



ITWM5113
Software Design and Development

FINAL PROJECT: ANIMAL KINGDOM

Instructor : DR. HADI NAGHAVIPOUR
Student Name : LEE LAY PENG
Student ID : (MC220316722)

Table of Contents

1. Abstract	2
2. Introduction	3
3. Project Description	4
4. Project Design	5
5. Program workflow and logics	7
5.1 Bear Class	7
5.2 Tiger Class	11
5.3 WhiteTiger Class	14
5.4 Giant Class	15
5.5 NinjaCat Class	17
6. Development details	18
7. Results	19
8. Conclusion	23

1. Abstract

Object-oriented programming (OOP) is a software design model that focuses on organizing programming logic around objects, which are data fields with unique attributes and behaviour. OOP is well-suited for large and complex programs that require active updates and maintenance. This report explores the concepts of OOP, including abstraction, encapsulation, inheritance, and polymorphism, with a focus on inheritance as one of the main concepts in OOP. The report also discusses the challenges and complexities of implementing OOP in Java for a project that involves modelling the behaviour of different animals in a simulated world. The research and testing conducted for this project are highlighted, as well as the benefits and applications of OOP in various industries such as manufacturing, design, and mobile applications.

2. Introduction

The Animal Kingdom project is a requirement for ITWM5113 - Software Design and Development, supervised by Dr. Hadi Naghavipour. The objective of this project is to gain a deeper understanding of Java classes and their implementation. The project involves writing a set of classes that define the behavior of specific animals, including Bear, Giant, NinjaCat, Tiger, and WhiteTiger, as outlined in the project description. The behaviour of these animals encompasses their actions, colours, and string representations. Through this project, we can practice creating classes that characterize different animals. The project provides a set of Java files, and our task is to incorporate the behaviour codes into the project file itself. Once compiled and run, the project will display a simulation window that represents the behaviour of each animal for review.

For this assignment, I have setup the Project with the name “Final Project - Lee Lay Peng MC220316722”. The language used is JAVA as per the requirement of this assignment. The build is using IntelliJ and the JDK chosen is 19 Oracle OpenJDK version 19.0.2.

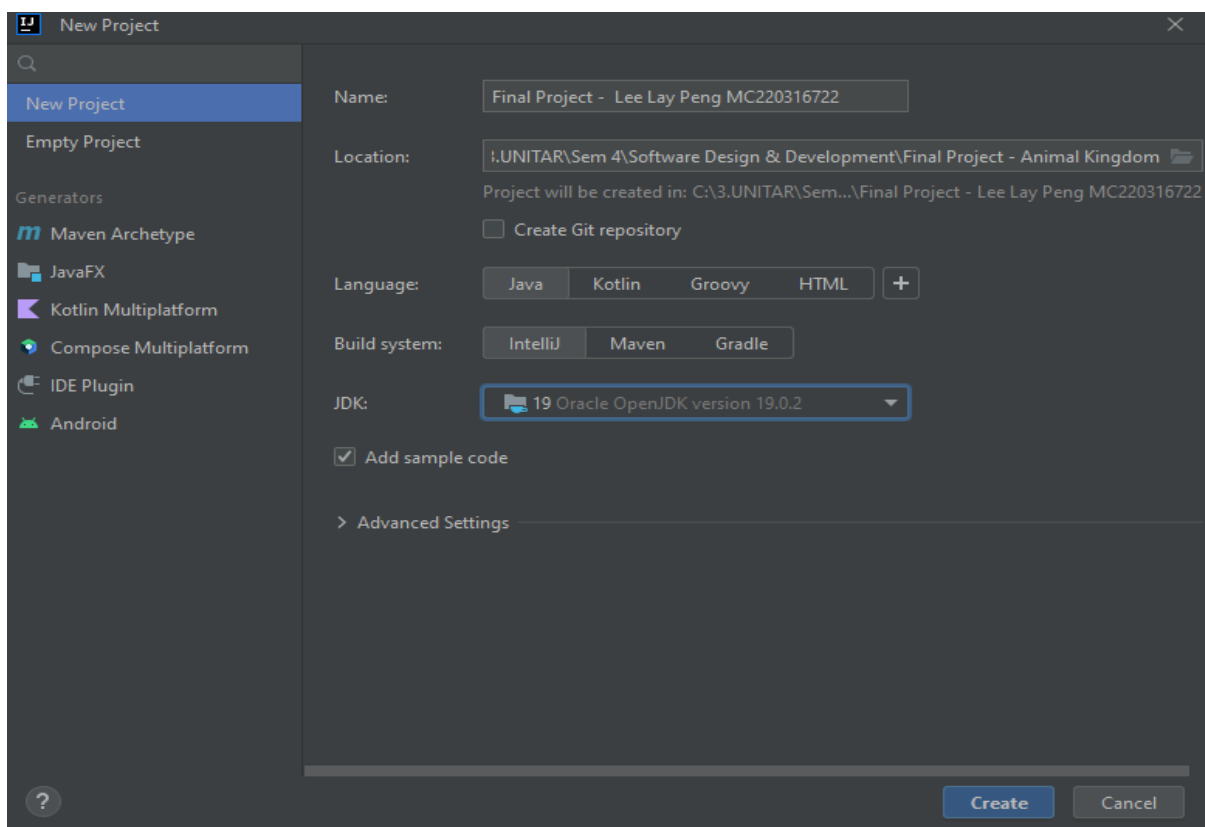


Figure 1 – IntelliJ IDEA IDE Setup

3. Project Description

For this project, we are given a program that runs a simulation of a world with many animals wandering around in it. Different kinds of animals will behave in different ways, and we are required to define those differences. We are also provided with eight (8) files as follow:

- a. Critter.java
- b. CritterFrame.java
- c. CritterInfo.java
- d. CritterMain.java
- e. CritterModel.java
- f. Critter Panel.java
- g. FlyTrap.java
- h. Food.java

The supporting codes will create the frames of the simulation, the default value of the classes, the critter class itself and the Critter main method that will run the simulation.

Initially there will be 2 existing critter (Fly Trap and Food Classes) that are by default existed in the critter world as per figure 2 below shown:

