OOP Development Report

Artur Baran

Feburary 2024

Contents

1 Introduction

The two classes provided needed to be rewritten for the purposes of the assignment. Most of this was done in line to the document provided, however the document didn't clarify some issues explicitly, so some inference was done on my behalf for the functionality of the program.

Rewriting the Horse class was perhaps the fastest part of this project. The goal was simple, use the given templates and using them, add functionality until the Horse class met it's given specifications.

Adding to the Race class took a while longer, since while there was a video that a lot of features could be added from, not all of them were quick changes. Some of them could be done in one or two extra lines, however some of the changes required testing to find extra bugs and reduce redundant code. For example, when making it so that instead of using variables, the **Race** class used an array to store all the horses, some sections of code were merged to make it look cleaner, however the way this was done resulted in more complicated code visually. Now it is less immediately obvious what the code does, but has the same functionality.

Some input validation was done when assigning values to methods in the Horse and Race classes, however it was minimal. Mainly checking if a value was within a range, or if a value was supposed to be set.

1.1 Use of Encapsulation

Encapsulation was used in the **Horse** class in order to prevent unauthorised and accidental changes to the attributes of Horse objects. This was done by making all of the attributes private, and limiting the access of them with Accessors / Getters, and Mutators / Setters. Not all attributes had getters or setters made for them, only when it was necessary.

Getters:

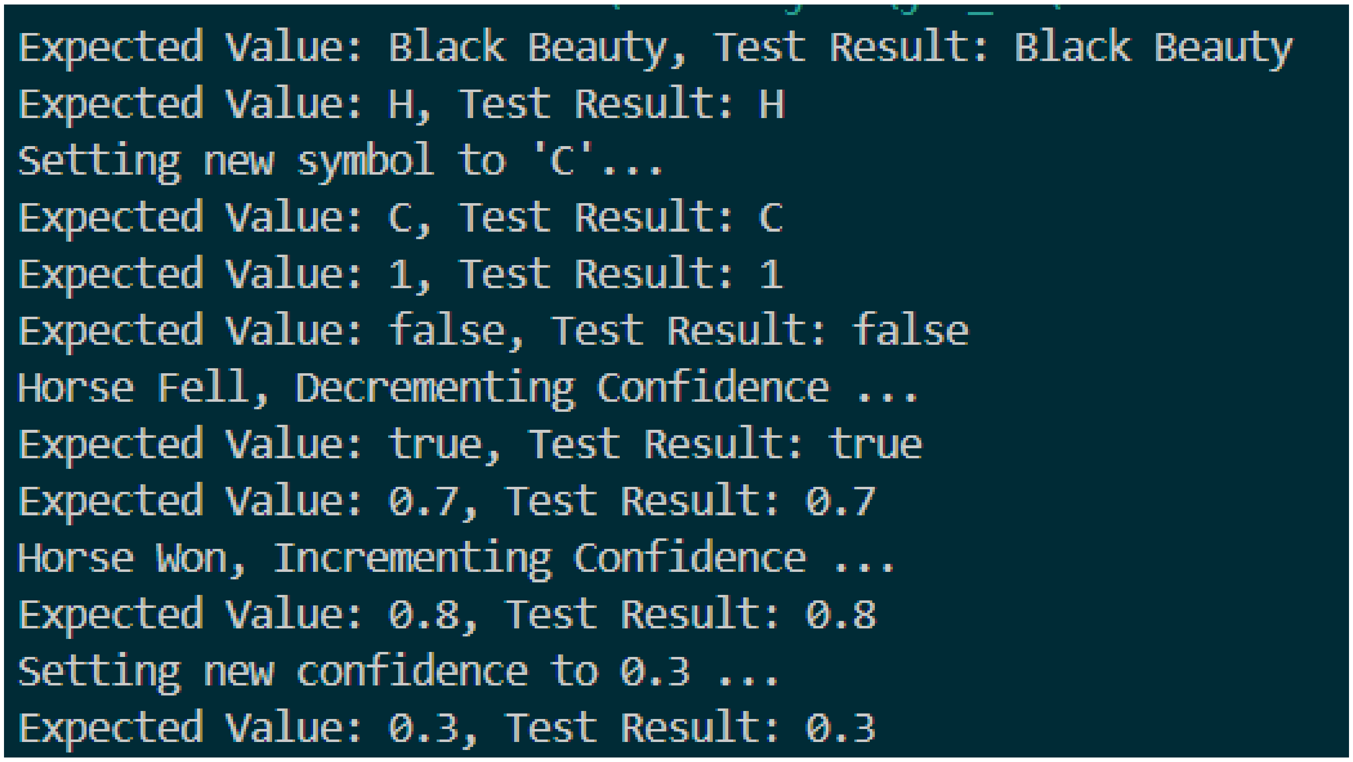
- **getConfidence**
- **getName**
- **getSymbol**
- **hasFallen**

Setters:

- **setConfidence**
- **setSymbol**

1.2 Horse Class Testing

In figure 1 we see test results of running tests on the Horse class.



```
Expected Value: Black Beauty, Test Result: Black Beauty
Expected Value: H, Test Result: H
Setting new symbol to 'C'...
Expected Value: C, Test Result: C
Expected Value: 1, Test Result: 1
Expected Value: false, Test Result: false
Horse Fell, Decrementing Confidence ...
Expected Value: true, Test Result: true
Expected Value: 0.7, Test Result: 0.7
Horse Won, Incrementing Confidence ...
Expected Value: 0.8, Test Result: 0.8
Setting new confidence to 0.3 ...
Expected Value: 0.3, Test Result: 0.3
```

Figure 1: Test Results

When testing I would set initial values for the constructor, the name as 'Black Beauty', the symbol as 'H', and the confidence as 0.8.

Then, I would set values for these attributes, and compare them to the expected values after the operations occur. Below is the testing code for this.

```java
String horseName1 = "Black Beauty";
char horseSymbol1 = 'H', horseSymbol2 = 'C';
double confidence1 = 0.8, confidence2 = 0.3;
Horse horse = new Horse(horseSymbol1, horseName1, confidence1);

System.out.printf("Expected Value: %s, Test Result: %s %n", horseName1, horse.getName());
System.out.printf("Expected Value: %s, Test Result: %s %n", horseSymbol1, horse.getSymbol());
horse.setSymbol(horseSymbol2);
System.out.println("Setting new symbol to 'C'...");
System.out.printf("Expected Value: %s, Test Result: %s %n", horseSymbol2, horse.getSymbol());
```

```java
horse.moveForward();
System.out.printf("Expected Value: %d, Test Result: %d %n", 1, horse.getDistanceTravelled());

System.out.printf("Expected Value: %b, Test Result: %b %n", false, horse.hasFallen());
horse.fall();
System.out.println("Horse Fell, Decrementing Confidence ...");
System.out.printf("Expected Value: %b, Test Result: %b %n", true, horse.hasFallen());
System.out.printf("Expected Value: %.1f, Test Result: %.1f %n", 0.7, horse.getConfidence());

horse.won(100.0);
System.out.println("Horse Won, Incrementing Confidence ...");
System.out.printf("Expected Value: %.1f, Test Result: %.1f %n", 0.8, horse.getConfidence());

System.out.println("Setting new confidence to 0.3 ...");
horse.setConfidence(confidence2);
System.out.printf("Expected Value: %.1f, Test Result: %.1f %n", 0.3, horse.getConfidence());
```

2 Issues with the Race Class:

The Race class was quite lacking. The main things that needed to be changed are listed below:

- Horse Confidence Didn't Change
- Only 3 Horses Allowed.
- Crashing if Lanes are Empty
- Not Printing Horses' Names and Confidence
- Not Displaying a Message if all Horses fall or a Horse wins
- Lack of input validation of race length
- Initial Race conditions not Displayed
- Form Feed Control Symbol Bug
- Implement a check for if there are no horses within the Race, and if there are no horses in a race, end it.
- The Race class would print a '?' when a horse would fall, meanwhile the example video used 'X' to indicate fallen horses.
- Other small changes

On the other hand, the Horse class just needed to be completely finished. It was a bare bones class initially, with none of the methods having any code written for them. There needed to be some changes that had to be inferred, however most of them were based on the instructions provided.

