**Horse Race Simulator Report**

This report’s purpose is to document improvements made to the **Horse** and **Race** class of the simulator. It documents identified issues alongside a thorough explanation, providing possible solutions and testing evidence. Furthermore, it also explains key programming concepts used.

**Horse Class:**

The horse class makes use of encapsulation, a fundamental concept in Object Oriented Programming. This involved bundling related data alongside methods that utilise that data into a single unit (i.e., Class). This is done by setting the access modifiers for the class fields to **private**, which prevents access to the fields outside the Class, only allowing them to be manipulated via accessor (**getter**) and mutator (**setter**) methods.

The following are the mutator methods:

* getConfidence(): Returns the confidence level of the horse
* getDistanceTravelled(): Returns the total distance travelled by the horse
* getName(): Returns the horse’s name
* getSymbol(): Returns the character used to represent the horse
* hasFallen(): Returns a Boolean value whether the horse has fallen

The following are the accessor methods:

* fall(): Sets the horse as fallen
* moveForward(): Increments the distance travelled variable by one
* setConfidence(): Sets the confidence rating of the horse to the given value
* setSymbol(): Sets the horse’s symbol to the character given

The following is a private method I have made:

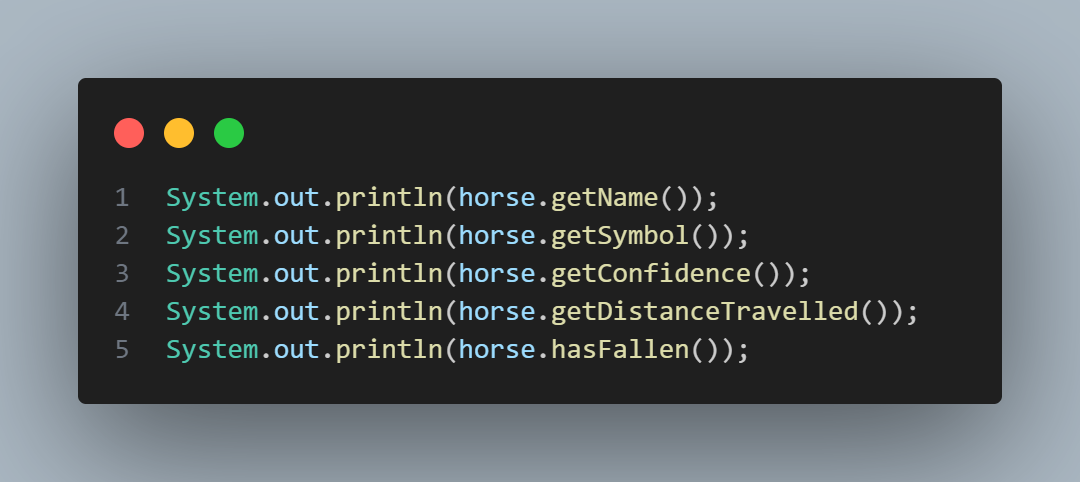
* validateConfidence():
  + This method takes one param: “double confidence”. Uses an if statement to check if the param exceeds the bounds. If the confidence is >= 1.0, it returns 0.99, else if the confidence is <= 0.0, it returns 0.01, else it returns the param as it was.
  + This method has been used within the constructor and setConfidence() to make sure the confidence value does not exceeds the bounds (0 and 1).

**Testing:**

I have provided tests for each method within the horse class to verify its correctness. Screenshots and explanations have also been provided. An external file Main.java has been used for testing. All tests will be conducted on the same instance of the Horse class:

