ITWM5013 SOFTWARE DESIGN AND DEVELOPMENT

Final Assignment

Instructor: DR. Hadi Naghavipour

Student Name & ID
LEELA SHANTI A/P SUNDRSEGARAN
(MC220919180)

UNITAR

TABLE OF CONTENTS

1.0	**ABSTRACT**	**3**
2.0	**INTRODUCTION**	**4**
3.0	**DESCRIPTION**	**7**
4.0	**DESIGN**	**8**
5.0	**WORKFLOW AND LOGIC**	**10**
6.0	**DEVELOPMENT**	**13**
7.0	**RESULTS**	**22**
8.0	**DISCUSSION**	**24**
9.0	**CONCLUSION**	**27**

FINAL PROJECT – ANIMAL KINGDOM SIMULATOR

1.0 ABSTRACT

With the correct architecture, we may structure programmes so that they are expandable, which is one of the advantages of object-oriented programming. In other words, a programmer can create a software with the idea that other coder will extend its capabilities using classes. Instead of creating a programme from scratch for the final assignment of the semester, we will practise developing classes in the context of someone adding functionality to an already-existing programme.The term "object-oriented programming," or OOPs, is used to describe programming languages that use objects as a main source for carrying out what is intended in the code. As objects perform the tasks you've assigned them, the user or viewer sees them in action. The aim of object-oriented programming is to implement actual principles from real life, such as polymorphism, inheritance, encapsulation, and abstraction, in code.

This report shows on how this animal kingdom simulation application is coded using Java's Object-Oriented Programming (OOP).

In this assignment, you will be given a programme that replicates a world filled with countless free-roaming animals as well as instructions on how to construct a set of classes that specify the behavior of particular creatures. You are defining the variations in animal behavior that occur between species. The animal kingdom simulation application was developed by using Eclipse IDE.

2.0 INTRODUCTION

The opportunity to practise using a set of classes that outline the traits of particular animals is offered in this application. The programme's output shows the numerous animals roaming the area. The utilisation of various objects and classes, as well as their interactions, are the main focus of this application.

You will receive a substantial amount of supporting code for this assignment, which executes the simulation. We are identifying these characteristics because different animal species will display different behaviours.

Each class in the Java programme that simulates the Animal Kingdom has a specific purpose. The below codes have been provided.

Critter.java

This class stands in for the base class for all Animal Kingdom species. It lists common traits and behaviors, like size, color, and modes of locomotion, that other animal classes inherit.

Critter.Frame.java

The graphical user interface (GUI) frame for the programme is represented by this class. It provides the viewing window for the virtual Animal Kingdom, which displays the animals and their interactions.

CritterInfo.java

This class is an interface that provides information on an animal's characteristics, including its location, movement, and amount of energy.

CritterMain.java

This class, which also serves as the programme's entry point, contains the main method for starting the simulation of the Animal Kingdom.

CritterModel.java

The simulation's model for the Animal Kingdom is this class. Along with capturing all of the simulation's organisms' states and activities, it also records their positions, moves, and interactions.

CritterPanel.java

This class represents the GUI panel where the drawings and displays of the creatures are displayed. It is in charge of updating the animals' positions and rendering the creatures during the simulation.

FlyTrap.java

In the Animal Kingdom, this class, FlyTrap.java, represents a specific kind of animal. By deriving from the Critter class, it describes its own traits and behaviors, such as its ability to capture other creatures.

Food.java

This class represents the foods that animals can eat to get energy in the Animal Kingdom simulation. It defines details about how food is produced and consumed, as well as its accessibility and location.

Bear.Java

This class is specifically an animal species called the Bear.

Giant.java

This class corresponds to the species of Giants in the Animal Kingdom. It explains its traits and behaviours by deriving from the Critter class, such as its size and gait.

NinjaCat.java

This class is specifically an animal species called the NinjaCat. By deriving from the Critter class, it establishes its distinctive traits and behaviors, such as its sly movement and interactions with other species.

Tiger.java

A specific species of animal in the animal realm is represented by this class: the tiger. It is a subclass of Critter and describes the creature's distinctive traits, including speed and propensity for hunting.

WhiteTiger.java

This class features a particular kind of animal from the Animal Kingdom: The WhiteTiger. The Tiger class is its ancestor, and it establishes its particular qualities, such as color and interactions with other animals.

3.0 DESCRIPTION

For this project, we must build Java code where the superclass Critter has predefined default behavior. Each of the five courses we'll be writing about represents a different kind of animal: Bear, Tiger, WhiteTiger, Giant, and NinjaCat. The correct subclass of every class should be Critter. Three questions, such as "How should it behave?" and "What hue is it?" are asked of each animal once during each simulation round. Which string best describes this animal? Three methods in each Critter class supply these three pieces of information, and it is the programmer's job to override these methods and implement the proper behavior.