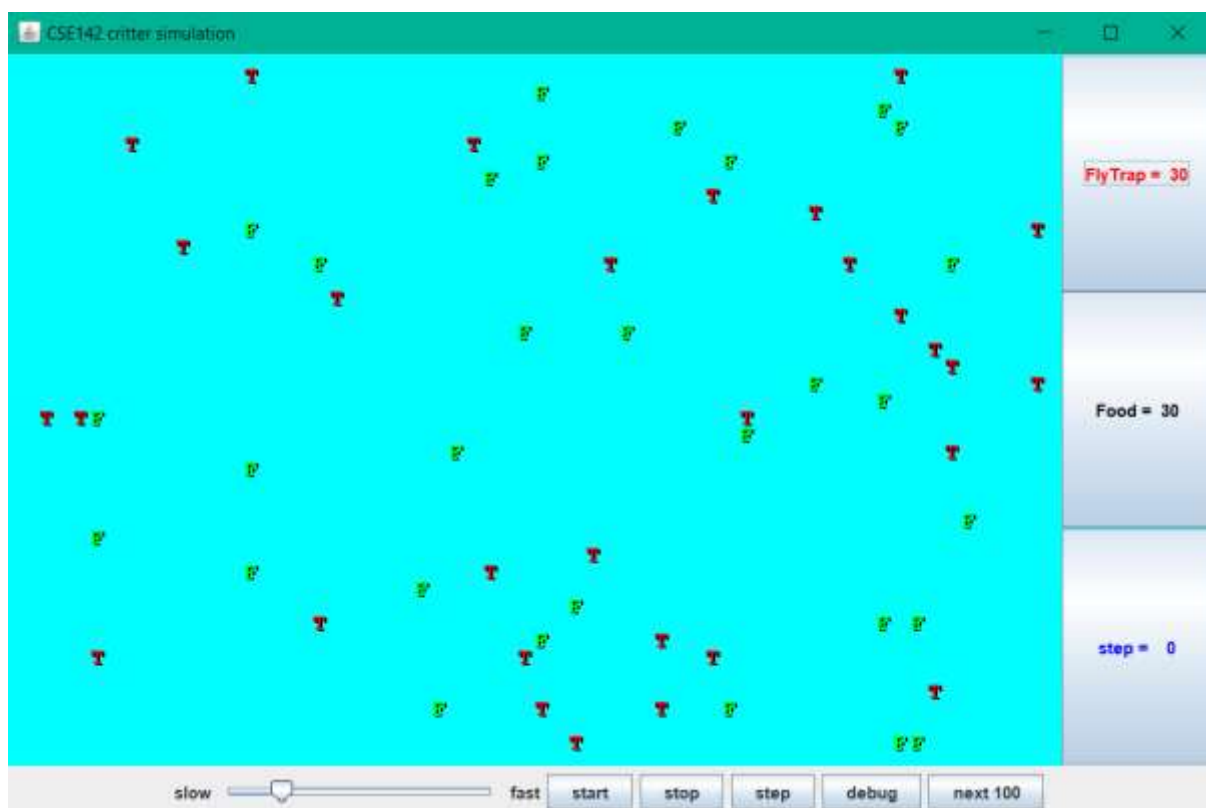


Figure 2 – Critter Simulator Window with Fly Trap and Food Classes

4. Project Design

The project design for this project is based on the Object-oriented classes that has the critter class as the parent class and has seven sub-classes, namely Fly Trap class, Food class, Bear class, Tiger class, White Tiger class, Giant class, and Ninja Cat class.

Class Relationship - "IS-A"

We determine the type of the relationship between provided classes, the relationship between sub classes and parent class is "IS-A" relationship. In object-oriented programming, the concept of IS-A is a totally based on Inheritance.

The methods in the sub-classes will be overriding the method it inherited from the Critter class, to implement the change of behavior of each sub-class. The following will be the methods and functions for each class as detailed in the source code (Figure 3-8).

Critter class
<pre>public static enum Neighbor { public static enum Action { public static enum Direction { public Action getMove(CritterInfo info) { public Color getColor() { public String toString() { public final boolean equals(Object other)</pre>

Figure 3 – Methods and Functions for Critter class

Bear extends Critter class
<pre>private Direction test private String bearchar public Action getMove(CritterInfo info) public Color getColor() public String toString()</pre>

Figure 4 – Methods and Functions for Bear extends Critter class

Giant extends Critter class
<pre>private static int count private static int count2</pre>
<pre>public Critter.Action getMove(CritterInfo info) public Color getColor() public String toString()</pre>

Figure 5 – Methods and Functions for Giant extends Critter class

NinjaCat extends Critter class
<pre>public Critter.Action getMove(CritterInfo info) public Color getColor() public String toString()</pre>

Figure 6 – Methods and Functions for NinjaCat extends Critter class

Tiger extends Critter class
<pre>private static int count private static int count2</pre>
<pre>public Action getMove(CritterInfo info) public Color getColor() public Color randomColor() public String toString()</pre>

Figure 7 – Methods and Functions for Tiger extends Critter class

WhiteTiger extends Critter class
<pre>private boolean test</pre>
<pre>public Critter.Action getMove(CritterInfo info) public Color getColor() public String toString()</pre>

Figure 8 – Methods and Functions for WhiteTiger extends Critter class

5. Program workflow and logics

The program workflow for this project is mainly done separately for each class. As the methods in each of the sub-class will be overriding the Critter class. So, the program of the data inputted in each class will be handled separately and modified in each class. The following section will describe the behaviour of each animal:

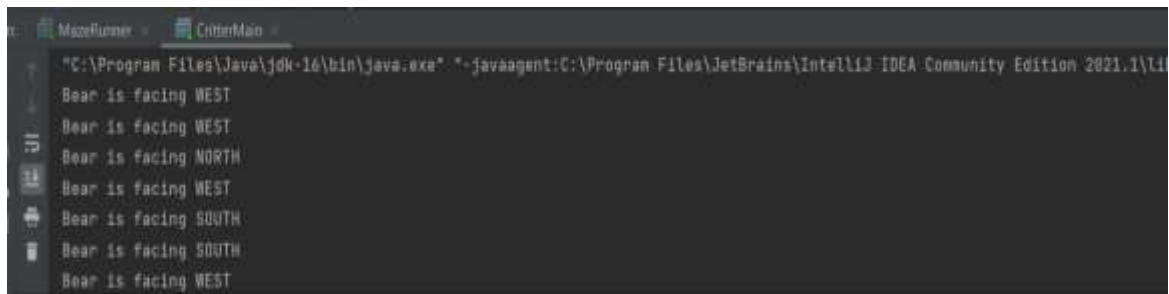
5.1 Bear Class

Constructor	public Bear(boolean polar)
getColor	Color.WHITE for a polar bear (when polar is true), Color.BLACK otherwise (when polar is false)
toString	Should alternate on each different move between a slash character (/) and a backslash character (\) starting with a slash.
getMove	always infect if an enemy is in front, otherwise hop if possible, otherwise turn left.

- To determine the color of the bear, a direction variable test has been created to accept the value of each bear to check to which direction it is facing.
- In Figure 9, a test println command is used to check what is the direction for each critter. Then and if else condition was implemented in the getColor to return the value of color for each bear depending on its direction it facing to the Critter class.
- As for the toString method to determine the string to represent the critter bear, an initial string variable of "bearchar" has been initiated as forward slash (/). If else, condition is used to alternate the string between (/) and (\).
- For getMove, the requirement is "always infect if an enemy is in front, otherwise hop if possible, otherwise turn left." In order do this, below input from the surroundings of the critter is explained.