2.1 Horse Confidence Doesn't Change

Initially, when a horse won or fell, it's confidence would remain the same, while these were the moments when it was supposed to change, as outlined in the specification document. To be able to do this, the method **fall** was edited so that when a horse falls, it's confidence also decreases, and a method called **won** was added that increases a horse's confidence when it wins, as well as doing any other actions related to a horse winning a race.

```java
RaceO r = new RaceO(34);
HorseO a = new HorseO('a', "A", 0.8), b = new HorseO('b', "B", 0.5), c = new HorseO('c', "C", 0.5);

r.addHorse(a, 1);
r.addHorse(b, 2);
r.addHorse(c, 3);
```

```
r.startRace();

System.out.printf("Horse: %s has confidence %.1f %n", a.getName(), a.getConfidence());
System.out.printf("Horse: %s has confidence %.1f %n", b.getName(), b.getConfidence());
System.out.printf("Horse: %s has confidence %.1f %n", c.getName(), c.getConfidence());
```

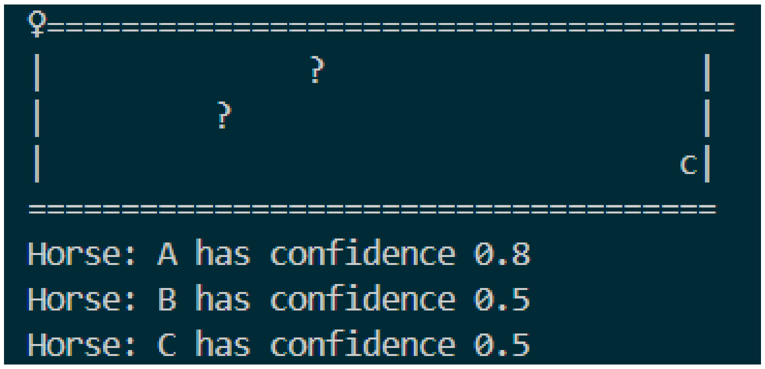
See figure 2 for an example. That was reached with the test code above, and the horse confidence had remained the same as what it was set to. The **Horse** class was changed to include the code below:

```
/**
 * This method simulates a horse falling down, and decrease the horse's
 * confidence as a result.
 */
public void fall() {
    this.setConfidence(this.confidence - 0.1);
    fallen = true;
}

/**
 * This method is called if a horse wins a race, and it increases the amount of
 * capital they have based on the earnings of that race for winning.
 *
 * @param earnings: The amount of money the horse has earned by winning.
 */
public void won() {
    setConfidence(confidence + 0.1);
}
```

And the **Race** class' **raceWonBy** method was changed to the one below:

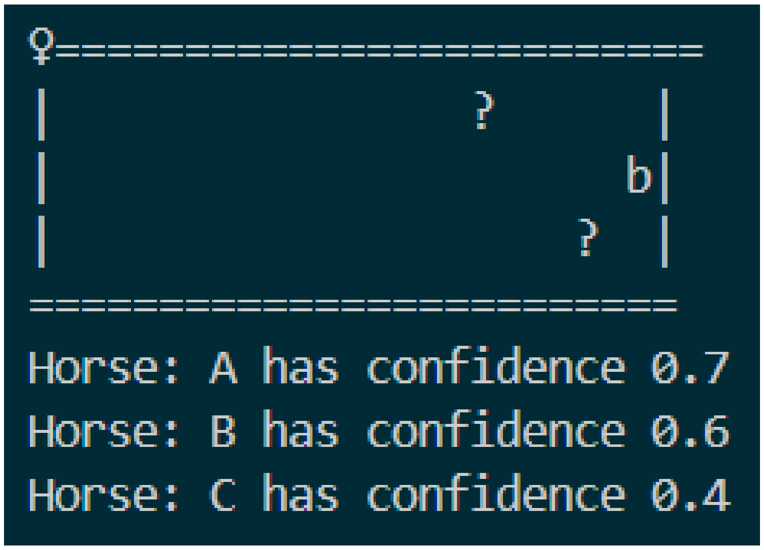
```
/**
 * Determines if a horse has won the race
 *
 * @param theHorse The horse we are testing
 * @return true if the horse has won, false otherwise.
 */
private boolean raceWonBy(Horse theHorse) { // DONE: make winning increase confidence
    if (theHorse.getDistanceTravelled() == raceLength) {
        theHorse.won();
        this.winner = theHorse;
        return true;
    }
    return false;
}
```

And with these changed made, the horses' confidence would change during the race.



```
♀=====================================
|              ?                     |
|         ?                           |
|                                   c|
=====================================
Horse: A has confidence 0.8
Horse: B has confidence 0.5
Horse: C has confidence 0.5
```

Figure 2: Stagnant Confidence at End of Race



```
♀===========================
|                 ?      |
|                       b|
|                     ?  |
===========================
Horse: A has confidence 0.7
Horse: B has confidence 0.6
Horse: C has confidence 0.4
```

Figure 3: The changed horse confidence values

2.2 Only 3 Horses Allowed

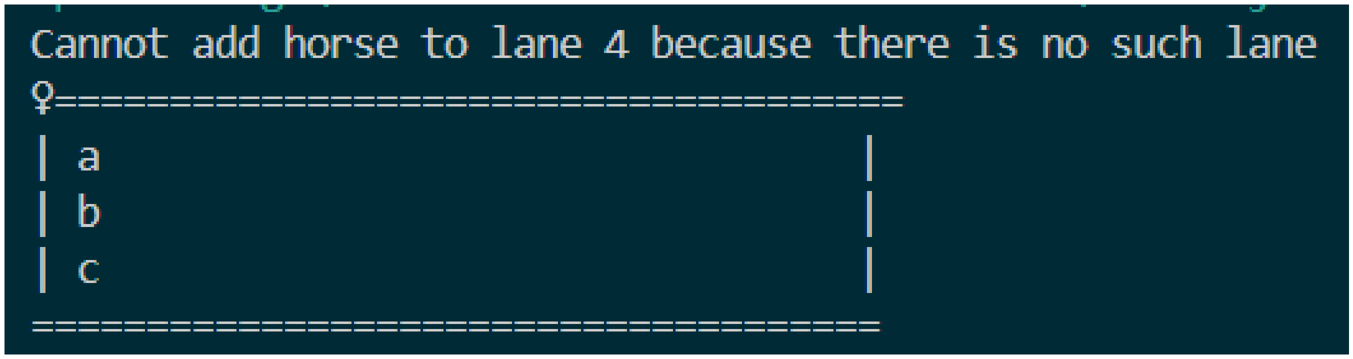
The given Race class was hard coded to only to 3 Horses to be assigned to any Race object. This substantially limited the scalability of the code, and required frequent chains of if statements since each lane would have to have certain logic hard coded into the program.

This wasn't a big issue, however it required longer code with chains of if statements, since it was implemented as 3 separate horse variables instead of an array. It also limited scalability of the project since you would need to add a variable if you wanted more than 3 lanes.

