------+
| Shadow moon      | 0.12       |         R |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Lightning dream  | 0.12       |         z |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Flying runner    | 0.02       |         Z |          0 |           6 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Thunder hoof     | 0.17       |         l |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Majestic shadow  | 0.04       |         9 |          0 |           4 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Galloping hoof   | 0.21       |         8 |          1 |           1 | 1.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Royal wind       | 0.12       |         b |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Shadow tail      | 0.12       |         F |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Dancing fire     | 0.10       |         U |          0 |           5 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Silver tail      | 0.10       |         Q |          0 |           5 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Galloping whisper | 0.14       |         D |          2 |           5 | 0.40     |
+------------------+------------+-----------+------------+-------------+----------+
| Black king       | 0.12       |         6 |          1 |           5 | 0.20     |
+------------------+------------+-----------+------------+-------------+----------+
| Red chaser       | 0.12       |         0 |          1 |           5 | 0.20     |
+------------------+------------+-----------+------------+-------------+----------+
| Mystic queen     | 0.17       |         T |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Midnight whisper | 0.17       |         6 |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| Golden king      | 0.12       |         H |          1 |           5 | 0.20     |
+------------------+------------+-----------+------------+-------------+----------+
| TEST 1 FILE      | 0.23       |         1 |          0 |           1 | 0.00     |
+------------------+------------+-----------+------------+-------------+----------+
| TEST 2 FILE      | 0.28       |         2 |          1 |           1 | 1.00     |
+------------------+------------+-----------+------------+-------------+----------+
```

And since both horse is now showing up in the file this test is also passed, and horses can be saved.

Horse confidence affected by weather.

Different weather types change the hors confidence a different amount.

```
Horse confidence is decreased by 50% due to mud.
```

In the first test the weather happened to be muddy, and the confidence has decreased by 50%.

```
==========================
| 8                      | Lane: 1, Shadow whisper (Current Confidence 0.13)
|2                       | Lane: 2, Silver whisper (Current Confidence 0.13)
|    k                   | Lane: 3, Dancing whisper (Current Confidence 0.13)
|  Y                     | Lane: 4, Diamond whisper (Current Confidence 0.13)
|J                       | Lane: 5, Lightning moon (Current Confidence 0.13)
|  A                     | Lane: 6, Mystic dancer (Current Confidence 0.13)
| Q                      | Lane: 7, Galloping shadow (Current Confidence 0.13)
|9                       | Lane: 8, Wild mane (Current Confidence 0.13)
==========================
```

And this is correct because the first confidence of each random horse was 0.25 and 50% of that is 0.125 and rounding that to 2 decimal places is 0.13. Let's run the program again and test another weather condition.

```
Horse confidence is decreased by 15%, due to a wet track.
```

In the second test the track is now wet, so the confidence has decreased by 15%. Since the confidence starts at 0.25 and were rounding to 2 decimal places the excepted value we should get is 0.21.

```
==========================
|    o                   | Lane: 1, Black moon (Current Confidence 0.21)
| D                      | Lane: 2, Spirit chaser (Current Confidence 0.21)
| Q                      | Lane: 3, Shadow king (Current Confidence 0.21)
|O                       | Lane: 4, Dancing flash (Current Confidence 0.21)
|e                       | Lane: 5, Black sun (Current Confidence 0.21)
|   4                    | Lane: 6, Brave flash (Current Confidence 0.21)
| W                      | Lane: 7, Black dancer (Current Confidence 0.21)
| 1                      | Lane: 8, Dancing prince (Current Confidence 0.21)
==========================
```

Since the confidence is 0.21 this test is also passed.

Horse confidence affected by losing/ winning race.

To test this, I will create a race with 8 horses and make them race 3 times, and we should notice a variance in each horse confidence as they race, keep in mind this is also affected by the weather.