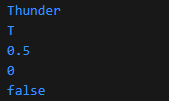
****

**Getter Method Testing:**

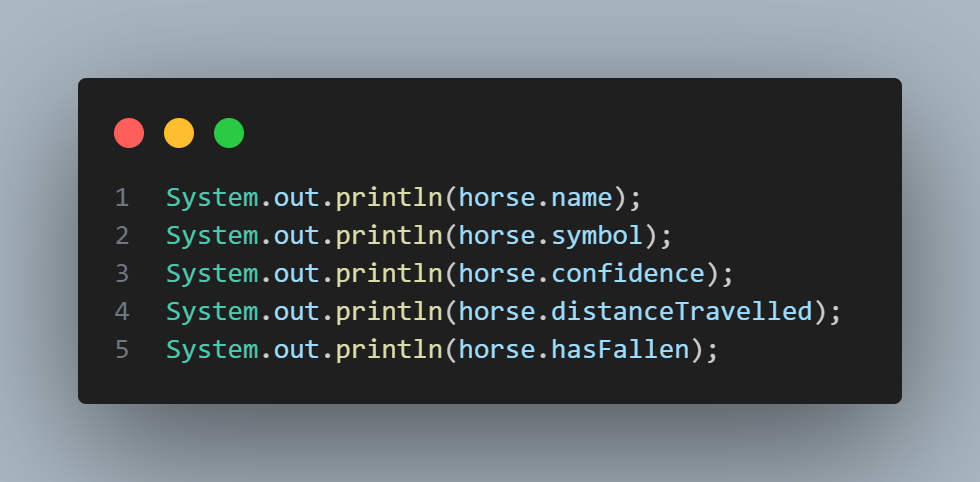


The above calls all the getter methods for each field of the Horse class.

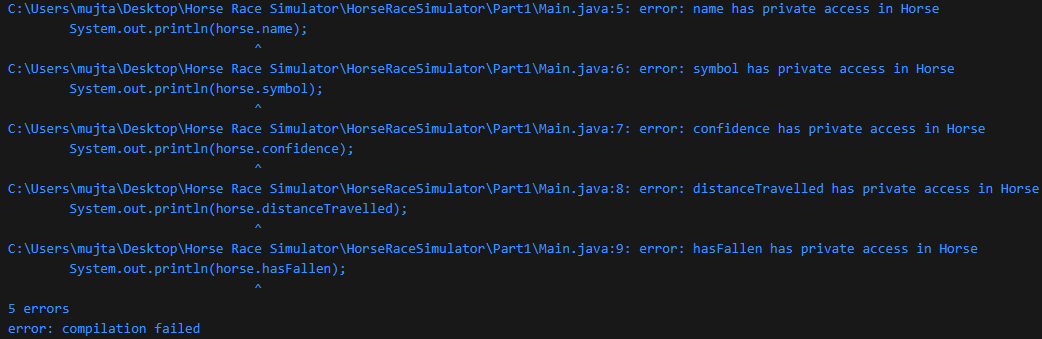
Expected output: Actual output:

* Thunder
* T
* 0.5
* 0
* False

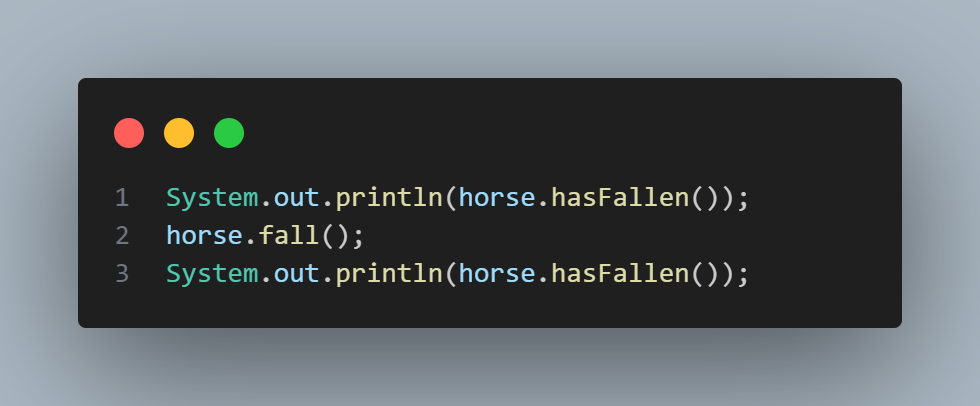
**Access Modifier Testing:**

****

The above code attempts to retrieve the Class fields using dot notation. As the access modifier has been set to **private** the code should not compile and throw an error.