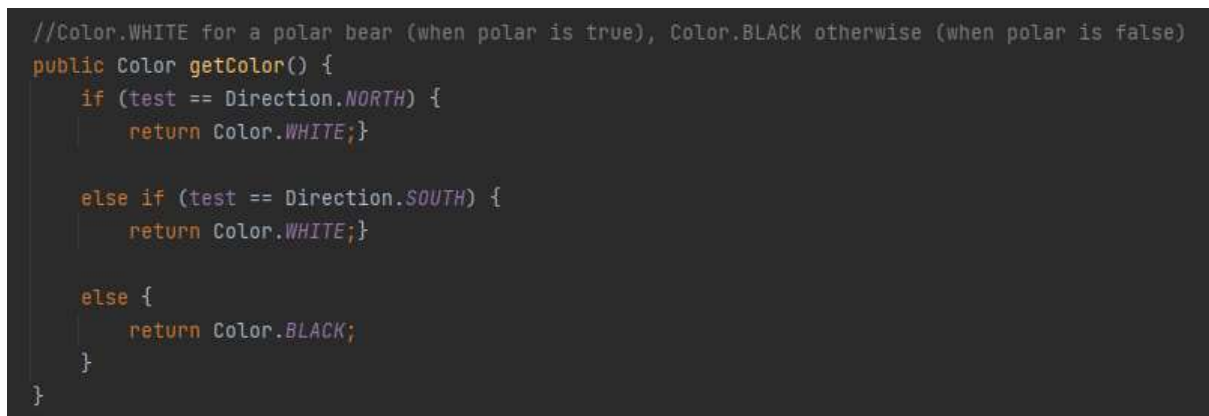
info.frontThreat() == true - if an enemy is in front

info.getFront() == Neighbor.EMPTY -there is nothing in front



```
"C:\Program Files\Java\jdk-16\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2021.1\lib\idea_rt.jar=1273:..."
Bear is facing WEST
Bear is facing WEST
Bear is facing NORTH
Bear is facing WEST
Bear is facing SOUTH
Bear is facing SOUTH
Bear is facing WEST
```

Figure 9 – Screenshot of test for `println info.frontThreat()` for Bear class



```
//Color.WHITE for a polar bear (when polar is true), Color.BLACK otherwise (when polar is false)
public Color getColor() {
    if (test == Direction.NORTH) {
        return Color.WHITE;
    }

    else if (test == Direction.SOUTH) {
        return Color.WHITE;
    }

    else {
        return Color.BLACK;
    }
}
```

Figure 10 – Condition that will return the value of color for each bear depending on its direction its facing.

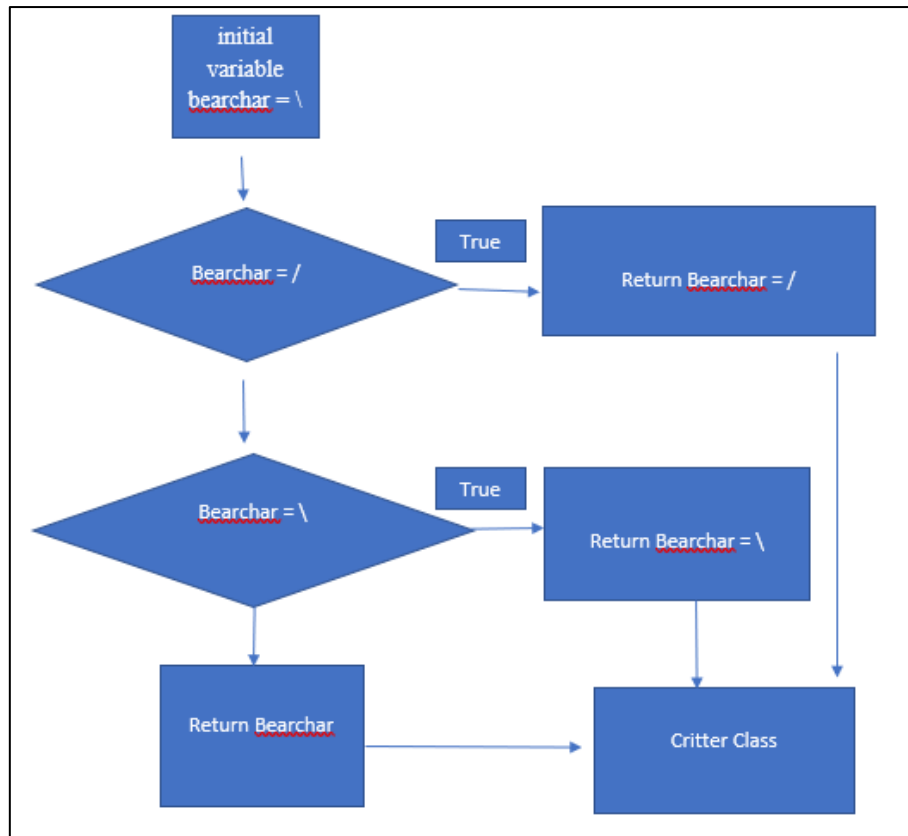


Figure 11 – getMove Constructor flow chart for Bear class

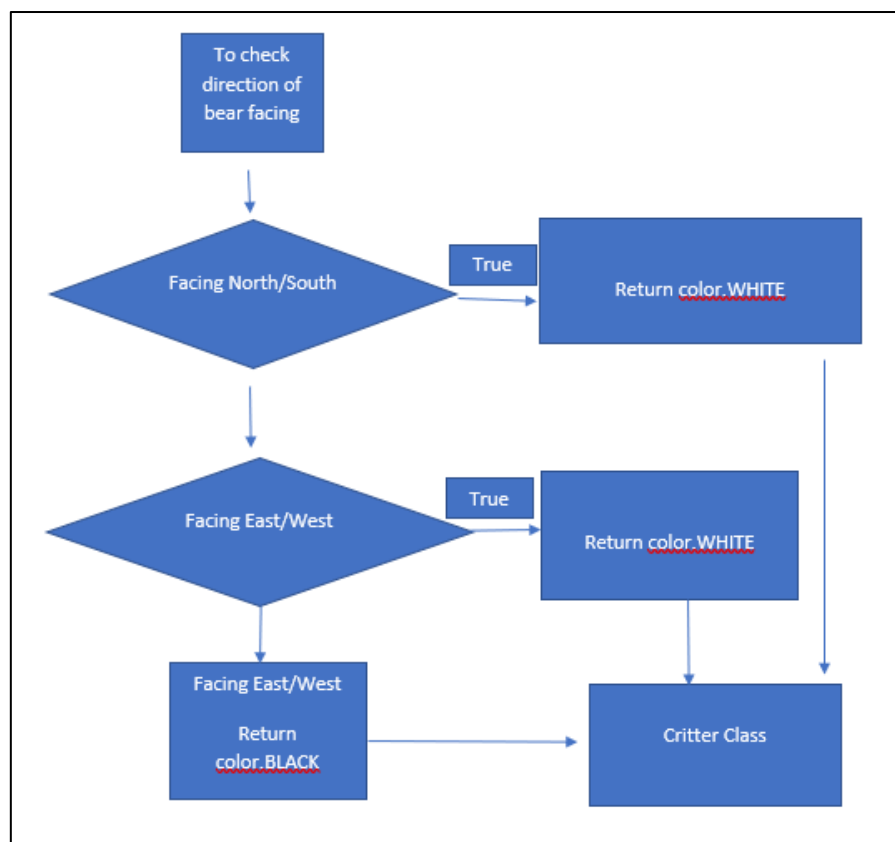


Figure 12 – getColor Constructor flow chart for Bear class

```

public Action getMove(CritterInfo info) {

    test = info.getDirection();

    if (info.frontThreat()) {
        return Critter.Action.INFECT;
    }
    else if (info.getFront() == Neighbor.EMPTY) {
        return Critter.Action.HOP;
    }

    else {
        return Action.LEFT;
    }
}

```

Figure 13 – Action to be taken when facing an enemy in front of when there is nothing in front.