```
==========================
|b                       | Lane: 1, Royal runner (Current Confidence 0.25)
| 8                      | Lane: 2, Crazy dancer (Current Confidence 0.25)
|  3                     | Lane: 3, Thunder shadow (Current Confidence 0.25)
|  y                     | Lane: 4, White shadow (Current Confidence 0.25)
|    3                   | Lane: 5, Golden prince (Current Confidence 0.25)
|?                       | Lane: 6, Shadow king (Current Confidence 0.25)
|     ?                  | Lane: 7, Crazy mane (Current Confidence 0.25)
|  y                     | Lane: 8, Shadow wind (Current Confidence 0.25)
==========================
```

In this race it was sunny, so confidence was not affected, and the winning horse had their confidence increase and all other horses had their confidence decrease.

```
Royal runner is in lane 1, Confidence: 0.23
Crazy dancer is in lane 2, Confidence: 0.23
Thunder shadow is in lane 3, Confidence: 0.23
White shadow is in lane 4, Confidence: 0.23
Golden prince is in lane 5, Confidence: 0.28
Shadow king is in lane 6, Confidence: 0.23
Crazy mane is in lane 7, Confidence: 0.23
Shadow wind is in lane 8, Confidence: 0.23
```

Now I will make them race again.

```
Royal runner is in lane 1, Confidence: 0.21
Crazy dancer is in lane 2, Confidence: 0.21
Thunder shadow is in lane 3, Confidence: 0.21
White shadow is in lane 4, Confidence: 0.21
Golden prince is in lane 5, Confidence: 0.31
Shadow king is in lane 6, Confidence: 0.21
Crazy mane is in lane 7, Confidence: 0.21
Shadow wind is in lane 8, Confidence: 0.21
```

The same horse has won again, so it has had a confidence increase and all other horse have had their confidence decrease.

Now I will make them race again one last time.

This time it was snowing so the confidence decreased by 50% for all horses

```
==========================
|         b               | Lane: 1, Royal runner (Current Confidence 0.11)
|                 8       | Lane: 2, Crazy dancer (Current Confidence 0.11)
|               3         | Lane: 3, Thunder shadow (Current Confidence 0.11)
|                       y | Lane: 4, White shadow (Current Confidence 0.11)
|                         3| Lane: 5, Golden prince (Current Confidence 0.16)
|             L           | Lane: 6, Shadow king (Current Confidence 0.11)
|       ?                 | Lane: 7, Crazy mane (Current Confidence 0.11)
|               y         | Lane: 8, Shadow wind (Current Confidence 0.11)
==========================
The race was completed in 16.67 seconds.
```

And the final confidence of each horse now is:

```
Royal runner is in lane 1, Confidence: 0.1
Crazy dancer is in lane 2, Confidence: 0.1
Thunder shadow is in lane 3, Confidence: 0.1
White shadow is in lane 4, Confidence: 0.1
Golden prince is in lane 5, Confidence: 0.18
Shadow king is in lane 6, Confidence: 0.1
Crazy mane is in lane 7, Confidence: 0.1
Shadow wind is in lane 8, Confidence: 0.1
```

This I believe is enough to call this test successful as well.

Lanes removed.

To remove lanes, I will create a race with 8 lanes and remove 4.

```
===========================
|  ?                      | Lane: 1, Silver blaze (Current Confidence 0.19)
|            ?            | Lane: 2, Dancing blaze (Current Confidence 0.19)
|                     7   | Lane: 3, Royal prince (Current Confidence 0.19)
|         ?               | Lane: 4, Crazy blaze (Current Confidence 0.19)
|                      I  | Lane: 5, Crazy dream (Current Confidence 0.19)
|  ?                      | Lane: 6, Mystic dream (Current Confidence 0.19)
|                        0| Lane: 7, Red star (Current Confidence 0.19)
|                   0     | Lane: 8, Silver chaser (Current Confidence 0.19)
===========================
The race was completed in 16.23 seconds.
```

This is the race with 8 lanes.