To deal with this issues, a Horse array field called **horses** was added, and this replaced **lane1Horse** through to **lane3Horse**. This required several small rewrites of the program's logic, mainly by looping through the array wherever some processing was done on all of the horses. The changes were made in the methods: **startRace**, **addHorse**, and **printRace**.

```
// Race0 and Horse0 represent the original Horse and Race classes before all of the changes were made to
    them
Race0 r = new Race0(34);
Horse0 a = new Horse0('a', "A", 1), b = new Horse0('b', "B", 1), c = new Horse0('c', "C", 1),
       d = new Horse0('d', "D", 1);

r.addHorse(a, 1);
r.addHorse(b, 2);
r.addHorse(c, 3);
r.addHorse(d, 4);

r.startRace();
```

```
/**
 * Adds a horse to the race in a given lane
 *
 * @param theHorse the horse to be added to the race
 * @param laneNumber the lane that the horse will be added to
 */
public void addHorse(Horse theHorse, int laneNumber) {
    if (laneNumber > this.numberHorses || laneNumber <= 0) {
        System.out.println("Cannot add horse to lane " + laneNumber + " because there is no such lane");
    }

    this.horses[laneNumber - 1] = theHorse;
```



```
Cannot add horse to lane 4 because there is no such lane
♀====================================
| a                                  |
| b                                  |
| c                                  |
=====================================
```

Figure 4: An error when a programmer tries to add more horses than lanes available.

```
}

/**
 * Start the race
 * The horse are brought to the start and
 * then repeatedly moved forward until the
 * race is finished
 */
public void startRace() {
    // declare a local variable to tell us when the race is finished
    boolean finished = false;

    if (this.raceLength < 1) { // prevents a race occuring when the raceLength is invalid
        System.out.println("Race too short or incorrect unit entered. ");
        return;
    }

    if (this.noHorses()) { // prevents a race from starting if there are no horses
        System.out.println("No horses have been added to this race.");
        return;
    }

    // resets all the lanes (all horses not fallen and back to 0).
    for (Horse h : this.horses) {
        if (h != null) {
            h.goBackToStart();
        }
    }

    while (!finished) {
        boolean allFallen = true, won = false;

        for (Horse h : this.horses) {
            if (h != null) {

                // move each horse
                moveHorse(h);

                // checks if the race should continue
                won = won || raceWonBy(h);
                allFallen = allFallen && h.hasFallen();
            }
        }

        finished = won || allFallen;
        // print the race positions
        printRace();

        // wait for 100 milliseconds
        try {
            TimeUnit.MILLISECONDS.sleep(300);
        } catch (Exception e) {
        }
    }

    // output who wins the race
    if (this.winner != null) {
        System.out.printf("And the winner is %s %n", this.winner.getName());
    } else {
        System.out.println("All of the horses fell, the race cannot continue. ");
    }

}

/***
 * Print the race on the terminal
 */
private void printRace() {

    clearTerminalWindow();

    multiplePrint(boundaryChar, raceLength + 3); // top edge of track
    System.out.println();

    for (Horse h : this.hores) {  if (h == null) {

    } else {
        printLane(h);
        }
        System.out.println();
    }

    multiplePrint(boundaryChar, raceLength + 3); // bottom edge of track
    System.out.println();
}

private void clearTerminalWindow() {
    // clear the terminal window
    System.out.print("\033\143");
}
```

These changes made it possible to have an arbitrary number of horses present in a race, and so two constructors can be used. One for a default of race with only 3 lanes, and another with no restriction on lane count.
Overall these changes made most sections of code edited cleaner, and simpler to understand.

```
/**
 * Constructor for objects of class Race
 * Initially there are no horses in the lanes
 *
 * @param raceTrackLength the length of the racetrack in metres
 * @param numLanes      the number of lanes that horses can run on, a upper
 *                      value to the number of horses that can be added
 * @param unit          the unit of the racetrack length, measured in meters,
 *                      however it accepts the value "yards", upon which the
 *                      race length will be the equivalent in meters
 */
public Race(int raceTrackLength, int numLanes, String unit) {
    // initialise instance variables
    this.horses = new Horse[numLanes];
    this.numberHorses = numLanes;

    // calculates race length in meters
    if (unit.equals("yards")) {
        this.raceLength = (int) Math.ceil(raceTrackLength * 0.9144);
    } else if (unit.equals("meters")) {
        this.raceLength = raceTrackLength;
    } else {
        this.raceLength = 0;
    }

}

/**
 * The default constructor for the race class if the number of lanes in a race
 * is not specified. It also assumes the units to be meters.
 *
 * @param raceTrackLength the length of the racetrack in metres
 */
public Race(int raceTrackLength) {
    this(raceTrackLength, 3, "meters");
}

/**
 * A constructor for the race class if the unit is not specified. It assumes the
 * unit is in meters.
 *
 * @param raceTrackLength the length of the racetrack in metres
 * @param numLanes        the number of lanes that horses can run on, a upper
 *                        value to the number of horses that can be added
 */
public Race(int raceTrackLength, int numLanes) {
    this(raceTrackLength, numLanes, "meters");
}
```

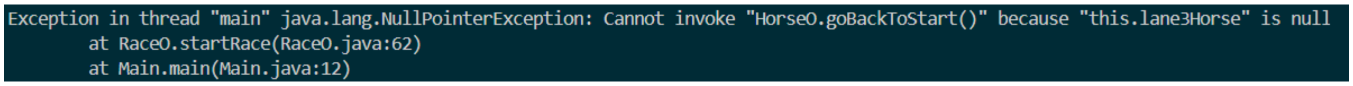
2.3 Crashing when encountering Empty Lanes

The given Race class wasn't able to process empty lanes, and would instead crash if a lane was empty. This issue persisted when lanes were being processed as arrays, however it simply required extra validation to check that the lane being processed wasn't null.

If a Horse wasn't present, a different instruction may occur as necessary. This was done in **startRace**, since it was the only place where it was necessary to check. Methods that required a horse to be present were called only when a horse was present, however it would likely be beneficial to add some input validation to these just in case.

```
Race0 r = new Race0(34);
Horse0 a = new Horse0('a', "A", 1), b = new Horse0('b', "B", 1), c = new Horse0('c', "C", 1),
    d = new Horse0('d', "D", 1);

r.addHorse(a, 1);
r.addHorse(b, 2);

r.startRace();
```

As visible in figure 5, when adding to a lane that doesn't exist, the program would crash, however, as visible in figure 4, when trying to add a Horse to a lane that doesn't exist, the program now simply outputs that adding a Horse was unsuccessful.



```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "Horse0.goBackToStart()" because "this.lane3Horse" is null
	at Race0.startRace(Race0.java:62)
	at Main.main(Main.java:12)
```

Figure 5: A crash when a lane was empty

2.4 Not Printing Horses' Names and Confidence

The Race class initially wouldn't show the names or confidence of horses during a Race. Furthermore, if all Horses fell, it would simply run forever, never terminating.