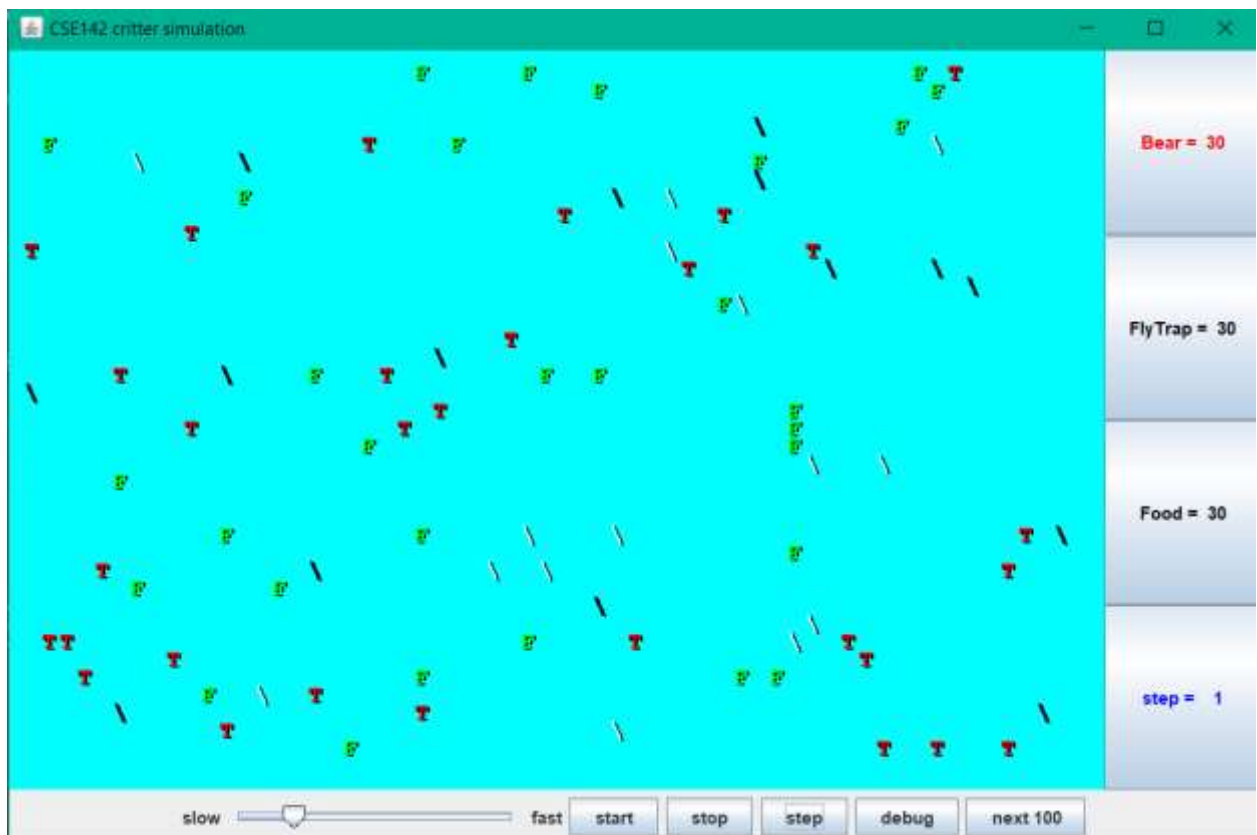


Figure 14 – Testing done just with Bear active.

5.2 Tiger Class

Constructor	public Tiger()
getColor	Randomly picks one of three colors (Color.RED, Color.GREEN, Color.BLUE) and uses that color for three moves, then randomly picks one of those colors again for the next three moves, then randomly picks another one of those colors for the next three moves, and so on.
toString	"TGR"
getMove	always infect if an enemy is in front, otherwise if a wall is in front or to the right, then turn left, otherwise if a fellow Tiger is in front, then turn right, otherwise hop.

- In Figure 16, a test println command is used to check if the randomize function works and the randomize number will pinpoint to one out of all three color red, green and blue. Then, the color will pass to the getColor() method and to pass to critter class. As for the requirement for getMove, this is done as per figure 17.
- For toString, this is a straightforward to override the Critter class with the string "TGR".
- For getMove, the tiger will always infect if an enemy is in front, otherwise if a wall is in front or to the right, then turn left, otherwise if a fellow Tiger is in front, then turn right, otherwise hop.

