HORSE CLASS:

Encapsulation:

Encapsulation is a fundamental concept in OOP that involves putting together attributes(variables) and methods(code) into a single class, or restricting access to certain parts of code.

This is done by making fields or methods private and then using Accessor and Mutator(Getters and Setters) to change or access these fields. Encapsulations prevents unauthorised access to fields, and hence protects data integrity.

In the Horse Class, Encapsulation is used by:

- **Private Fields:** The variables **name, symbol, distanceTravelled, fallen** and **confidence** are private. This means that they are not accessible outside the class, and hence cannot be directly changed which prevents Misuse.
- **Accessor Methods:** These methods provide a way of retrieving the data from fields marked as private, without directly accessing them and risking data integrity, the list of accessor methods are:
    - **getSymbol()**
    - **getName()**
    - **getDistanceTravelled()**
    - **getConfidence()**
    - **hasFallen()**
    -
- **Mutator Methods:** These methods allow us to change the data in private fields, through methods, not by directly accessing the variable, ensuring data integrity. The list of mutator methods are:
    - **setSymbol(char newSymbol)**
    - **setConfidence(double newConfidence)**
    - **fall()**
    - **goBackToStart()**
    - **moveForward()**

Testing The Horse Class:

In order to test our Accessor and Mutator methods, I've set up a testHorse class. Test data is below. Before we can use the Horse class, we need to create it's constructor:

```
    public Horse(char horseSymbol, String horseName, double horseConfidence)
    {
       this.symbol = horseSymbol;
       this.name = horseName;
       this.confidence = horseConfidence;
       this.fallen = false; //by default horse is ready to go at the start of
race, not fallen
       this.distanceTravelled = 0; //dsitance is 0 at start
    }
```

This simply creates a new instance of the Horse class, and sets the variables to the parameters entered. Now we can test this constructor by simply calling it in the testHorse class:

1) Constructor Test:

```
        Horse newHorse = new Horse('♘', "Stallion", 0.7 );
```

This should create an instance of the Horse class with the symbol of a horse, name Stallion and a confidence rating of 0.7.

2) Movement Test:

```
    // We check the distance before we make the horse step
        System.out.println("Distance before step: " +
newHorse.getDistanceTravelled());
        //We make the horse step
        newHorse.moveForward();
        //Check the distance after we step, if the distance is 1, our
methods are working correctly.
        System.out.println("Distance after step: " +
newHorse.getDistanceTravelled());
```

3) Fallen Status:

```
        //Check if the horse is currently fallen, should return false.
        System.out.println("Has fallen?" + newHorse.hasFallen());
```

```
        //Call the fall method.
        newHorse.fall();
        //Check if the horse has now fallen, should return true if
method is working correctly.
       System.out.println("Has fallen?" + newHorse.hasFallen());
```

4) **Confidence Test:**

```
        //We know that the current horse confidence is 0.7, this method
call should set it to 0.3
        newHorse.setConfidence(0.3);
        //Expected output of 0.3 if methods are working correctly.
        System.out.println("Horse Confidence: " +
newHorse.getConfidence());
```

5) **Symbol Update Test:**

```
        //We know the current symbol is a 🐴, we want to change it to S
        newHorse.setSymbol('S');
        //Should output S if working correctly.
        System.out.println("Horse Symbol: " + newHorse.getSymbol());
```

6) **Back to Start Test**

```
        System.out.println("\nBack to start\n");
        //This should reset distance back to 0, and fall back to false.
        newHorse.goBackToStart();
        System.out.println("Has fallen?" + newHorse.hasFallen());
     System.out.println("Distance after step: " +
newHorse.getDistanceTravelled());
```

7) **Get Horse Name**

```
        System.out.println("Horse Name: " + newHorse.getName());
```

8) **Get Horse Symbol**

```
       System.out.println("Horse Symbol: " + newHorse.getSymbol());
```

9)

```
    System.out.println("Horse Confidence: " + newHorse.getConfidence());
```

RESULTS:

Green Number = Success , Red Number = Fail

1 and 2) 1 will be successful if 2 is(as 2 requires 1 to exist).

```
Distance before step: 0
Distance after step: 1
```

3)
```
Has fallen?false
Has fallen?true
```

4)
```
Horse Confidence: 0.3
```

5)
```
Horse Symbol: S
```

6)
```
Back to start

Has fallen?false
Distance after step: 0
```

7)
```
Horse Name: Stallion
```

8)
```
Horse Symbol: ?
```
//VSCode doesn't support some UNICODE values so the horse appears as a ?