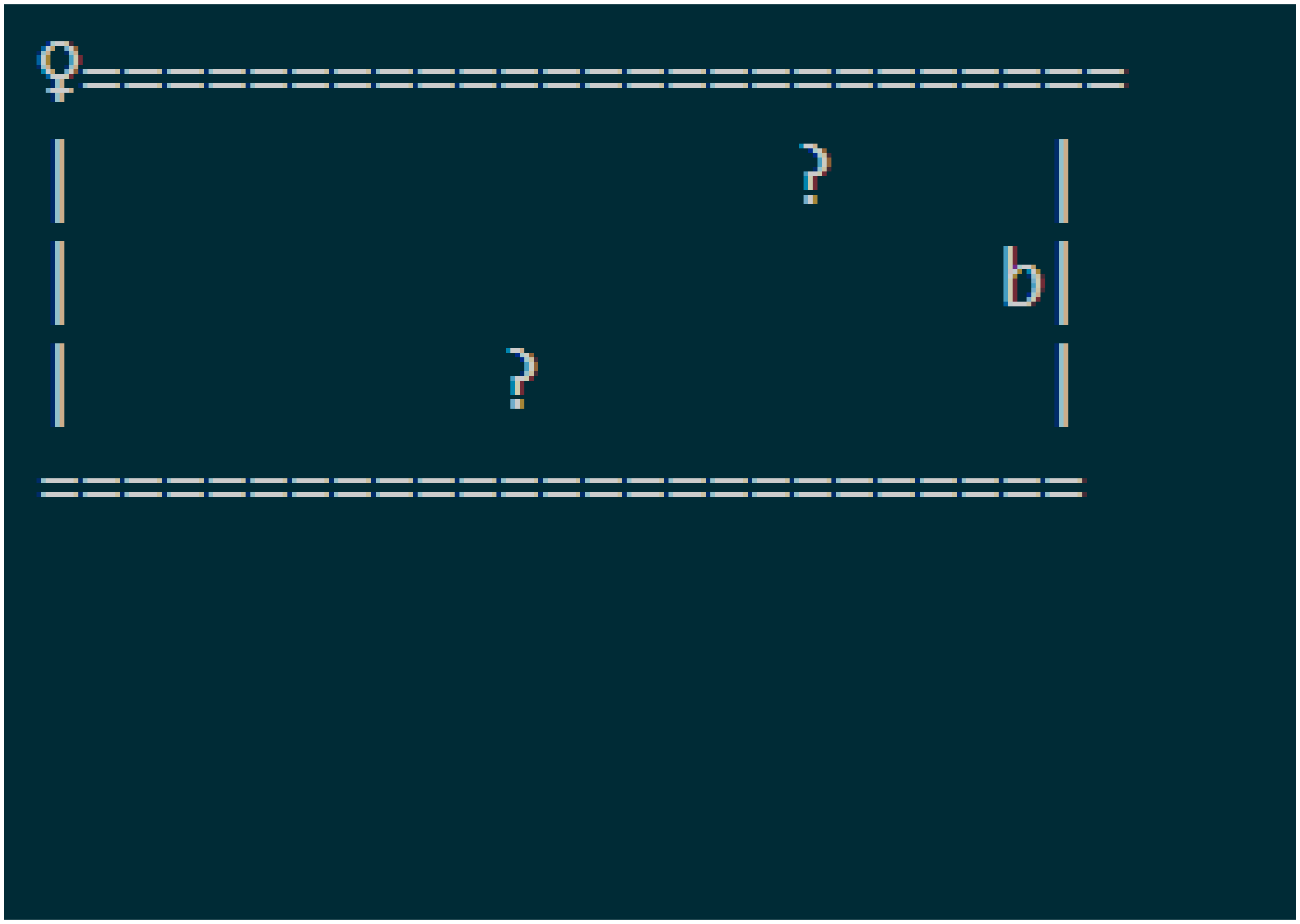
To fix this, a simple change was made to the **printLane** method to display a horses' name and confidence when the lane gets printed.

```java
// Race0 and Horse0 represent the original Horse and Race classes before all of the changes were made to
    them
Race0 r = new Race0(34);
Horse0 a = new Horse0('a', "A", 1), b = new Horse0('b', "B", 1), c = new Horse0('c', "C", 1);

r.addHorse(a, 1);
r.addHorse(b, 2);
r.addHorse(c, 3);

r.startRace();
```

When printing out a lane, Horse name and Confidence were displayed as seen in figure 7. This was a simple fix, just printing another line in **printLane**, as seen below:

```java
private void printLane(Horse theHorse) {

    // calculate how many spaces are needed before
    // and after the horse
    int spacesBefore = theHorse.getDistanceTravelled();
    int spacesAfter = this.raceLength - theHorse.getDistanceTravelled();

    // print a | for the beginning of the lane
    System.out.print(Race.start);

    // print the spaces before the horse
    multiplePrint(Race.emptyLane, spacesBefore);

    // if the horse has fallen then print dead
    // else print the horse's symbol
    if (theHorse.hasFallen()) {
        // System.out.print('\u2322'); // DONE: make print X
        System.out.print('X');
    } else {
        System.out.print(theHorse.getSymbol());
    }

    // print the spaces after the horse
    multiplePrint(Race.emptyLane, spacesAfter);

    // print the | for the end of the track
    System.out.print(Race.stop);

    // print the name and confidence of the horse
    System.out.printf(" %s (Confidence: %.1f)", theHorse.getName(), theHorse.getConfidence());
}
```

The very last line prints the Horse's name and confidence.



```
♀=========================
|                 ?     |
|                      b|
|          ?            |
=========================
```

Figure 6: The program not printing the names of horses

2.5 Not Displaying a Message if all Horses Fall, or a Horse wins

Initially, when the race ended, it wouldn't output why it terminated. Why it terminated being:

1. One of the horses won the race.
2. All of the horses have fallen.

This is also visible in figure 6.

Furthermore, the way the program determines a winner is that it checks which horse has won the race, by calling the method **raceWonBy** on every single horse. This method assigns the horse that wins first to the private attribute **winner**, and returns whether or not the inputted horse has won. This is used by the program to print out which horse won the race.

```java
/**
 * Determines if a horse has won the race
 *
 * @param theHorse The horse we are testing
 * @return true if the horse has won, false otherwise.
 */
private boolean raceWonBy(Horse theHorse) { // DONE: make winning increase confidence
    if (theHorse.getDistanceTravelled() == raceLength) {
        theHorse.won();

        this.winner = theHorse;
        return true;
    }

    return false;
}
```

To output the winning horse, the following if statement was added to the end of the Race class. Note, the else is there to check if no horse won the race, IE if all of the horses have fallen. If this happens, the race ends and it displays that the race has ended since all horses have fallen.

```java
if (this.winner != null) {
    System.out.printf("And the winner is %s %n", this.winner.getName()); // DONE: error if there is no
        winner
} else {
    System.out.println("All of the horses fell, the race cannot continue. ");
}
```

This was done because if all horses fell, the program would run forever, and it wouldn't output any information about a winner, and this would solve both problems.

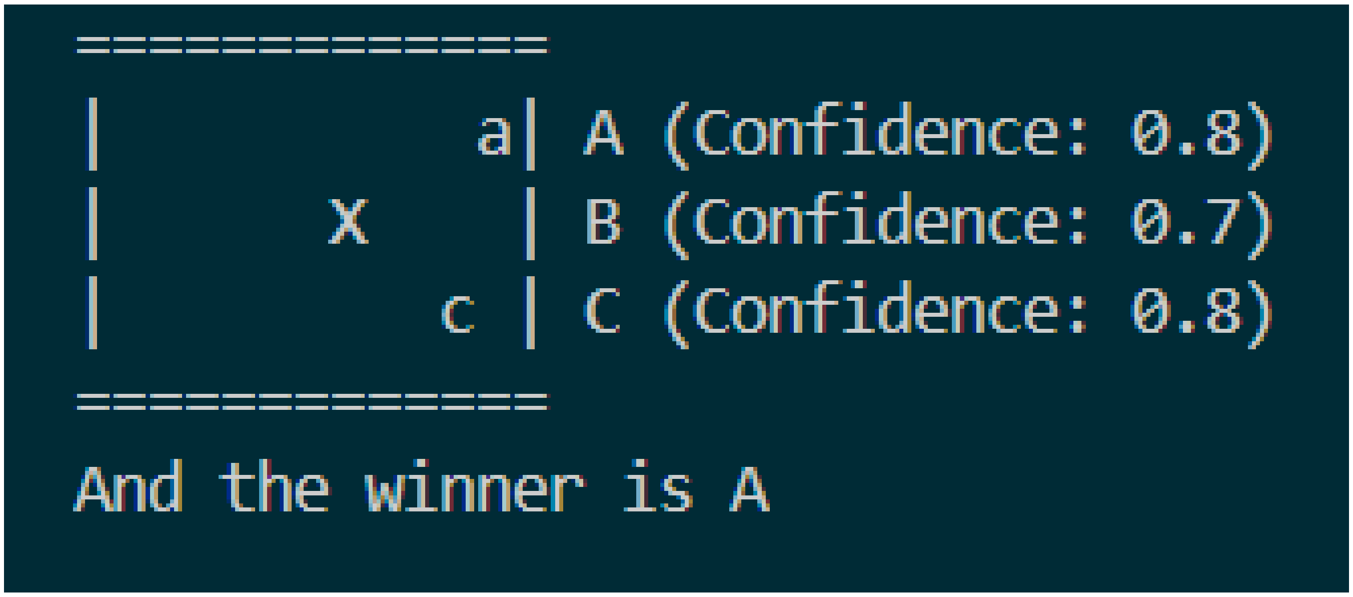
An example of the program displaying the winner of a race is seen in figure 7.

2.6 Lack of Input Validation to Race Length

Initially the Race class didn't have any input validation to check if the length of the Race class was a positive integer, causing it to run indefinitely if set to a negative integer, or 0. This is viewable in figure 8.