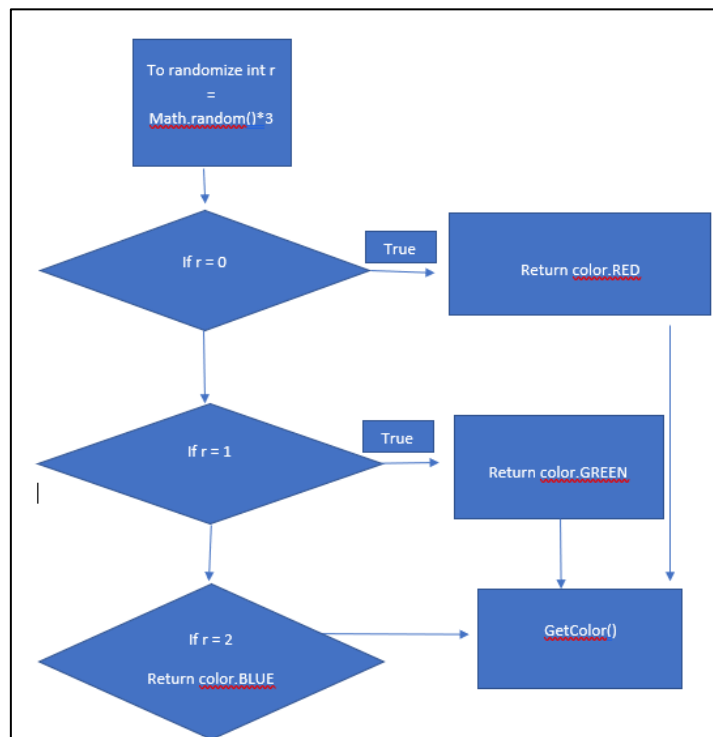


Figure 15 – getColor Constructor flow chart for Tiger Class

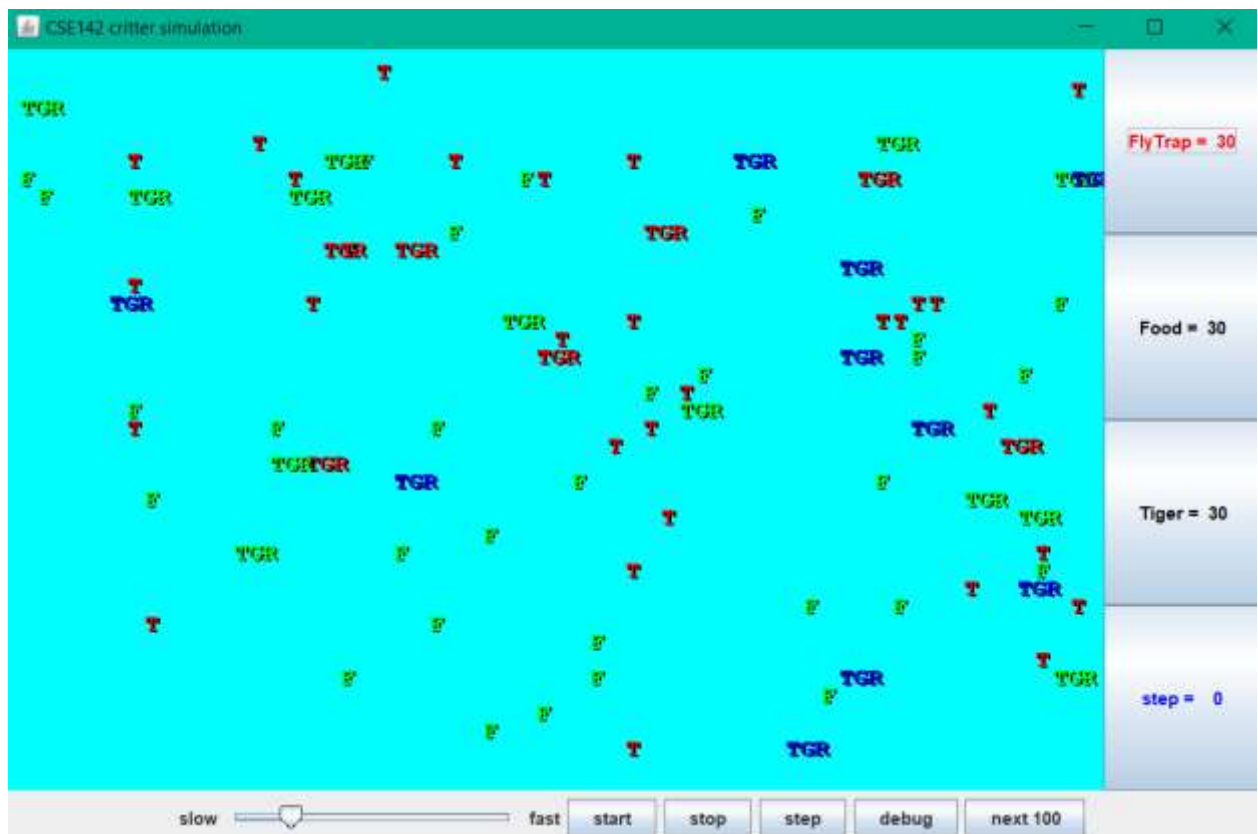


Figure 18 – Testing done just with Tiger active.

5.3 WhiteTiger Class

Constructor	public WhiteTiger()
getColor	Always Color.WHITE.
toString	"tgr" if it hasn't infected another Critter yet, "TGR" if it has infected.
getMove	Same as a Tiger. Note: you'll have to override this method to figure out if it has infected another Critter.

WhiteTiger behaves the same as a Tiger.

- For this WhiteTiger class, the complicate part is to identify if the WhiteTiger critter has infected other critters for the ToString method. Once it has infected the string of WhiteTiger will change to capital letters TGR.
- So, a Boolean variable test is introduced to recognized if there is any enemy in front of the WhiteTiger and infected the enemy. Once the variable test indicates true value, this will be passed down to the ToString () method. Then a condition will be applied and if the test variable true will make the WhiteTiger string as TGR and lowercase tgr otherwise as shown below:

```
// "tgr" if it has not infected another Critter yet, "TGR" if it has infected.  
public String toString() {  
  
    if (test) {  
        return "TGR";  
    }  
  
    else {  
        return "tgr";  
    }  
}
```

Figure 19 – toString construction for WhiteTiger Class

5.4 Giant Class

Constructor	public Giant()
getColor	Color.GRAY
toString	"fee" for 6 moves, then "fie" for 6 moves, then "foe" for 6 moves, then "fum" for 6 moves, then repeat.
getMove	always infect if an enemy is in front, otherwise hop if possible, otherwise turn right.

- For the Giant class, the method that needs attention is toString whereby the string will change every 6 moves as shown from the behaviour. To complete this method two counter (count and count2) variables have been initiated.
- The first counter is to count the number of the critter, and once the critter reaches 30, the next counter will divide the critter number to 30, this will show the step counter. The step counter is important to determine that once the step counter reaches 6 it will pass a different string to the critter class.
- Below 2 screenshots shown how the counter were used and the test println to show that the counter works for each step.

```
// "fee" for 6 moves, then "fie" for 6 moves, then "foe" for 6 moves, then "fum" for 6 moves, then repeat.
public String toString() {

    count++;
    int count2 = count / 30;

    if (count2 == 24)
    {
        count = 0;
    }

    if (count2 <= 6)
    {
        return "fee";
    }

    if (count2 <= 12)
    {
        return "fie";
    }
}
```



```
{  
    return "foe";}  
  
else  
  
{  
    return "fum";  
}  
  
}
```

Figure 20 – toString construction for Giant Class

5.5 NinjaCat Class

Constructor	public NinjaCat()
getColor	You decide
toString	You decide
getMove	You decide

- **getColor** – Color.ORANGE
- **toString** – “*”
- **getMove** - Always infect if any other than NinjaCat is in front, Right, Left and Back. Also always infect if an enemy is in front, Right, Left and Back, then, if Wall on the right or in front turn left, if meet Ninja Cat in front turn right and if in front is empty space then hop, if possible, otherwise hop again.

For own's critter Ninja cat, the strategy for the getMove() method is trying to be offensive as possible and to infect anything in front, right, left and back critter.

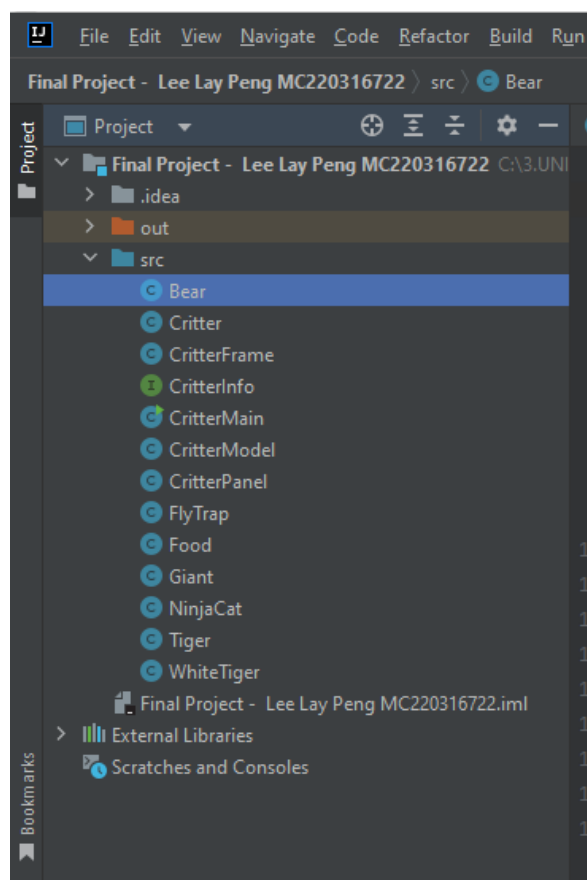


Figure 21 – Screenshot for all classes had been successfully create

6. Development details

The development of the project uses IntelliJ and Java as the programming language. This final project was a bit complicated due to a lot of understanding of Object-oriented programming that was researched and studied to enable this program to be completed at the end. All the animal classes were developed individually and tested separately as demonstrated in Figure 22. This was done to make it easier to identify any errors that were detected. In some of the classes, print statements were also added to ensure that calculations were correct and that the values of variables were correctly passed to the intended methods and eventually to the Critters class.