```
Would you like to Remove any lanes yes [1], no [0]: 1

There are currently 8 lanes.
The number of lanes cannot be less than 2
Enter the lane number you want to remove: 8
Stop removing Lanes yes [1], no [0]0

There are currently 7 lanes.
The number of lanes cannot be less than 2
Enter the lane number you want to remove: 7
Stop removing Lanes yes [1], no [0]0

There are currently 6 lanes.
The number of lanes cannot be less than 2
Enter the lane number you want to remove: 6
Stop removing Lanes yes [1], no [0]0

There are currently 5 lanes.
The number of lanes cannot be less than 2
Enter the lane number you want to remove: 5
Stop removing Lanes yes [1], no [0]1
```

I have decided to remove lanes 5 to 8 including so only the lanes 1 to 4 including should remain.

```
==========================
|   2                    | Lane: 1, Silver blaze (Current Confidence 0.14)
|  j                     | Lane: 2, Dancing blaze (Current Confidence 0.14)
| 7                      | Lane: 3, Royal prince (Current Confidence 0.14)
| K                      | Lane: 4, Crazy blaze (Current Confidence 0.14)
==========================
```

And this test is also successful.

Lanes added.

To test this, I will create a race with 4 lanes all full and add 4 more lanes.

```
==========================
|    S                   | Lane: 1, Lightning shadow (Current Confidence 0.13)
|     9                  | Lane: 2, Red chaser (Current Confidence 0.13)
|   0                    | Lane: 3, Galloping wind (Current Confidence 0.13)
|       i                | Lane: 4, Midnight storm (Current Confidence 0.13)
==========================
```

Above is the race with 4 lanes.

```
Would you like to Add any lanes yes [1], no [0]: 1

There are currently 5 lanes.
Would you like to add a lane yes [1], no [0]1

There are currently 6 lanes.
Would you like to add a lane yes [1], no [0]1

There are currently 7 lanes.
Would you like to add a lane yes [1], no [0]1

There are currently 8 lanes.
Would you like to add a lane yes [1], no [0]0
```

Above is me adding lanes.

Below is the result.

```
==========================
| S                      | Lane: 1, Lightning shadow (Current Confidence 0.1)
|  9                     | Lane: 2, Red chaser (Current Confidence 0.1)
|   0                    | Lane: 3, Galloping wind (Current Confidence 0.1)
|  i                     | Lane: 4, Midnight storm (Current Confidence 0.12)
|                        | Empty Lane
|                        | Empty Lane
|                        | Empty Lane
|                        | Empty Lane
==========================
```

Lanes have been added; this test is successful.

Horses removed.

I will create a with 8 horses to test this and remove 2 random ones to test this; this should result in a race with 8 lanes and 6 horses.

```
============================
|  K                       | Lane: 1, Mystic hoof (Current Confidence 0.21)
|  U                       | Lane: 2, Flying chaser (Current Confidence 0.21)
| w                        | Lane: 3, Diamond whisper (Current Confidence 0.21)
|  2                       | Lane: 4, Lightning flash (Current Confidence 0.21)
|  0                       | Lane: 5, Silver prince (Current Confidence 0.21)
|   4                      | Lane: 6, Thunder star (Current Confidence 0.21)
|8                         | Lane: 7, Shadow queen (Current Confidence 0.21)
|0                         | Lane: 8, Royal blaze (Current Confidence 0.21)
============================
```

And below is where I have decided to remove horses in lanes 3 and 7.

```
Would you like to Remove any horses yes [1], no [0]: 1

Lanes with Horses:
This lane has a horse: 1
This lane has a horse: 2
This lane has a horse: 3
This lane has a horse: 4
This lane has a horse: 5
This lane has a horse: 6
This lane has a horse: 7
This lane has a horse: 8

Enter the lane number of the horse you want to remove: 7

Stop removing horses yes [1], no [0]: 0

Lanes with Horses:
This lane has a horse: 1
This lane has a horse: 2
This lane has a horse: 3
This lane has a horse: 4
This lane has a horse: 5
This lane has a horse: 6
This lane has a horse: 8

Enter the lane number of the horse you want to remove: 3

Stop removing horses yes [1], no [0]: 1
```

And here is the result below.