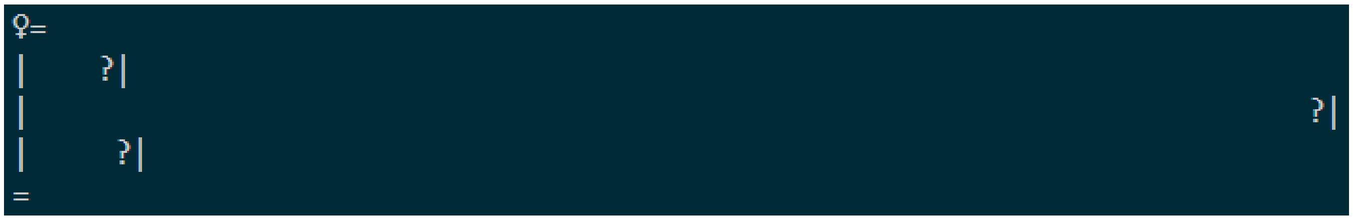
In figure 8, whenether a horse moved forwards, it would move the position of the '|' after it, since there wasn't anything checking when printing lanes if the variable **spacesAfter** was positive or not when attempting to print out a lane.

```java
Race0 r = new Race0(-2);
Horse0 a = new Horse0('a', "A", 0.8), b = new Horse0('b', "B", 0.5), c = new Horse0('c', "C", 0.5);

r.addHorse(a, 1);
r.addHorse(b, 2);
r.addHorse(c, 3);

r.startRace();
```



```
==============
|          a| A (Confidence: 0.8)
|      X    | B (Confidence: 0.7)
|         c | C (Confidence: 0.8)
==============
And the winner is A
```

Figure 7: The program displaying Horse confidence and Winner



```
♀=
|   ?|
|                                                                      ?|
|    ?|
=
```

Figure 8: A race that ran indefinitely

```
Race too short or incorrect unit entered.
```

Figure 9: The program now displays that an invalid distance was entered as the distance for a race.

However, simply checking if this variable was negative wouldn't actually solve the problem, the crux of it being that the horses would move regardless of whether the length of the race was positive.

A way to deal with this issue is to implement a case where if the race's length was negative, the horses would move backwards. This would be an odd choice, but it would solve the problem of races with invalid lengths running indefinitely. However, a simpler solution is simply to not start a race in the first place if the race length is invalid.

Alternatively, a solution that was used within the Horse class was to set this value purely through constructors, so you would'nt have to rewrite the input validation each and every time, limiting the Horse's confidence to be between 0 and 1. This is a potential solution for raceLength, but since race length doesn't change over time there wasn't a need to do this.

The following code was added to the **startRace** method:

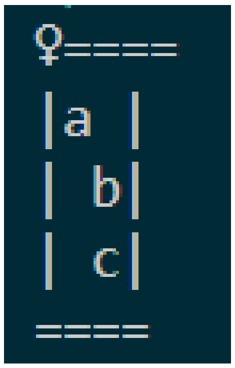
```java
if (this.raceLength < 1) {
    System.out.println("Race too short or incorrect unit entered. ");
    return;
}
```

The above does have the issue of accepting the creation of an invalid race, thus using an exception inside the constructor might work better since it would terminate the program if not caught, instead of wating until the race is started to tell the user that the race length is invalid. Alternatively, having a static wrapper method that creates races and returns them if they are valid would also work, but due to the lack of instructions on what changes to the constructors of the classes, this solution wasn't used.

2.7 Initial Race Conditions not Displayed

Initially, the program would move the horses' before actually printing out the very first state of the horse race. This is an issue so far as it results in sometimes horses just moving and it looks unusual when you are trying to have a race. It doesn't look fair if the very first view you have of a race is one or more horses having moved. This results in interesting bugs where technically, if the race length is 1, a horse can win automatically, with no prior indication that implies the horse has moved. Essentially, to the user, it would look inconsistent if a horse just happens to seemingly start ahead of the other horses. This is viewable in figure 10. Here, you wouldn't know that this was the first stage of the race since it looks like the horses have been moving already. However the race is already over the instant this is shown. To fix this, a single line was code was added to the method **startRace**, the following:

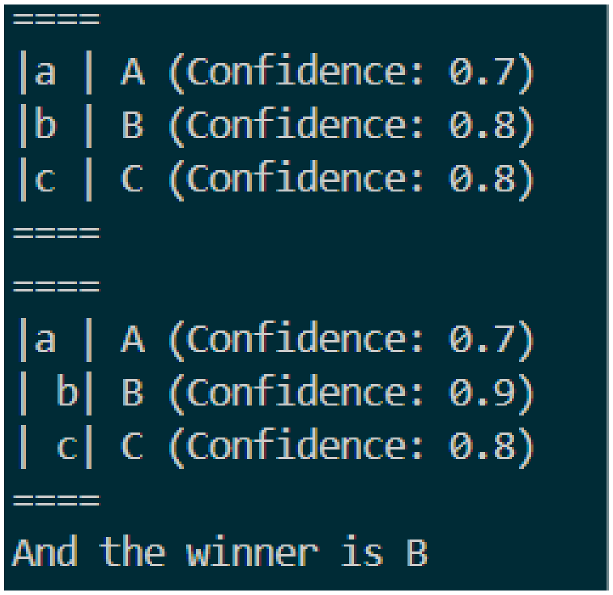
```java
// print the initial state
printRace();
```



```
♀====
|a |
| b|
| c|
====
```

Figure 10: a

And this was added just before the main loop in **startRace** in order to print the initial conditions, just before any horses have moved. An example of the results of this is visible in figure 11



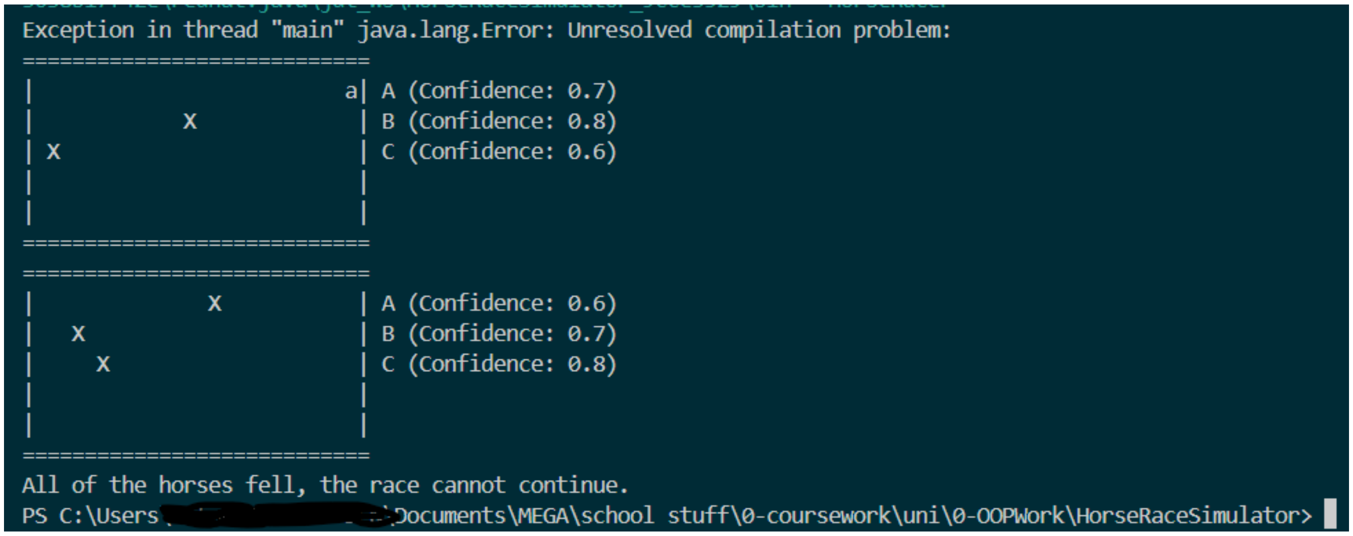
```
====
|a | A (Confidence: 0.7)
|b | B (Confidence: 0.8)
|c | C (Confidence: 0.8)
====
====
|a | A (Confidence: 0.7)
| b| B (Confidence: 0.9)
| c| C (Confidence: 0.8)
====
And the winner is B
```

Figure 11: a

Thus more appropriate information was communicated to the user, decreasing confusion.