The assignment questions provided insight into what requires to be developed along with the testing scenario as follow:

Below are some suggestions for how you can test your critters:

- **Bear:** Try running the simulator with just 30 bears in the world. You should see about half of them being white and about half being black. Initially they should all be displayed with slash characters. When you click "step", they should all switch to backslash characters. When you click "step" again they should go back to slash characters and so on. When you click "start", you should observe the bears heading towards walls and then hugging the walls in a counterclockwise direction. They will sometimes bump into each other and go off in other directions, but their tendency should be to follow along the walls.
- **Tiger:** Try running the simulator with just 30 Tigers in the world. You should see about one third of them being red and one third being green and one third being blue. Use the "step" button to make sure that the colors alternate properly. They should keep these initial colors for three moves. That means that they should stay this color while the simulator is indicating that it is step 0, step 1, and step 2. They should switch colors when the simulator indicates that you are up to step 3 and should stay with these new colors for steps 4 and 5. Then you should see a new color scheme for steps 6, 7, and 8 and so on. When you click "start" you should see them bouncing off of walls. When they bump into a wall, they should turn around and head back in the direction they came. They will sometimes bump into each other as well. They shouldn't end up clustering together anywhere. **WhiteTiger:** This should behave just like a Tiger except that they will be White. They will also be lower-case until they infect another Critter, then they "grow up".
- **Giant:** Try running the simulator with just 30 giants in the world. They should all be displayed as "fee". This should be true for steps 0, 1, 2, 3, 4, and 5. When you get to step 6, they should all switch to displaying "fie" and should stay that way for steps 6, 7, 8, 9, 10, and 11. Then they should be "foe" for steps 12, 13, 14, 15, 16, and 17. And they should be "fum" for steps 18, 19, 20, 21, 22, and 23. Then they should go back to "fee" for 6 more steps, and so on. When you click "start", you should observe the same kind of wall-hugging behavior that bears have, but this time in a clockwise direction.

Figure 22 – Steps to test the critter

7. Results

Upon completion of the critter's development, the critter simulation is being carried out. Figure 23 demonstrate the screenshot of the simulator before initiated. The default number of animal is 30 for every group of animals.

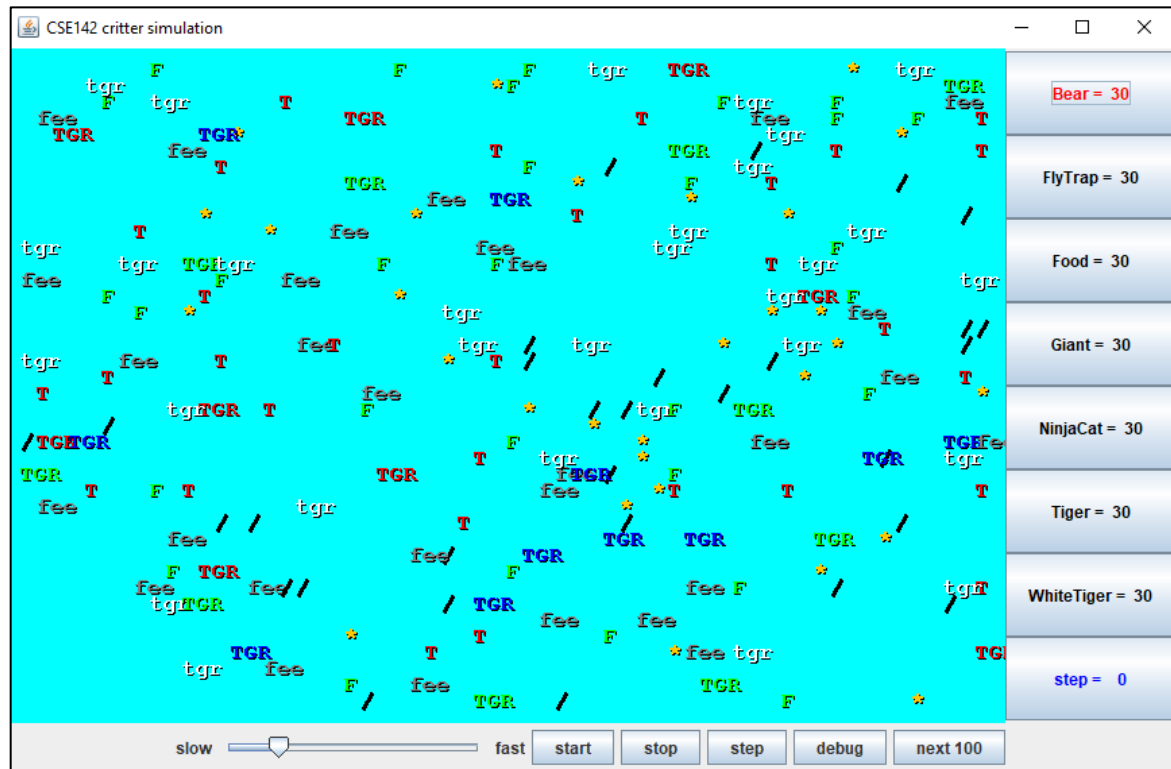


Figure 23 – Critter Simulator Window before initiate

In order to start the stimulation, we can press the “start” button. All animals were released and start roaming according to their behaviour as required by this project as in Figure 24. There are 2 ways to speed up the stimulation process.

- (i) By press “next 100” button as shown in Figure 25
 - It will display the result of next 100 steps
- (ii) By toggle the slow or fast slider as shown in Figure 26
 - It can adjust the stimulation speed