```
==========================
|K                       | Lane: 1, Mystic hoof (Current Confidence 0.1)
|  U                     | Lane: 2, Flying chaser (Current Confidence 0.1)
|                        | Empty Lane
|2                       | Lane: 4, Lightning flash (Current Confidence 0.12)
| 0                      | Lane: 5, Silver prince (Current Confidence 0.1)
|    4                   | Lane: 6, Thunder star (Current Confidence 0.1)
|                        | Empty Lane
|  0                     | Lane: 8, Royal blaze (Current Confidence 0.1)
==========================
```

As expected, we have now empty lanes where we previously had the horses we have removed.

This test is successful.

Horses added.

For this test I will create a race with 5 lanes and 2 horses, and I will add 2 extra horses.

```
==========================
|    1                   | Lane: 1, 1 (Current Confidence 0.19)
|   2                    | Lane: 2, 2 (Current Confidence 0.19)
|                        | Empty Lane
|                        | Empty Lane
|                        | Empty Lane
==========================
```

Above is the race with 5 lanes and 2 horses.

Now I will add 2 more horses in lanes 4 and 5.

```
Would you like to Add any horses yes [1], no [0]: 1

Empty Lanes:
Empty Lane: 3
Empty Lane: 4
Empty Lane: 5

Enter the lane number you want to add a horse to: 4

Horse Name: 4
Horse Character: 4
Stop adding horses yes [1], no [0]: 0

Empty Lanes:
Empty Lane: 3
Empty Lane: 5

Enter the lane number you want to add a horse to: 5

Horse Name: 5
Horse Character: 5
Stop adding horses yes [1], no [0]: 1
```

And the result is below.

```
==========================
|    1                   | Lane: 1, 1 (Current Confidence 0.18)
|   2                    | Lane: 2, 2 (Current Confidence 0.14)
|                        | Empty Lane
|   4                    | Lane: 4, 4 (Current Confidence 0.21)
|  5                     | Lane: 5, 5 (Current Confidence 0.21)
==========================
```

This test is also successful since 2 more horses have been added into lanes 4 and 5.

Changing the length of the race.

To test this, I will create a race which is 25m long and then change it to 50m in the next race.

```
Choose the length of race [25m - 100m]: 25
```

```
==========================
|     0                  | Lane: 1, Mystic moon (Current Confidence 0.19)
|        r               | Lane: 2, Flying storm (Current Confidence 0.19)
|     7                  | Lane: 3, Shadow runner (Current Confidence 0.19)
|     z                  | Lane: 4, Diamond storm (Current Confidence 0.19)
|       V                | Lane: 5, Red whisper (Current Confidence 0.19)
|  8                     | Lane: 6, White chaser (Current Confidence 0.19)
|  ?                     | Lane: 7, Majestic storm (Current Confidence 0.19)
|     B                  | Lane: 8, Midnight mane (Current Confidence 0.19)
==========================
```

Now I will change the race length.

```
 Would you like to change the length of the race yes [1], no [0]: 1
 Length of Race [25m - 100m]: 50
```

Result:

```
==================================================
| 0                                              | Lane: 1, Mystic moon (Current Confidence 0.09)
| r                                              | Lane: 2, Flying storm (Current Confidence 0.09)
|  7                                             | Lane: 3, Shadow runner (Current Confidence 0.09)
| z                                              | Lane: 4, Diamond storm (Current Confidence 0.09)
|V                                               | Lane: 5, Red whisper (Current Confidence 0.11)
|8                                               | Lane: 6, White chaser (Current Confidence 0.09)
|  f                                             | Lane: 7, Majestic storm (Current Confidence 0.09)
|B                                               | Lane: 8, Midnight mane (Current Confidence 0.09)
==================================================
```

As we can see this test is also successful as the race length has been changed.

Betting

Here I will test if all aspects of the betting system work.

Bets lost.