2.8 Form Feed Control Symbol Bug

The Form Feed Unicode Control Character, '\u000c', didn't process correctly on my compiler for whatever reason. This resulted in the Race class not functioning as desired in the command line.



```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
============================
|                         a| A (Confidence: 0.7)
|            X             | B (Confidence: 0.8)
| X                        | C (Confidence: 0.6)
|                          |
|                          |
============================
============================
|              X           | A (Confidence: 0.6)
|   X                      | B (Confidence: 0.7)
|     X                    | C (Confidence: 0.8)
|                          |
|                          |
============================
All of the horses fell, the race cannot continue.
PS C:\Users[redacted]Documents\MEGA\school stuff\0-coursework\uni\0-OOPWork\HorseRaceSimulator>
```

Figure 12: The text output has cut into the line displaying the command and location, as well as any previously written text

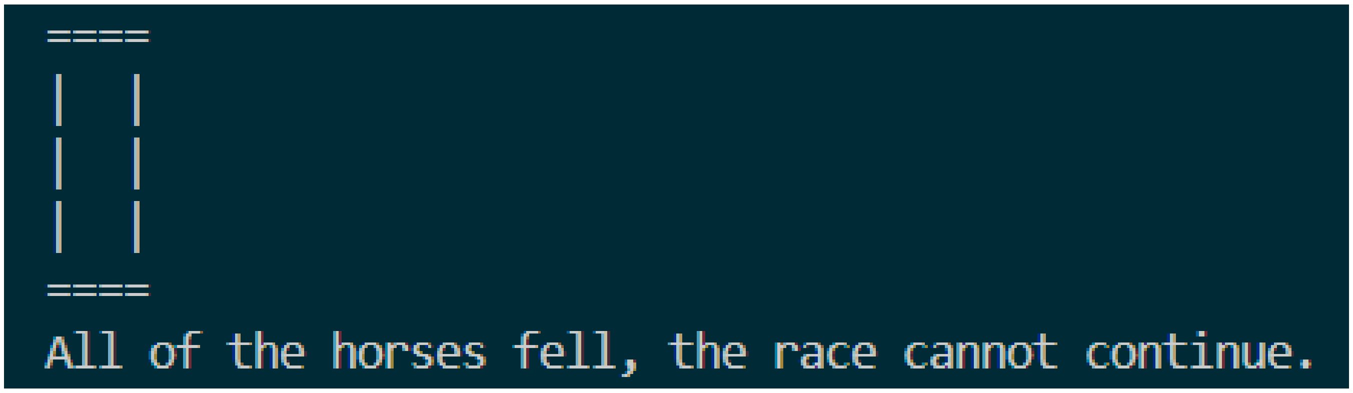
Partial fix: After researching about it, I found an alternative to the form feed sybmol, an escape character, "\033\143", that was able to implement the desired functionality in my IDE, VSCode. However, this didn't actually solve the problem when running it from the command line, instead it just changed the symbol being printed from '♀' to '←c' when running from the terminal.

So, I assume that the provided form feed character worked on the compiler or IDE that the Starter zip file's contents were made in, however doesn't for me for some reason that I didn't find out other than some nuances with terminals. I wasn't able to find out a clear reason why, so this issue has been left unaddressed for the time being.

Other potential solutions would include writing enough new lines to undo this, however that would likely be inconsistent since if the window size is changed, then the problem remains. This creates difficulties with actually creating an animation in the first place.

2.9 Lack of a Check to see if the Race has any Horses

Sometimes, there are scenarios where no horses get added to a race track, when this happens, initially the program would have crashed immediately, but because of checking to see if a lane has a horse, now instead of crashing it results in figure 13.



```
====
|  |
|  |
|  |
====
All of the horses fell, the race cannot continue.
```

Figure 13: A race with no horses

To fix this problem, two changes were implemented. First was adding the following code at the start of the **startRace** method:

```java
if (this.noHorses()) { // prevents a race from starting if there are no horses
    System.out.println("No horses have been added to this race.");
    return;
}
```

And secondly, the following method was added:

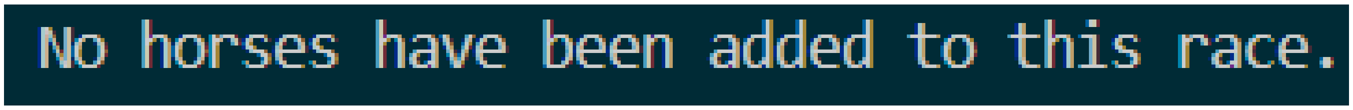
```java
/**
 * Check if there are any horses in horses array.
 *
 * @return bool: false if there is a single horse in the race
 */
private boolean noHorses() {
    for (Horse h : this.horses) {
        if (h != null) {
            return false;
        }
    }
    return true;
}
```

The method in question loops through the horses array attribute, and results in the following being outputted when the if statement above, the output in figure 14

Another way to solve this problem would be to assign the value of a boolean attribute, and return it. This attribute would default to returning true, for no horses, and if a horse is added, it's value would be false. Alternatively, instead of using a boolean, a counter could be added that could be incremented upon each time a horse gets added to a lane.

The following test code was then used to check the functionality:

```java
Race r = new Race(1);

r.startRace();
```

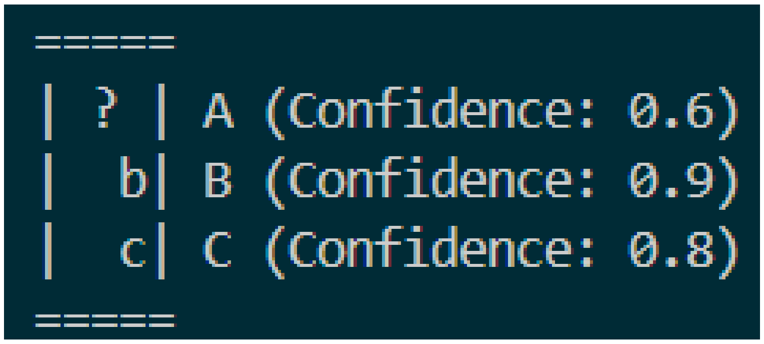


```
No horses have been added to this race.
```

Figure 14: The output screen if a race has no horses

2.10 Fallen Horse ? instead of X

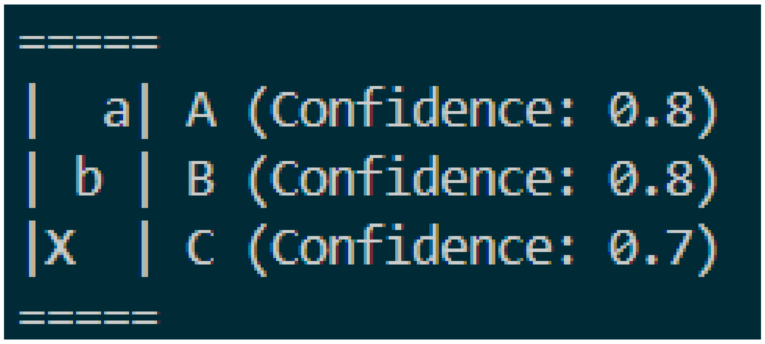
Initially, the program used the unicode character '\u2322' to represent a fallen horse. However, unfortunately this just resulted in printing out '?' whenever a horse fell. This is because by default, the vscode font doesn't support unicode symbols in the terminal for whatever reason. See figure 15.



```
=====
| ? | A (Confidence: 0.6)
|  b| B (Confidence: 0.9)
|  c| C (Confidence: 0.8)
=====
```

Figure 15: A race with horse A having fallen. Displayed by an '?'