**hasFallen() testing:**

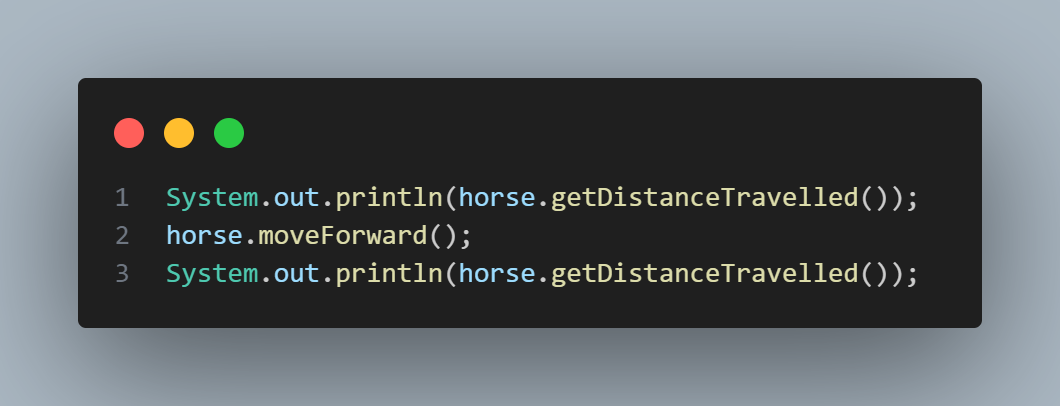


The above code first prints out the **hasFallen** field of the Horse Class. It then proceeds to call **fall()** which sets the field to true. The **hasFallen** field is then printed once more to show the change.

Expected output: Actual output:

* false
* true

**moveForward()** **testing:**

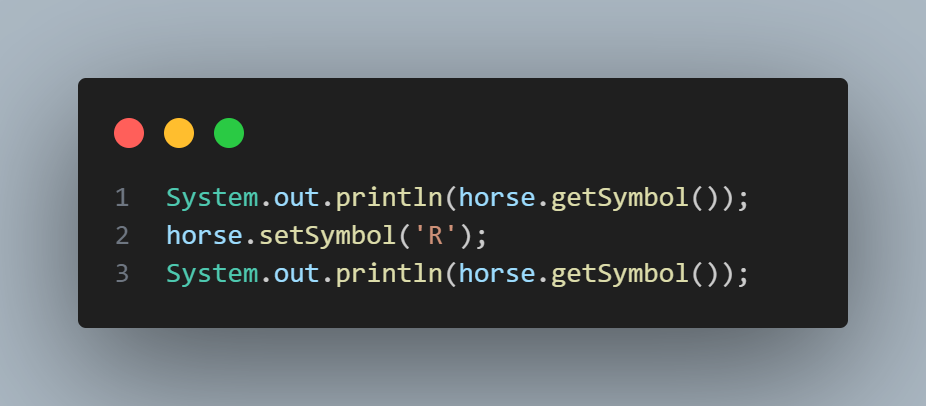


The above code prints out the **distanceTravelled** field of the Horse Class. It then proceeds to call **moveForward()** which increments the **distanceTravelled** by 1. The **distanceTravelled** field is then printed once more to show the change.

Expected output: Actual output:

* 0
* 1

**setSymbol() testing:**



The above code prints out the horse’s **symbol**. It then proceeds to call **setSymbol()** which takes a char parameter, and sets that param as the new symbol. The horse’s **symbol** is then printed once more to show the change.

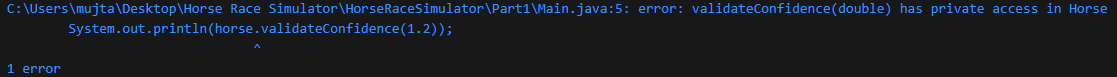
Expected output: Actual output:

* T
* R

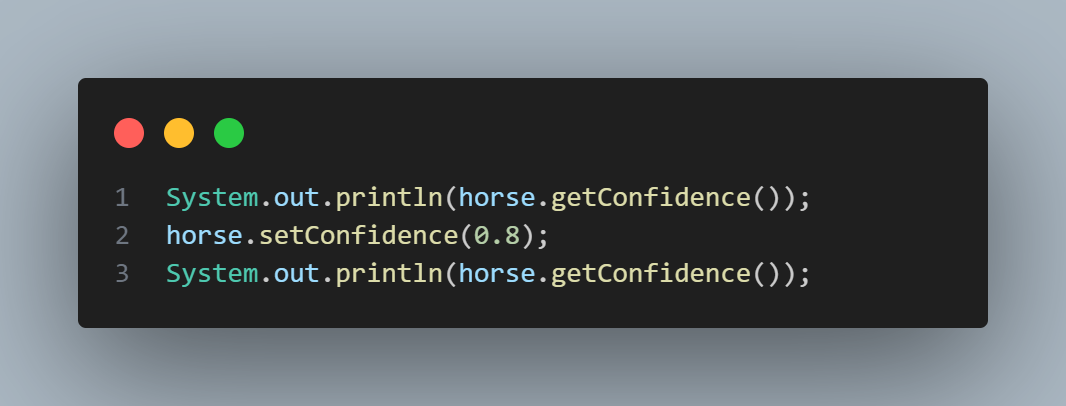
**Attempt to access validateConfidence() outside the class:**



The above code attempts to call the validateConfidece() method of the Horse Class. The access modifier has been set to **private** so the code should not compile and throw an error.



**setConfidence() testing:**

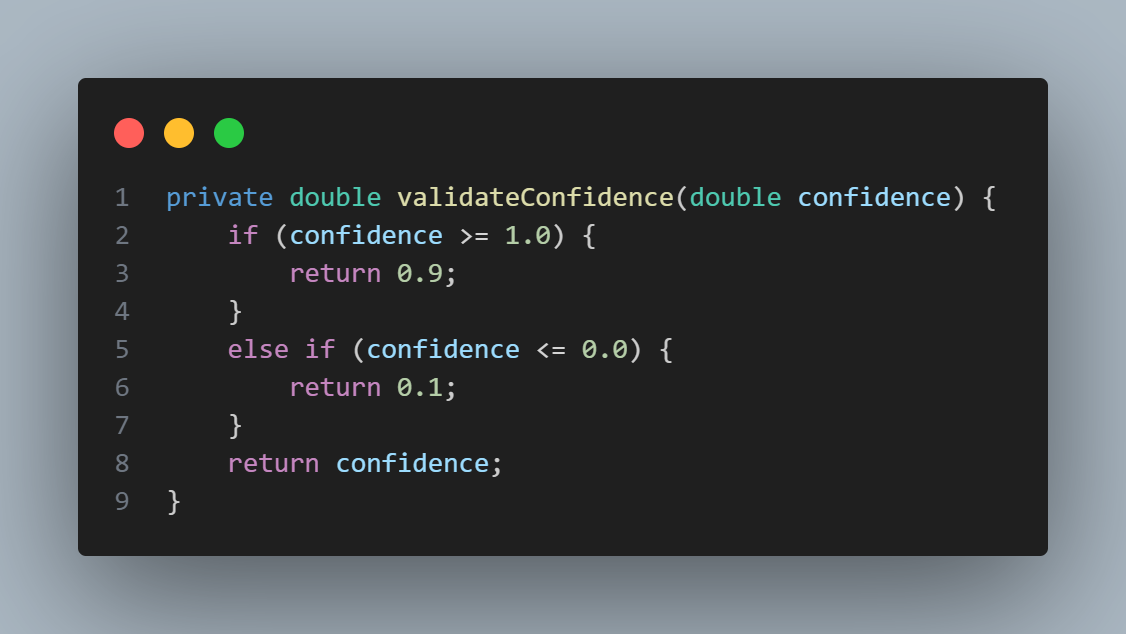


The above code prints out the horse’s **confidence**. It then proceeds to call **setConfidence()** which takes a double parameter, and sets that param as the new confidence. The horse’s **confidence** is then printed once more to show the change.