```
Would you like to bet on a horse yes [1] no [0]: 1
The balance: 100.0
How much money are you putting on this race: 25
Bet has been placed.
Current Balance: 75.0

If Golden prince wins then the payout will be £57.75
If Royal dream wins then the payout will be £57.75
If Wild queen wins then the payout will be £57.75
If White whisper wins then the payout will be £57.75
If Royal fire wins then the payout will be £57.75
If Lightning storm wins then the payout will be £57.75
If Diamond dancer wins then the payout will be £57.75
If Midnight whisper wins then the payout will be £57.75
Enter the name of the horse you want to place the bet on: Diamond dancer
```

Here we can see I have placed a bet on a horse, and it has not won. This means that I should see my balance decrease at the end of the race.

```
============================
|                         z| Lane: 1, Golden prince (Current Confidence 0.21)
|                        P | Lane: 2, Royal dream (Current Confidence 0.21)
|                      Z   | Lane: 3, Wild queen (Current Confidence 0.21)
|                    F     | Lane: 4, White whisper (Current Confidence 0.21)
|                    b     | Lane: 5, Royal fire (Current Confidence 0.21)
|                    b     | Lane: 6, Lightning storm (Current Confidence 0.21)
|          ?               | Lane: 7, Diamond dancer (Current Confidence 0.21)
|                7         | Lane: 8, Midnight whisper (Current Confidence 0.21)
============================
The race was completed in 12.14 seconds.


Golden prince has won the Race!
The horse you placed the bet on has lost
Current balance: 75.0

Current length of the race: 25
Current number of lanes: 8
Current number of horses: 8
```

And this test is also successful.

Bets won.

To make the winning easier to test I have decided to only use 2 horses.

```
Would you like to bet on a horse yes [1] no [0]: 1
The balance: 100.0
How much money are you putting on this race: 50
Bet has been placed.
Current Balance: 50.0

If Mystic fire wins then the payout will be £111.5
If Majestic hoof wins then the payout will be £111.5
Enter the name of the horse you want to place the bet on: Mystic fire
```

So here I decided to put a 50 bet on Mystic fire and the expected result should be the remaining amount of money left (which is also 50) + the winnings which is 111.50.

So, the expected balance is 161.50.

```
==========================
|                        O| Lane: 1, Mystic fire (Current Confidence 0.13)
|                      B  | Lane: 2, Majestic hoof (Current Confidence 0.13)
==========================
The race was completed in 21.52 seconds.


Mystic fire has won the Race!
Current balance: 161.5
```

And from this we can conclude the test is successful.

Betting amount edge cases.

The bet the user puts on cannot be less than 0 and it cannot be greater than their current balance.

So now I will test this and none of my inputs should be accepted.

```
Would you like to bet on a horse yes [1] no [0]: 1
The balance: 100.0
How much money are you putting on this race: -0.1
Bet cannot be less than £0.
Try again.
How much money are you putting on this race: -1
Bet cannot be less than £0.
Try again.
How much money are you putting on this race: 100.01
Not eneough money to place bet.
Try again.
How much money are you putting on this race: 101
Not eneough money to place bet.
Try again.
How much money are you putting on this race: 102
Not eneough money to place bet.
Try again.
How much money are you putting on this race:
```

And as we can see this is also successful.

Horse symbol must be a singular character.

To test this, I will try creating a horse which has a symbol which is not a single character.