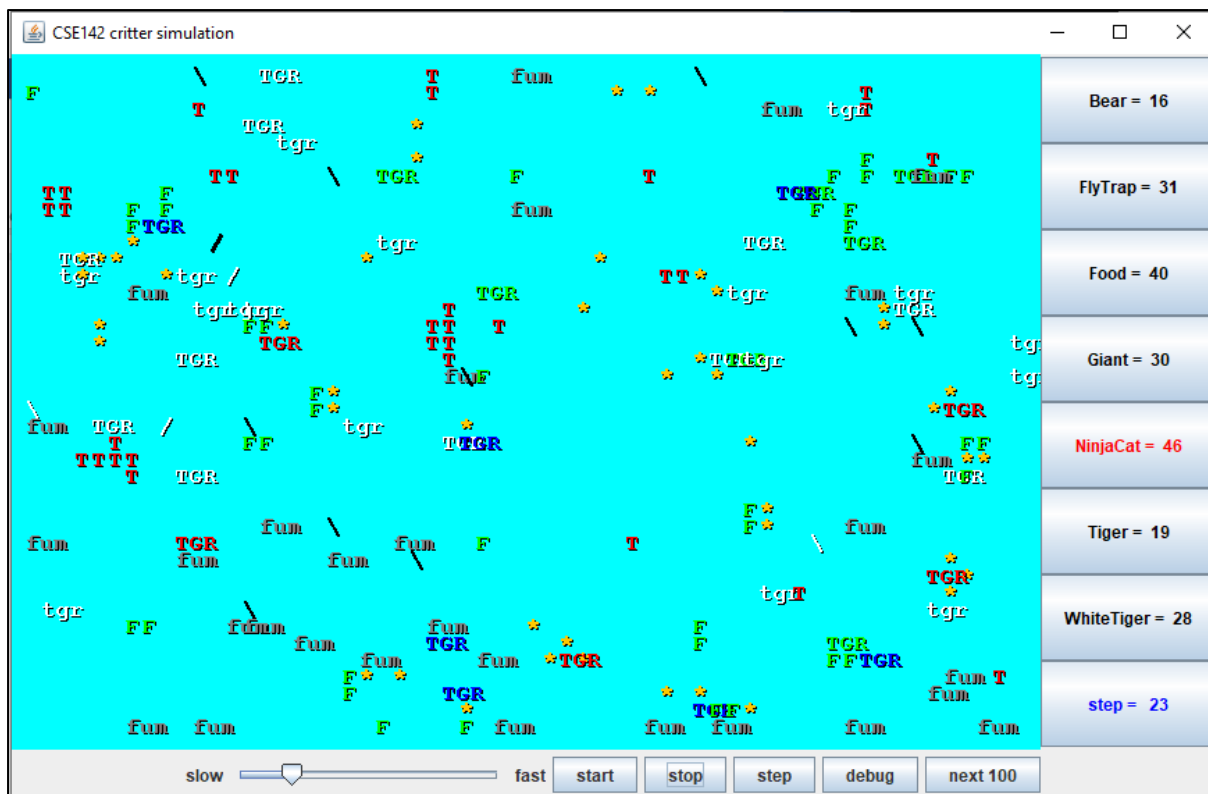


Figure 24 – Critter Simulator Window upon initiated (when animal start roaming)

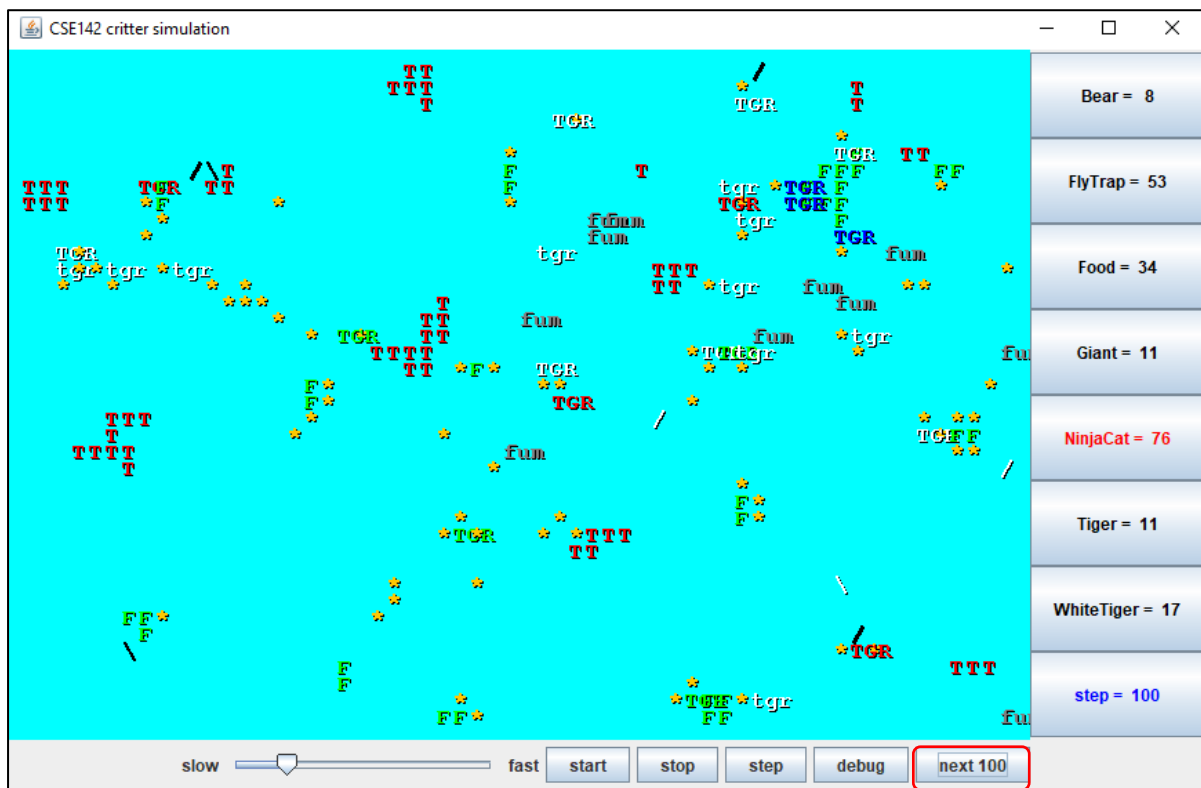


Figure 25 – Critter Simulator Window that display the next 100 results

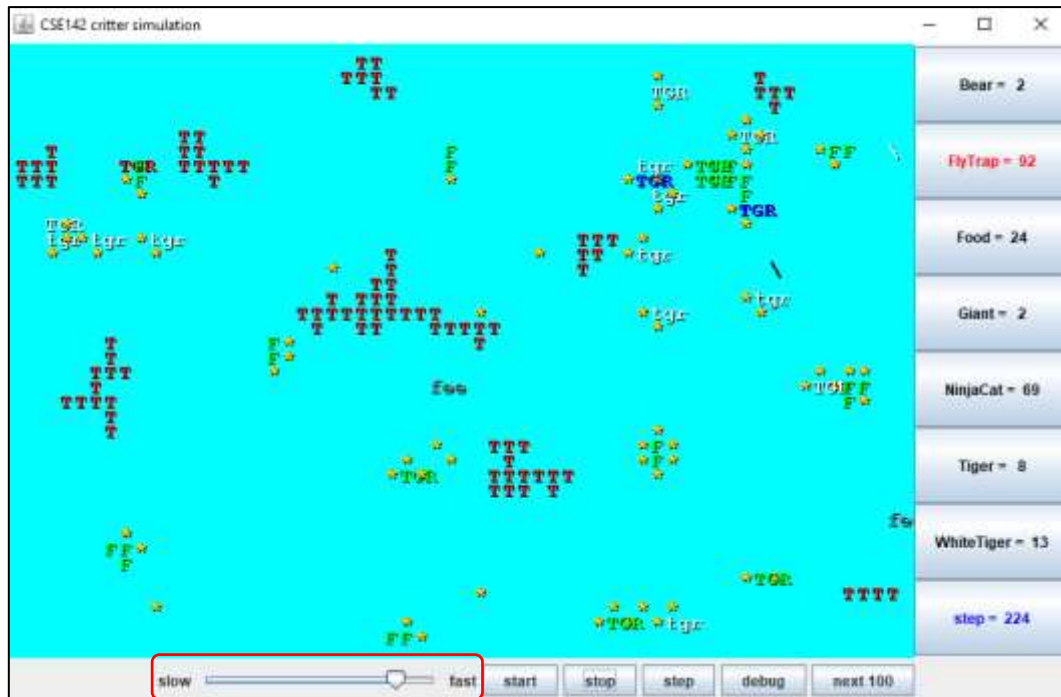


Figure 26 – Critter Simulator Window upon initiated with fast speed

The stimulation stopped when the counter showed 2000 steps. It was also noted that the FlyTrap performed exceptionally well (101) compared to the other animals, followed by NinjaCat (67). while the Bear and Giant species went extinct completely (0). The Tiger (8) and White Tiger (13) species barely survived. In essence, the results show that all the animal classes performed effectively.