9)
```
Horse Confidence: 0.7
```

Accessor Methods Testing:

Method	Function	Expected	Actual	Success
newHorse.getConfidence()	Returns horses confidence rating	Returns confidence of 0.7	Returned 0.7	Success
newHorse.getDistanceTravelled()	Returns the distance horse has travelled	Returns distanceTravelled	Returned 1	Success
newHorse.getName()	Returns the	Returns horseName	Returned Stallion	Success

	horse's name			
newHorse.getSymbol()	Returns the horse's symbol	Returns horse Symbol	Return ?'VS doesn't support Unicode symbols'	Success
newHorse.hasFallen()	Returns whether horse has fallen or not(T or F)	Returns T or F to whether the horse has fallen or not.	Returned F and then T after .fall called.	Success

Mutator Methods Testing:

Method	Function	Expected	Actual	Success
newHorse.setSymbol('S')	Sets horses's symbol	Horses' symbol should be set to 'S'	Was set to 'S'	Success
newHorse.setConfidence(0.3)	Set's confidence rating.	Horses' confidence should be set to 0.3	Was set to 0.3	Success
newHorse.fall()	Set's the fall Boolean to true.	Horses' fall Boolean should be set to true	Was set to true	Success
newHorse.goBackToStart()	Reset's the fall Boolean back to false and distance back to 0	Horses' fall Boolean should be set to false and distance set to 0, after being	Was reverted back to false and 0.	Success

		set to true and 1		
newHorse.moveForward()	**Increments distance by 1.**	**Horses' should increment to 1**	**Was incremented by 1.**	**Success**

Race Class:

Updates:

Updated the race class to handle simulation of horses, and to announce a winner at the end of the races.

Main Changes:

- A **List<Horse>** named winners has been added to track all horses that finish the race.
- The **raceWonBy()** method has been changed to add winning horses to the winners list.
- The **printResults**() method iterates over the winners list to announce all winning horses.
- The **startRace()** method logic now evaluates each horse's win condition separately to ensure multiple winners are printed if they win at the same time.

The constructor for the Race.java file is:

```
public Race(int distance)
{
    // initialise instance variables
    raceLength = distance;
    lane1Horse = null;
    lane2Horse = null;
    lane3Horse = null;
}
```

Takes in 3 horses to race, and a value for the distance of the track.

Like for the Horse.java, I created a testRace.java file to test whether the race functions or not.

1) **Constructor test:**

```
        Horse horse1 = new Horse('a', "Stallion", 0.6);
        Horse horse2 = new Horse('b', "WhiteArabian", 0.8);
        Horse horse3 = new Horse('c', "YellowMustard", 0.3);

        Race race = new Race(10);
```

2) **Adding Horses:**

```
        race.addHorse(horse1, 1);
        race.addHorse(horse2, 2);
        race.addHorse(horse3, 3);
```

3) **Starting the Race using startRace():**

```
        race.startRace();
```

4) **Race should display and results should print.**

```
            printRace();
```