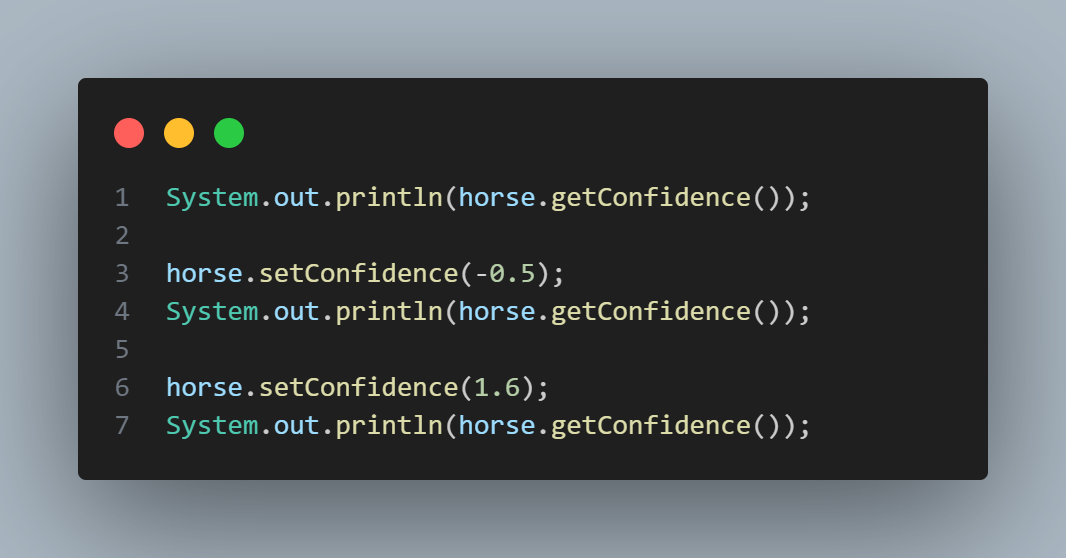
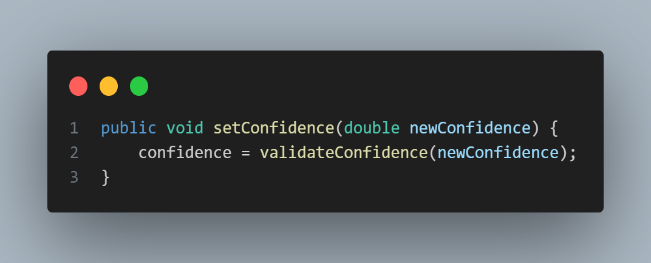
Expected output: Actual output:

* 0.5
* 0.8

**validateConfidence() testing**

****

The above is the **validateConfidence()** method.

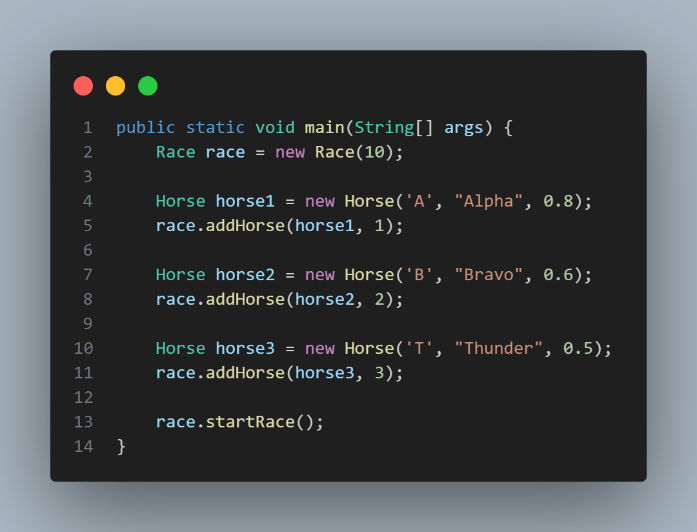


The above code first prints the horse’s **confidence**. It is then set to -0.5, and the **confidence** is printed out again. The **confidence** is then set to 1.6, and is printed once more. As -0.5 is below the bounds, the confidence should be set to 0.01, and as 1.6 is greater than the bounds, the confidence should be set to 0.99.

The code to the right is how the **validateConfidence()** is called within **setConfidence().**

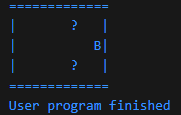
Expected output: Actual output:

* 0.5
* 0.01
* 0.99

**Race Class:**

The race class implements the race logic. The following covers all issues found with their solutions. I have also made enhancements to the code logic, making it more efficient. My testing shall be done within Main.java (which is the file that will run the program) and will look like the following, unless specified within a test case:

**1. Winner of the race is not displayed**

****Once a horse has finished the race, the program shuts without printing the winner as seen below:

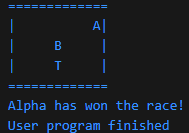
The “?” characters are another issue that will be dealt with later