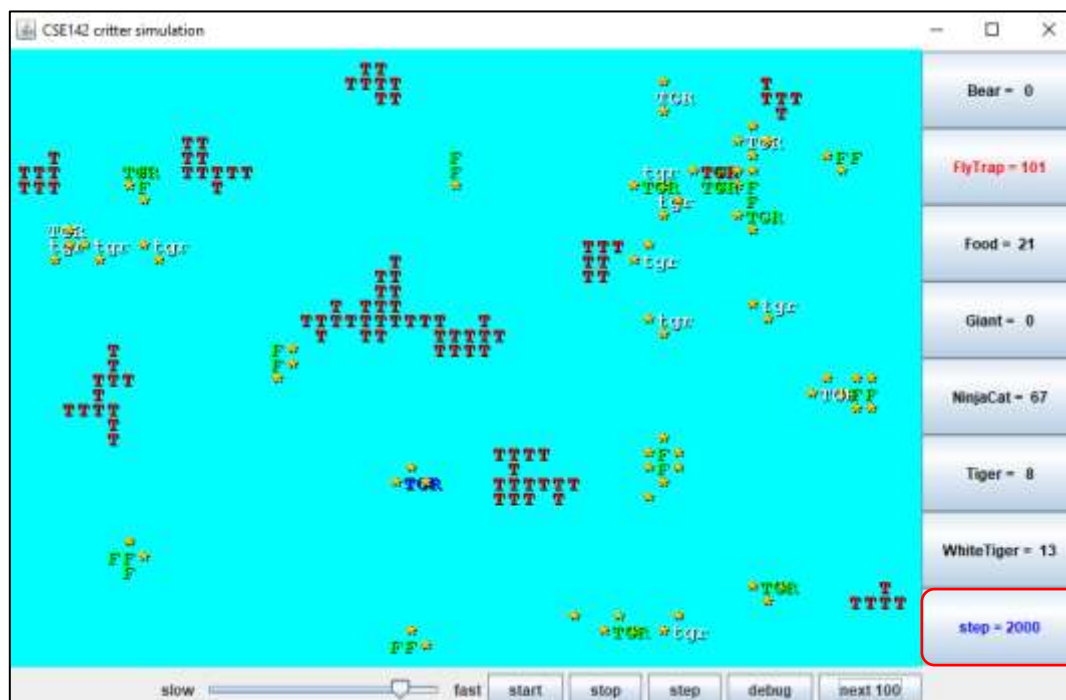


Figure 27: Critter Simulator Window upon 2000 steps

The simulator can also be placed in debug mode to observe the movement of the animals as shown in Figure 28.

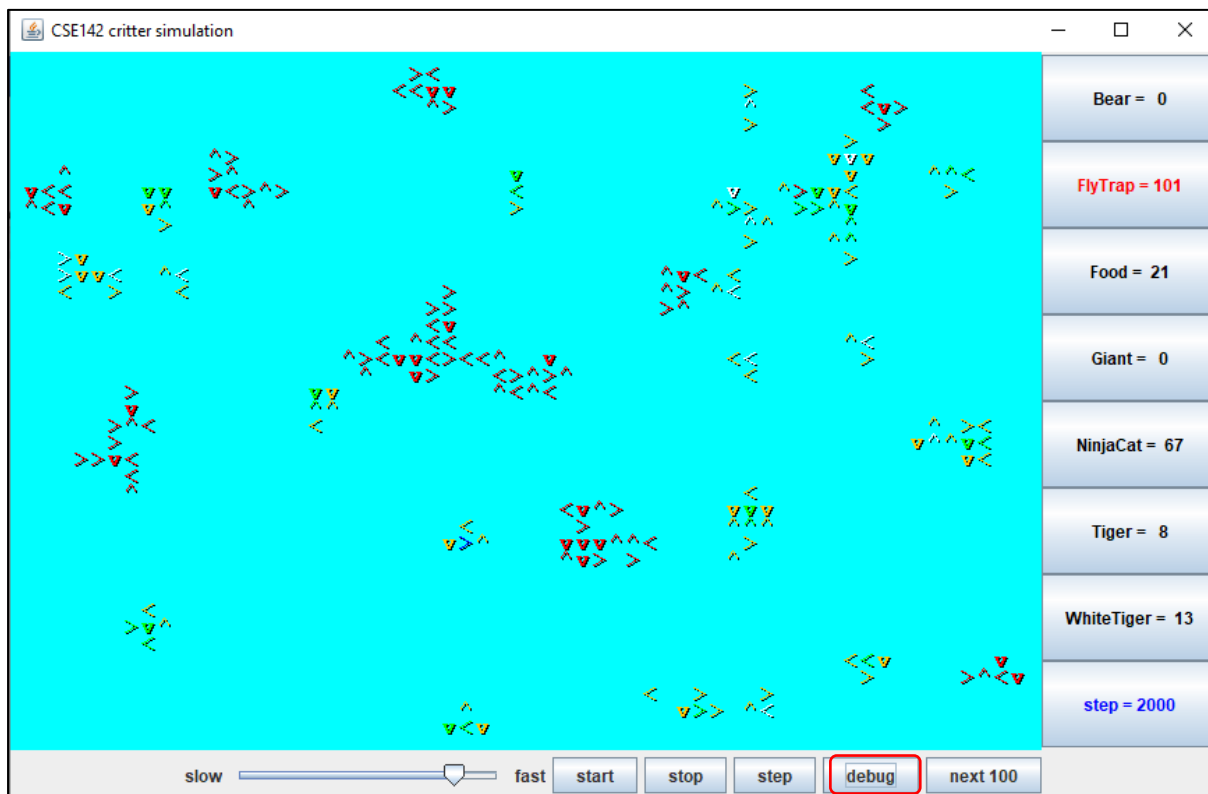


Figure 28 – Critter Simulator Window in debug mode

8. Conclusion

The conclusion drawn after completing this final project is that a better understanding of object-oriented programming has been achieved. It was learned that a class can be inherited by other subclasses, and all the variables and methods from the parent class are inherited by the subclasses. Additionally, methods in the parent class can be overridden by subclasses, allowing them to add new values that override the methods in the parent class. This was evident in the animal subclasses, where all the methods in the subclasses were overridden to update the default values in the methods of the parent class. Furthermore, the manipulation of variable values was successfully implemented to meet the requirements of this project. Lastly, I would like to express my gratitude to Dr. Hadi for his exceptional lectures and generous sharing of knowledge throughout the entire course. Thank you!