I must programme the behavior by overriding each of the methods in each of the Critter classes because each class will contain the code for these behaviours. The colour of the Critter is chosen by the simulator. I have to specify a colour using the colour enter and colour dot codes. There ought to be an equal likelihood that each choice will be made at random. Therefore, I can use Random object or the Math.random() method or an object.random(). ToString will be discussed as the following method. This gives you back a text string with instructions for how to display your Critter in the simulator. The method that responds to the behaviours in the end is getMove.

Only four objects at most can be returned by this process. action.Right, get to work.Action on the left.Up, and take action.Infect. Investigate the Critters' surroundings and environs using other approaches to decide which direction the method should return. Due to the Critter's numerous actions, there may come a moment when one of the variables needs to be changed. A move counter should be added to getMove so that alternative methods can be applied when necessary. I've verified that everything is working perfectly as I've finished each class.

Testing is important in this situation. To isolate the Critter I was working on in the frame, I commented out the frame in the simulator.Add the lines into the Critter main file. This will enable me to separate the Critter and gradually check the operation of my code. The simulator has a diagnostic mode that shows your Critter as an arrow pointing in the direction it is staring for directions.

4.0 DESIGN

The programming language JAVA must be used to create the Animal Kingdom Game. For my simulation application, I'm using Eclipse IDE for Java Developers Version: 2022-12 (4.26.0). The entire structure of the project is built using classes. To invoke the main form of the interface, CritterMain.java must be compiled using the JAVA 18 SDK and run as the main programme.

The building blocks of an object are specified using classes in Java. In the Animal Kingdom application, several animal species are represented by classes like Giant, NinjaCat, Tiger, and WhiteTiger. Each class describes a specific animal species' or activity's features, such as size, color, and speed. (for example, motion or interaction).

In this game I have used Objects, Inheritance, Polymorphism and Encapsulation concepts.

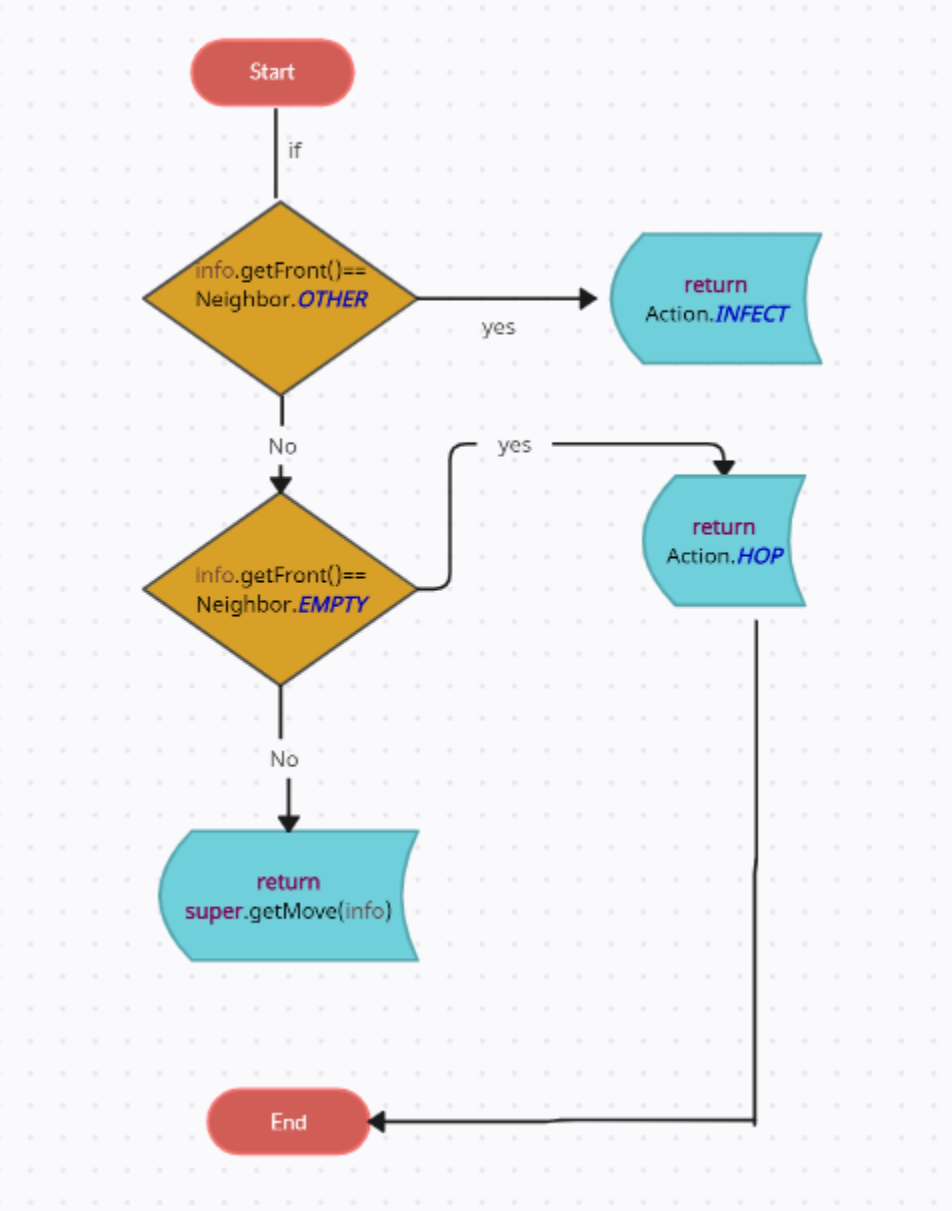
Making effective use of these Java concepts will enable the Animal Kingdom programme to depict the characteristics and behaviours of animals in a structured and organized manner. Objects represent individual animals with their unique characteristics and behaviors, inheritance establishes hierarchical relationships, polymorphism allows for the unified handling of multiple animal types, encapsulation protects data integrity. These elements combine to illustrate various animal types or behaviours. All of these Java tenets work together to provide the programme's correct depiction of the animal realm and enable the modelling of a variety of animal behaviours and features.