As seen, the program exits without printing the winner message

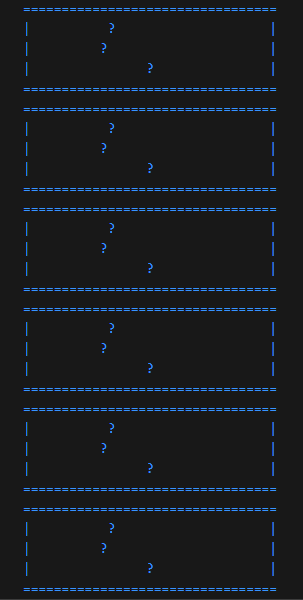
I have added a print statement:

System.out.println(theHorse.getName() + " has won the race!");

within the if block of **raceWonBy()** which checks if a horse has reached the finish line. If they have, a winner message is printed.



**2. Program continuously loops when all horses have fallen**

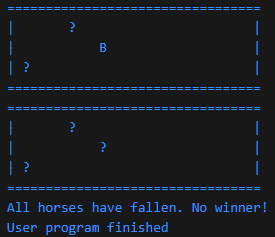
When all horses have fallen, the program repeatedly prints the race track:

As seen on the left, all horses have fallen (an error causes fallen horses being displayed as “?”). Despite this, the program still runs, continuously printing the race track. The user would have to force shut the program. Below I have provided my solution:

I have created a new method **allHorsesFallen()** which checks if all horses have fallen, using their accessor method **hasFallen()**. “&&” makes it set **allFallen** false if there is at least a single false.

boolean allFallen = lane1Horse.hasFallen() && lane2Horse.hasFallen() && lane3Horse.hasFallen();

If **allFallen** is true (which is checked within an if statement), an appropriate message is displayed. The method has been called within **startRace()** in the if statements condition. If it is true, the Boolean **finished** is set to true and the program finishes.

Correct Output:

**3. Unicode characters being displayed as “?”**

As seen in the above tests, when a horse has fallen, a “?” is displayed. The program attempts to print ‘\u2322’, but for some reason it is unable to display the Unicode character. ‘\u2322’ represents ‘⌢’. I do not deem this appropriate so I have replaced it with ‘\u2620’ which represents ‘💀’.

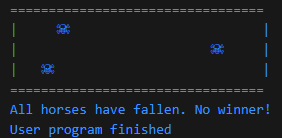
PrintStream printStream = new PrintStream(System.out, true, StandardCharsets.UTF\_8);

System.setOut(printStream);

This code has been placed within Main.java. I have initialised a new **PrintStream** object which takes 3 params:

* **System.out:** represents the standard output stream
* **True:** this enables auto-flushing (output is automatically written to the stream when a newline is encountered)
* **StandardCharsets.UTF\_8:** returns the charset UTF-8

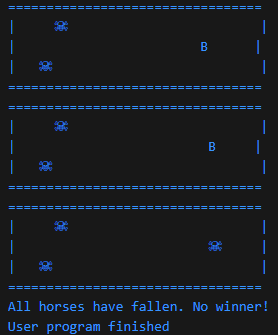
**System.setOut(printStream)** replaces the standard output stream (**System.out**) with my **printStream**. This ensures correct character encoding when printing out to the console.

As seen this works. One thing I am confused about, is that the code snippet sets the output stream to use “UTF-8”, which makes it work. However, my default output stream for VS Code is “UTF-8” yet it does not work unless done like this.

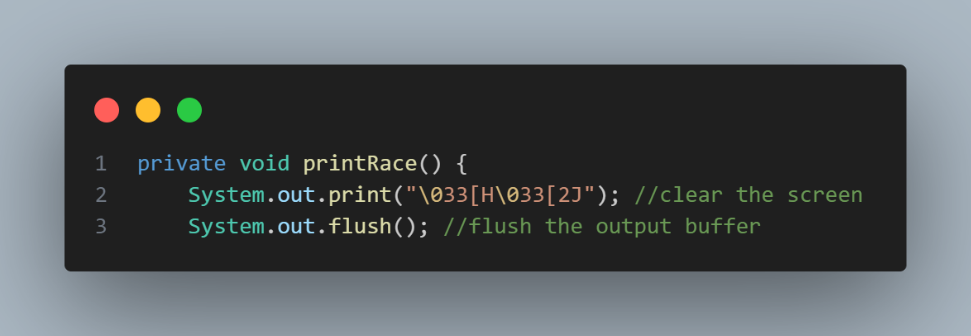
Another thing to keep note of, Window users will have to use the command: **chcp 65001** within Command Prompt to set the character encoding of the terminal to UTF-8.