```
Horse Character:
Single character only.
Try Again!
Horse Character: 12
Single character only.
Try Again!
Horse Character: 123
Single character only.
Try Again!
Horse Character: 00
Single character only.
Try Again!
Horse Character: character
Single character only.
Try Again!
Horse Character: -1
Single character only.
Try Again!
```

Since the program prevents this from happening input has been well validated and checked so this also passes the test.

Race timer.

For this test, I will test to make sure that the race timer correctly displays the time it took for the race to finish, although this was most likely shown in the background of other tests, I thought I would show this in its own test.

So, I will create a race of 25m and a race of 100m and let's check the time of each.

25m race:

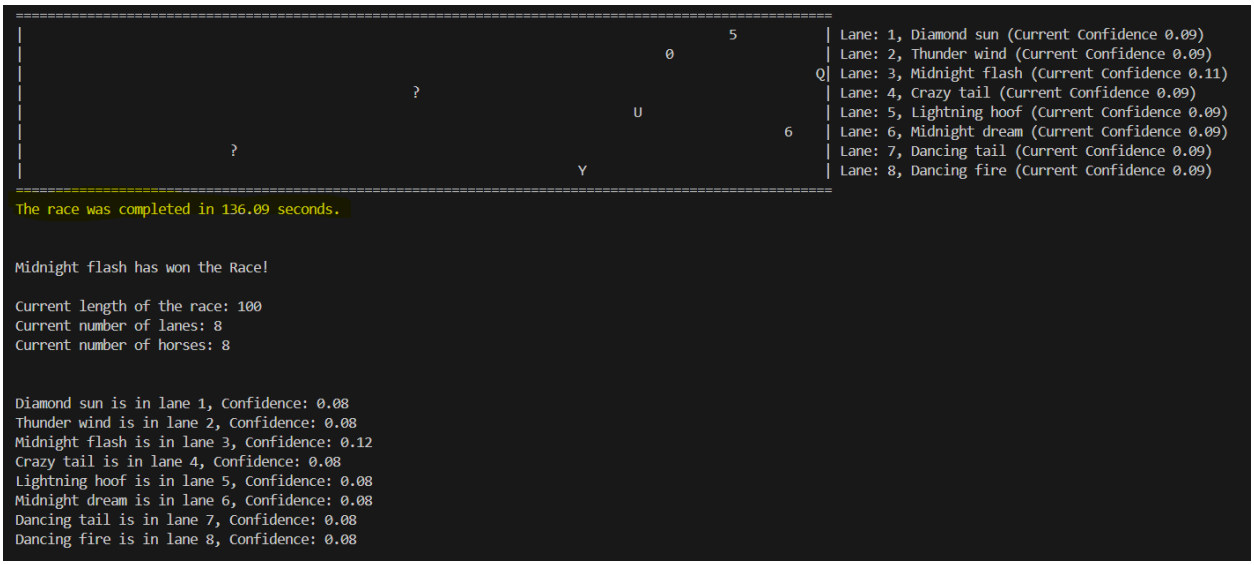
```
Choose the length of race [25m - 100m]: 25
```

```
============================
|                  5       | Lane: 1, Diamond sun (Current Confidence 0.13)
|                  0       | Lane: 2, Thunder wind (Current Confidence 0.13)
|                        Q| Lane: 3, Midnight flash (Current Confidence 0.13)
|                  U       | Lane: 4, Crazy tail (Current Confidence 0.13)
|     ?                    | Lane: 5, Lightning hoof (Current Confidence 0.13)
|                        6 | Lane: 6, Midnight dream (Current Confidence 0.13)
|                  ?       | Lane: 7, Dancing tail (Current Confidence 0.13)
|                       Y  | Lane: 8, Dancing fire (Current Confidence 0.13)
============================
The race was completed in 21.11 seconds.


Midnight flash has won the Race!

Current length of the race: 25
Current number of lanes: 8
Current number of horses: 8


Diamond sun is in lane 1, Confidence: 0.12
Thunder wind is in lane 2, Confidence: 0.12
Midnight flash is in lane 3, Confidence: 0.14
Crazy tail is in lane 4, Confidence: 0.12
Lightning hoof is in lane 5, Confidence: 0.12
Midnight dream is in lane 6, Confidence: 0.12
Dancing tail is in lane 7, Confidence: 0.12
Dancing fire is in lane 8, Confidence: 0.12
```

The 25m race was done in 21 seconds.

Now I will change the race length to 100m, and we can check the time for that.