The character used to display a fallen horse was changed to what the PDF and example video used, an X, however without any colour.



```
=====
|  a| A (Confidence: 0.8)
| b | B (Confidence: 0.8)
|X  | C (Confidence: 0.7)
=====
```

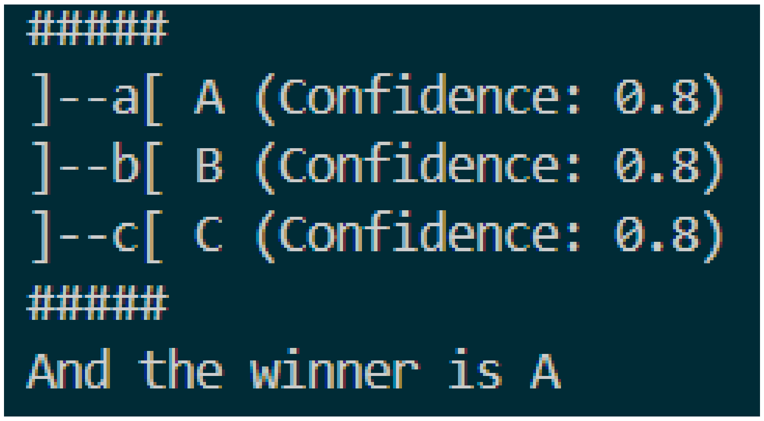
Figure 16: A race with horse C having fallen. Displayed by an 'X'

2.11 Other Edge cases

A variety of small changes were made to the Race and Horse classes throughout the process, including making each element of the tracks customisable, by printing specific characters when printing the boundaries of a track, an empty space within a lane, and the start and end blocks of the track.

This for example, results in figure 17 being a potentially valid race. In this image, the boundary of the race, normally represented by '=', was replaced by '#'; the starting pipes, '|', were replaced by ']', and the ending pipes were replaced by '['.

Furthermore, a small change was made to the main constructor that allowed units to be used to determine the race length. Race length was measured in meters, so the race length would be converted from a given unit to meters before setting the race length. The only 2 units in use currently are meters and yards, and yards being only slightly smaller than meters, any race using yards is about 9% shorter for any given race length. This isn't really noticeable, but is done by the program. The code for this is given below:

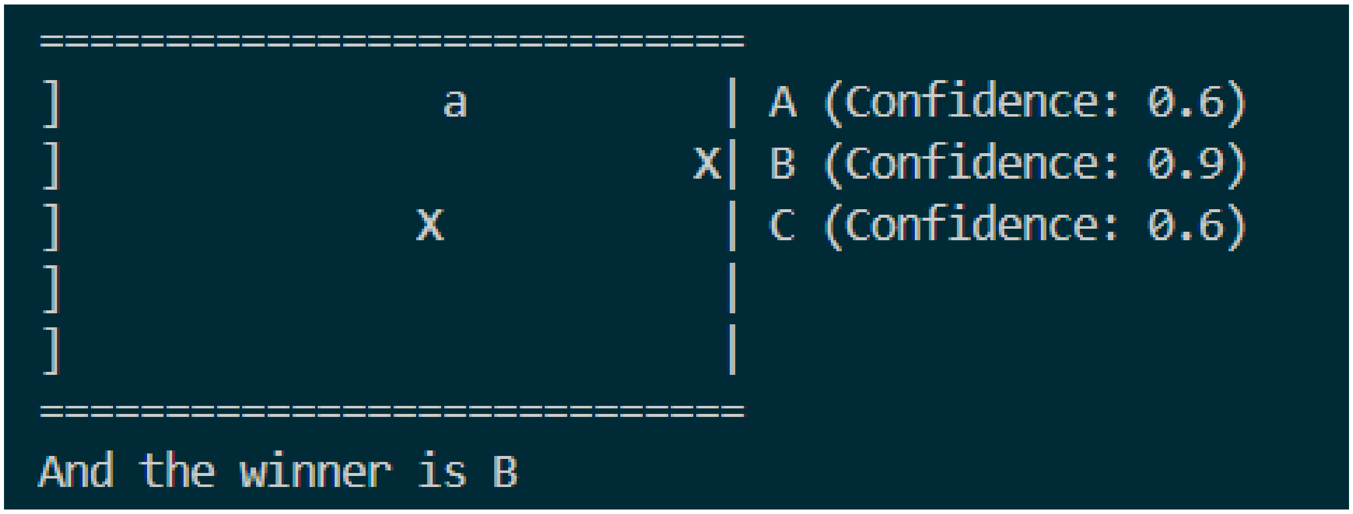


```
######
]--a[ A (Confidence: 0.8)
]--b[ B (Confidence: 0.8)
]--c[ C (Confidence: 0.8)
######
And the winner is A
```

Figure 17: A race with symbols changed for certain characters

```java
// calculates race length in meters
if (unit.equals("yards")) {
    this.raceLength = (int) Math.ceil(raceTrackLength * 0.9144);
} else if (unit.equals("meters")) {
    this.raceLength = raceTrackLength;
} else {
    this.raceLength = 0;
}
```

And this code is ran in the constructor, and sets the value of the **raceLength**.



```
============================
]               a          | A (Confidence: 0.6)
]                         X| B (Confidence: 0.9)
]              X           | C (Confidence: 0.6)
]                          |
]                          |
============================
And the winner is B
```

Figure 18: Winner of the Race has Fallen, hence is an X

There were likely other, small, changes made during the development of this program, however I either do not remember making them or did not take note of them.

3 Potential improvements

The following is a list of potential improvements that could yet be made to the Race class.

- Change the movement engine to be based on horse kinetic power and the weight of the equipment on the horse
- Consider adding a rider to the aspects of the horse that can be customised

- Increase efficiency by storing the to be printed strings as an array, and only updating them when there is a change, instead of remaking the same strings time and time again, even when unnecessary (e.g. if a horse falls, or the top and bottom line of the race)
- Implement a feature that if more than one horse wins, there is a tie and all horses in the tie win.
- Implement a feature where the race continues after there is a single winner and display placement of individual horses in the race.
- Storing the unit of the race, for later reference

4 Final versions of Race and Horse in phase 1

These are the final versions of the classes **Race** and **Horse** after modifications.