**4. Console not clearing after each track print**

After each print of the track, the console should clear itself. Unfortunately, it does not, cluttering the console.





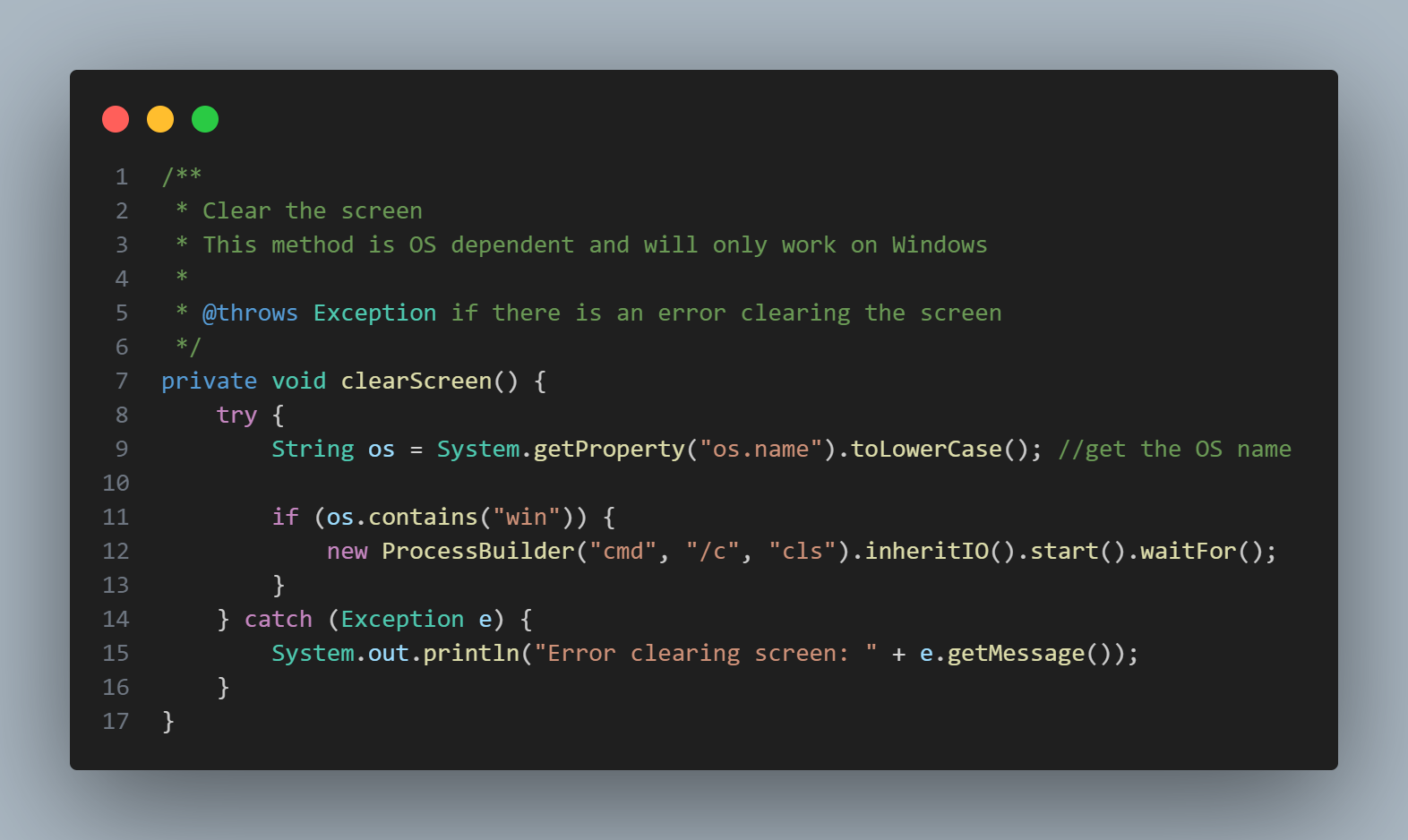
The current program attempts to print ‘\u000C’, which is a Unicode escape sequence to “advance to the next page”. Historically, this was used in printers, and does not have any effect in most modern consoles.