```
Would you like to change the length of the race yes [1], no [0]: 1
Length of Race [25m - 100m]: 100
```



And now we can see that the 100m race has taken 136 seconds to finish, and this proves that the race timer is working.

So, this test is also successful.

This is the end of the final testing section.

Testing Conclusion:

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.

The program has been tested thoroughly during development by performing unit tests for methods separately in the program. This was done to ensure that the program is robust and correct. In the final testing I have shown the test for key features in the program and made sure that the program is a complete and finished. Each core section was tested individually to ensure that it does what it is supposed to and if I noticed anything which could be better, then I fixed it and tested again. The evidence of testing was done through screenshots in the final testing section, and I explained everything.

Java Conventions

- **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.

Throughout the development of this horse racing simulator, I have made sure to comment all my code, and this is visible in every screenshot of code I have provided in this document. Furthermore, I have made sure to use clear and meaning full and descriptive variable names and method names to make this easier to develop, debug and understand.

Bug Identification and Resolution

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.

There are a few issues which I have found in the original race class, and I have come up with a solution for each.

The first issue which I found was that the user forced to have 3 lanes, this means that the simulation had no customization because it was hard coded in with no user input. The solution I came up was to have all the horses in an Array List, and this meant that the length of the array list determines the number of lanes in the simulation, then I made sure that the user was able to customize this by asking them how many lanes and how many horses they would like.

The second issue I found was that lanes could not be empty, empty lanes were not handled correctly, and every lane was expected to have a horse. This issue was also fixed by adding the horses into an Array List this is because I could have horses in the Array List set to null and if the horse is null then that lane is empty. So, I made changes in the drawing method to draw an empty lane if the horse is null.

Another issue in the first race class was that the horse class was not finished and it was being used in the race class. This means that horses were not being created, horse confidence was not being validated, and horse names were not unique. This is 3 issues all at once due to the lack of the horse class. To be able to create horses I finished the code

for horse class and let the user create horses with their input. Secondly the horse confidence not being validated also occurred because the horse class was not finished, the confidence could be set to anything, so I had to add my own restrictions to make sure it stays between 0 and 1 as needed. The 3rd issue which I mentioned from the lack of the horse class was that the names of the horses were not unique, I fixed this issue by creating a 2nd Array List which stored the names of each horse currently in the race. So, if a horse name came up twice then the user would have to enter a new name.

Once the horse class issues were fixed the next issue was that the race would not finish when all the horses fell, it would be in an endless loop because the check which was done to end the race only checked if a horse has reached the finish line, and in the case that all horse have fell this would make the code end up in an endless loop. So, the way I had fixed this issue is to make the check for finishing the race include checking for at least 1 standing horse.

Another issue I found was that there were no final variables in the code given, this means that everything could have been changed but they were not changed during run-time. This made it difficult to test and maintain, to fix this I added final variables and made sure that once they were declared they do not have to change during run-time. This fix made the code easier to read, debug and maintain.

Improvements to Race Class

- **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.

Displaying winning horse to the user was one of the first features which I developed once I finished the horse class initially, and this was even shown in the testing section of this document. All the bugs and issues which I have mentioned previously have all been fixed and I have also added extra functionality to the race class by adding a betting system and weather which affects the horse confidence as well as a race timer which times the length of the race and a file system which allows the user to save the horse and reuse horses if they choose to do so.

The readability and the maintainability of the code was also enhanced by comments which were short and precise and meaningful.

Conclusion:

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).

Encapsulation in the program has been a particularly important part; I have used private methods almost everywhere and only made methods public if they needed to be used elsewhere in the program and I tried to reduce this as much as possible. All attributes and all variables however in the program are all private. All variables in any class are private and they all need getter and setter methods I did this because I do not want accidental changes in data without me being aware of them happening. This was especially important alongside all the validation I implemented to make sure all User inputs are valid.

Almost all of the testing happened during the development of the program. Catching errors was important. I did not move on from 1 feature to the next until I was certain that the earlier feature was completely implemented and fully functional. The final testing was done by testing the program as a whole and not just the methods separately. This was important to make sure that the program was well put together and any issues would be very noticeable.

This is the end of the Report for Part I.