Race class:

```java
import java.util.concurrent.TimeUnit;
import java.lang.Math;

/**
 * A three-horse race, each horse running in its own lane
 * for a given distance
 *
 * @author McFarewell, Artur Baran
 * @version 2.0 2023.03.11
 */
public class Race {
    private int raceLength, numberHorses;
    private Horse[] horses;

    private Horse winner;

    private static char boundaryChar = '=', emptyLane = ' ', start = '|', stop = '|';

    /**
     * Constructor for objects of class Race
     * Initially there are no horses in the lanes
     *
     * @param raceTrackLength the length of the racetrack in metres
     * @param numLanes        the number of lanes that horses can run on, a upper
     *                        value to the number of horses that can be added
     * @param unit            the unit of the racetrack length, measured in meters,
     *                        however it accepts the value "yards", upon which the
     *                        race length will be the equivalent in meters
     */
    public Race(int raceTrackLength, int numLanes, String unit) {
        // initialise instance variables
        this.horses = new Horse[numLanes];
        this.numberHorses = numLanes;

        // calculates race length in meters
        if (unit.equals("yards")) {
            this.raceLength = (int) Math.ceil(raceTrackLength * 0.9144);
        } else if (unit.equals("meters")) {
            this.raceLength = raceTrackLength;
        } else {
            this.raceLength = 0;
        }

    }

    // a default constructor for the Race class
```

```java
/**
 * The default constructor for the race class if the number of lanes in a race
 * is not specified. It also assumes the units to be meters.
 *
 * @param raceTrackLength the length of the racetrack in metres
 */
public Race(int raceTrackLength) {
    this(raceTrackLength, 3, "meters");
}

/**
 * A constructor for the race class if the unit is not specified. It assumes the
 * unit is in meters.
 *
 * @param raceTrackLength the length of the racetrack in metres
 * @param numLanes        the number of lanes that horses can run on, a upper
 *                        value to the number of horses that can be added
 */
public Race(int raceTrackLength, int numLanes) {
    this(raceTrackLength, numLanes, "meters");
}

/**
 * Adds a horse to the race in a given lane
 *
 * @param theHorse the horse to be added to the race
 * @param laneNumber the lane that the horse will be added to
 */
public void addHorse(Horse theHorse, int laneNumber) {
    if (laneNumber > this.numberHorses || laneNumber <= 0) {
        System.out.println("Cannot add horse to lane " + laneNumber + " because there is no such lane");
    }

    this.horses[laneNumber - 1] = theHorse;
}

/**
 * Start the race
 * The horse are brought to the start and
 * then repeatedly moved forward until the
 * race is finished
 */
public void startRace() {
    // declare a local variable to tell us when the race is finished
    boolean finished = false;

    if (this.raceLength < 1) { // prevents a race occuring when the raceLength is invalid
        System.out.println("Race too short or incorrect unit entered. ");
        return;
    }

    if (this.noHorses()) { // prevents a race from starting if there are no horses
        System.out.println("No horses have been added to this race.");
        return;
    }

    // resets all the lanes (all horses not fallen and back to 0).
    for (Horse h : this.horses) {
        if (h != null) {
            h.goBackToStart();
        }
    }
    while (!finished) {
        boolean allFallen = true, won = false;

        for (Horse h : this.horses) {
            if (h != null) {

                // move each horse
                moveHorse(h);

                // checks if the race should continue
                won = won || raceWonBy(h);
                allFallen = allFallen && h.hasFallen();
            }
        }

        finished = won || allFallen;

        // print the race positions
        printRace();

        // wait for 100 milliseconds
        try {
            TimeUnit.MILLISECONDS.sleep(300);
        } catch (Exception e) {
        }
    }

    // output who wins the race
    if (this.winner != null) {
        System.out.printf("And the winner is %s %n", this.winner.getName());
    } else {
        System.out.println("All of the horses fell, the race cannot continue. ");
    }

}

/**
 * Randomly make a horse move forward or fall depending
 * on its confidence rating
 * A fallen horse cannot move
 *
 * @param theHorse the horse to be moved
 */
private void moveHorse(Horse theHorse) {
    // if the horse has fallen it cannot move,
    // so only run if it has not fallen

    if (!theHorse.hasFallen()) {
        // the probability that the horse will move forward depends on the confidence;
        if (theHorse.canMove()) {
            theHorse.moveForward();
        }

        // the probability that the horse will fall is very small (max is 0.1)
        // but will also will depends exponentially on confidence
        // so if you double the confidence, the probability that it will fall is *2
        if (theHorse.canFall()) {
            theHorse.fall();
        }
    }
}
```

```java
/**
 * Determines if a horse has won the race
 *
 * @param theHorse The horse we are testing
 * @return true if the horse has won, false otherwise.
 */
private boolean raceWonBy(Horse theHorse) {
    if (theHorse.getDistanceTravelled() == raceLength) {
        theHorse.won();

        this.winner = theHorse;
        return true;
    }

    return false;
}

/***
 * Print the race on the terminal
 */
private void printRace() {

    clearTerminalWindow();

    multiplePrint(boundaryChar, raceLength + 3); // top edge of track
    System.out.println();

    for (Horse h : this.hor
es) {     if (h == null) {

    } else {
        printLane(h);
        }
        System.out.println();
    }

    multiplePrint(boundaryChar, raceLength + 3); // bottom edge of track
    System.out.println();
}

private void clearTerminalWindow() {
    // clear the terminal window
    System.out.print("\033\143");
}

private void printLane(Horse theHorse) {

    // calculate how many spaces are needed before
    // and after the horse
    int spacesBefore = theHorse.getDistanceTravelled();
    int spacesAfter = this.raceLength - theHorse.getDistanceTravelled();

    // print a | for the beginning of the lane
    System.out.print(Race.start);

    // print the spaces before the horse
    multiplePrint(Race.emptyLane, spacesBefore);

    // if the horse has fallen then print dead
    // else print the horse's symbol
    if (theHorse.hasFallen()) {
            // System.out.print('\u2322');
            System.out.print('X');
        } else {
            System.out.print(theHorse.getSymbol());
        }

        // print the spaces after the horse
        multiplePrint(Race.emptyLane, spacesAfter);

        // print the | for the end of the track
        System.out.print(Race.stop);

        // prints the name and confidence of the horse
        System.out.printf(" %s (Confidence: %.1f)", theHorse.getName(), theHorse.getConfidence());
    }

    private void printEmptyLane() {
        int spaces = this.raceLength + 1;

        // print a | for the beginning of the lane
        System.out.print(start);

        // print the empty lane
        multiplePrint(emptyLane, spaces);

        // print the | for the end of the track
        System.out.print(stop);

    }

    /***
     * print a character a given number of times.
     * e.g. printmany('x',5) will print: xxxxx
     *
     * @param aChar the character to Print
     */
    private void multiplePrint(char aChar, int times) {
        for (int i = 0; i < times; i++) {
            System.out.print(aChar);
        }
    }

    /**
     * Check if there are any horses in horses array.
     *
     * @return bool: false if there is a single horse in the race
     */
    private boolean noHorses() {
        for (Horse h : this.horses) {
            if (h != null) {
                return false;
            }
        }

        return true;
    }

}
```

Horse class:

```java
import java.util.LinkedList;
import java.util.HashMap;
import java.util.Map;
import java.util.Iterator;

/**
 * This class is used to represent horses, and uses a vari
 *
 * @author Artur Baran
 * @version v1 2023.03.10
 */

public class Horse {
    // Fields of class Horse
    private String name;
    private char symbol;
    private int distanceTravelled = 0;
    private boolean fallen = false;
    private double confidence;

    /**
     * Constructor for objects of class Horse
     */
    public Horse(char horseSymbol, String horseName, double horseConfidence) {
        this.symbol = horseSymbol;
        this.name = horseName;
        this.setConfidence(horseConfidence);
    }

    /**
     * To string method for the Horse class, there to have an easy way to output the
     * horse's name.
     */
    public String toString() {
        return String.format("%s", this.name); // name, confidence, breed, total distance travelled?,
             equipment list?,
                                              // symbol?
    }

    // Other methods of class Horse

    /**
     * This method simulates a horse falling down, and decrease the horse's
     * confidence as a result.
     */
    public void fall() {
        this.setConfidence(this.confidence - 0.1);
        fallen = true;
    }

    /**
     * This method resets the distance travelled for the horse, and whether it's
     * fallen yet or not.
     */
    public void goBackToStart() {
        this.distanceTravelled = 0;
        this.fallen = false;
    }

    /**
     * Moves the horse forward by one unit, and increases the distance travelled by
     * 0.1.
     */
```

```
public void moveForward() {
    this.distanceTravelled++;
}

/**
 * This method checks if the horse has moved forward based on a random number.
 *
 * @return boolean: Whether the horse has moved forward.
 */
public boolean canMove() {
    return Math.random() < (this.confidence);
}

/**
 * This method checks if the horse can fall based on a random number.
 *
 * @return boolean: Whether the horse has fallen.
 */
public boolean canFall() {
    return Math.random() < (0.1 * this.confidence * this.confidence);
}

/**
 * This method is called if a horse wins a race. It increases the horse's
 * confidence if they win.
 */

public void won() {
    setConfidence(confidence + 0.1);
}

// Getters / Accessors

public double getConfidence() {
    return this.confidence;
}

public int getDistanceTravelled() {
    return this.distanceTravelled;
}

public String getName() {
    return this.name;
}

public char getSymbol() {
    return this.symbol;
}

public boolean hasFallen() {
    return this.fallen;
}

// Setters / Mutators

public void setConfidence(double newConfidence) {
    if (newConfidence > 0.0 && newConfidence < 1.0) {
        this.confidence = newConfidence;
        return;
    }
    System.out.println("ERROR: Incompatible Confidence");
}
```

```
    public void setSymbol(char newSymbol) {
        this.symbol = newSymbol;
    }

}
```