The original print has been replaced with my 2 lines as seen above. Line 2 makes use of ANSI escape codes to clear the console screen:

* **\003**: Escape character (**ESC**)
* **[H**: Moves the cursor to the top-left corner
* **[2J**: Clears the entire screen

When combined, it clears the screen and resets the cursor position. **System.out.flush()** forces Java to immediately write out everything in the output buffer to the console.

However, this ANSI escape code does not work properly on Windows Command Prompt. To fix this I added another method as seen below:



First the operating system is retrieved using **System.getProperty(“os.name”)** and is converted to lowercase. The if statement checks if the name contains “win” (**Windows OS**).

Line 12: **new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();**

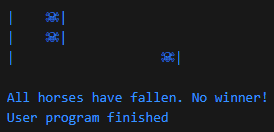
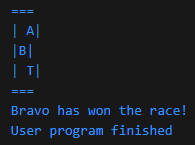
Creates a new process, and tells it to execute the following:

* **Cmd:** open windows command prompt
* **/c:** tells cmd to excecute the following command, then exit
* **Cls:** clear the screen in Windows

**.inheritIO()** makes the new process use the same terminal as the java program, **.starts()** starts the process, and **.waitFor()** waits for the process to finish before moving on.

**5. Race can be created with distance <= 0**

A race object can be created like this **Race race = new Race(30);**. The Race class constructor has no validation for distances <= 0, or if no distance is given.



Distance: -10 Distance: 0

**My Solution:**

I have given the **Race** class two constructors, one which sets a custom race length (or defaults to 10 if it is invalid). The other, which takes no params, sets the race length to 10. Both constructors call **initialiseHorses()** to set all lane horses to **null**.

**Race with length <= 0:**

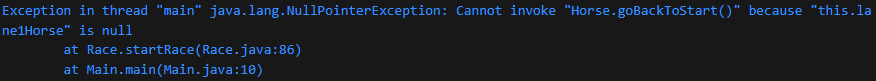


**Race with no length given:**

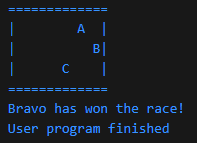


**6. NullPointerException thrown if no horse has been added**

Horses are initially set to **null** within the **Race** class. If **startRace()** is called without adding any horses, **NullPointerException** is thrown.

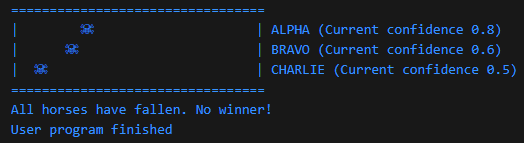


My solution is to initialise three default horses when the **Race** class constructor is called. This can be done easily as I created a method **initialiseHorses()** in the previous test case.



**7. Displays horse name and confidence during race**

The horse race video provided to use displays the horse name, and current confidence in front of the track.



This was implemented by adding a print statement at the end of the **printLane()** method:

System.out.printf(" %s (Current confidence %.2f)", theHorse.getName().toUpperCase(), theHorse.getConfidence());

Makes use of the getter methods of the **Horse** class. Also prints confidence to 2 decimal place

**8. Program updates confidence rating if horse wins or falls**

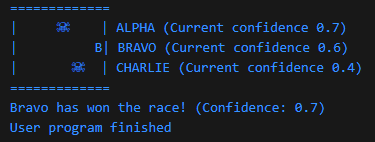
If a horse has won, the confidence rating increases by 0.1, and if it falls, it decreases by 0.1.

theHorse.setConfidence(theHorse.getConfidence() - 0.1);

Has been placed within **moveHorse()**. The code for both scenarios is quite similar. The setter method is used to set the value. The original confidence is retrieved using the getter method and 0.1 is subtracted from it.

theHorse.setConfidence(theHorse.getConfidence() + 0.1);

The above is the code for the win scenario and is placed within **raceWonBy()**.



Original confidences:

* 0.8
* 0.6
* 0.5