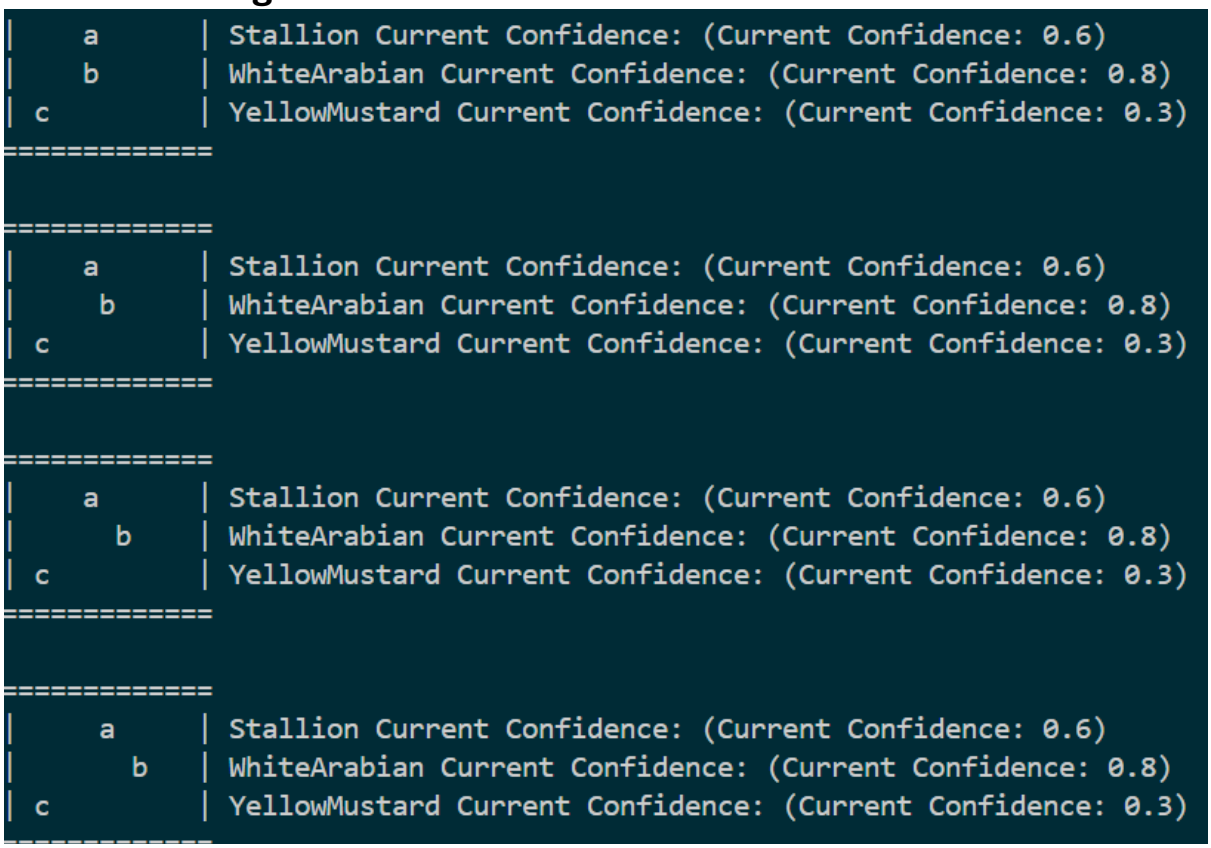
Results:

1, 2) 1 is checked with 2 as 2 depends on 1. We can see 3 horses and the race with it's borders so an instance is created and the horses are input.

```
=============
| a          | Stallion Current Confidence: (Current Confidence: 0.6)
| b          | WhiteArabian Current Confidence: (Current Confidence: 0.8)
|c           | YellowMustard Current Confidence: (Current Confidence: 0.3)
=============
```

3,4) We can see that the horses move on each update, so the race has started and is working. The race is being printed so number 4

is also working.



Methods	Function	Expected Output	Actual Output	Success or failure
startRace()	Resets the variables Runs all the simulations acting as a main method	The race starts with the three horses. The user can see the race updating in their terminal as print statements.	User can see the race in their terminal. Working as expected.	Success
moveHorse()	When horse moves, the distance is increased by 1, and the horse has a chance to fall depending on it's confidence level.	Horse either moves forward, and falls. If falls, marked with an X.	Works as expected. Horses that fall are marked with X, horses that have distance increased	Success

			move forward	
raceWonBy()	Responsible for checking the distance covered by each horse. If equal to track distances, the hrose is a winner.	Adds the horse to the horse winners array. Also checks for duplicates, if duplicates	Successfully adds winners to the array list.	Success
printRace()	Prints the actual race with the horses moving forward update.	Should print the edges of the race as well as the actual race of the horses.	Correctly updates the horses and you can see them running in the terminal	Success
addHorse()	Adds a horse to a lane	Assigns a horse to the specified lane.	Works as expected.	Success
Race()	Constructor	Should create an instance of the Race.	Creates an instance of the Race.	Success
multiplePrint()	Adds spaces before and after the horse to give the illusion of it moving.	As the horse moves forward, the space behind increases and the space infront decreases giving the illusion of movement.	Spacing is working as expected	Success