5.0 WORKFLOW AND LOGIC

There are 5 animal classes that are written by myself. I've used one of the main constructors getMove from 5 animal classes to demonstrate the flowchart sequence as follows:

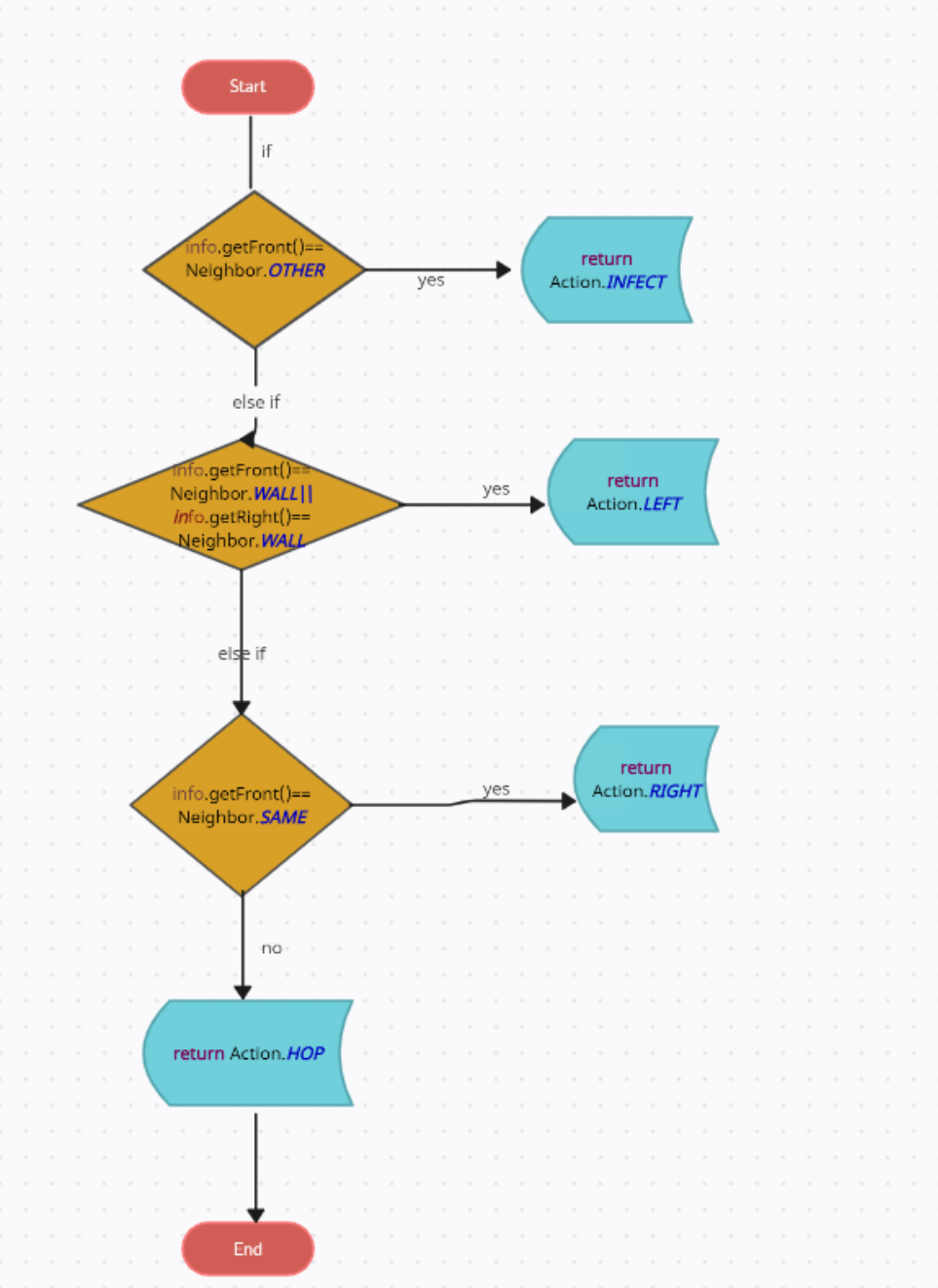
Bear

getMove()



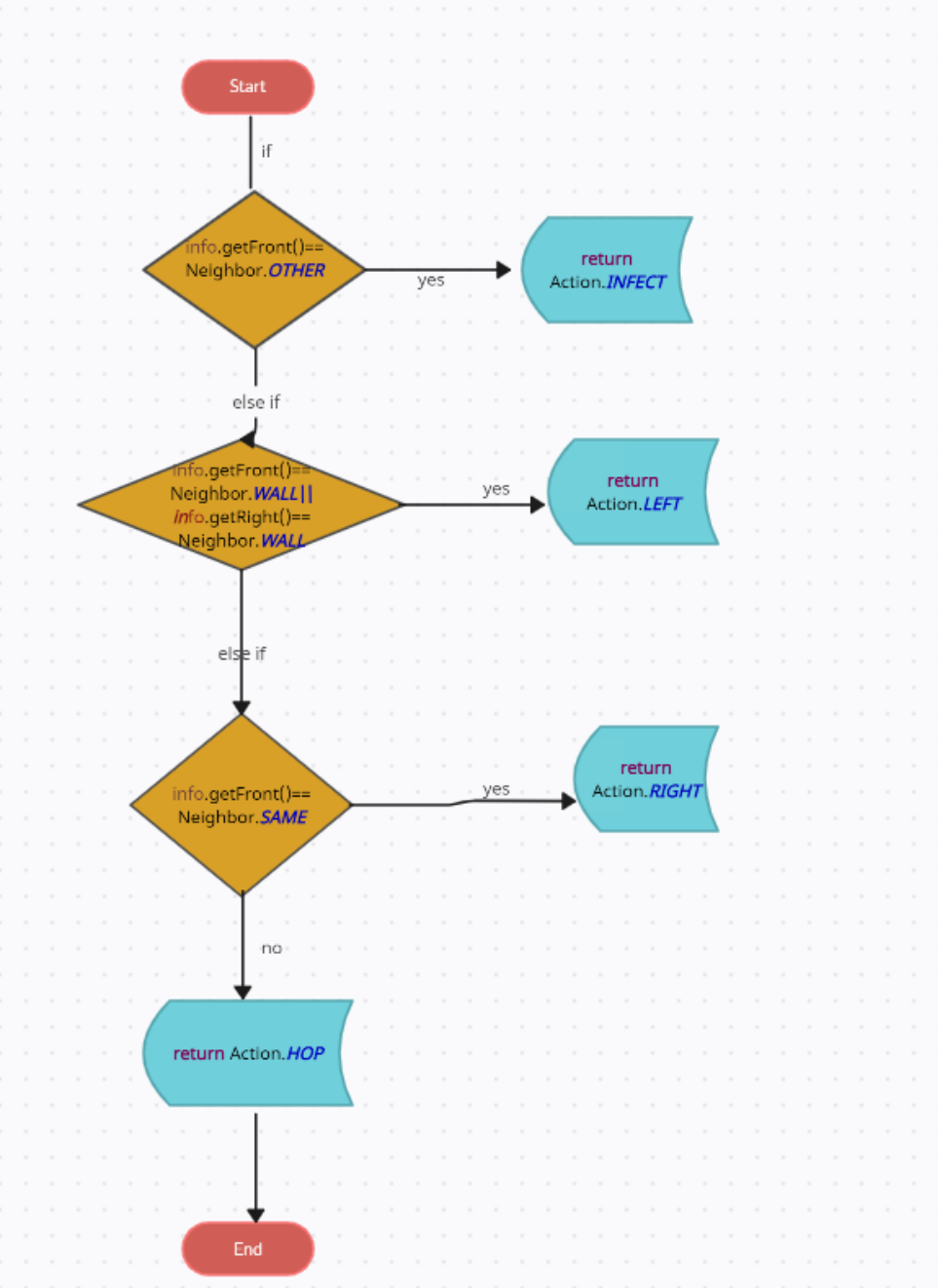
Tiger

getMove()



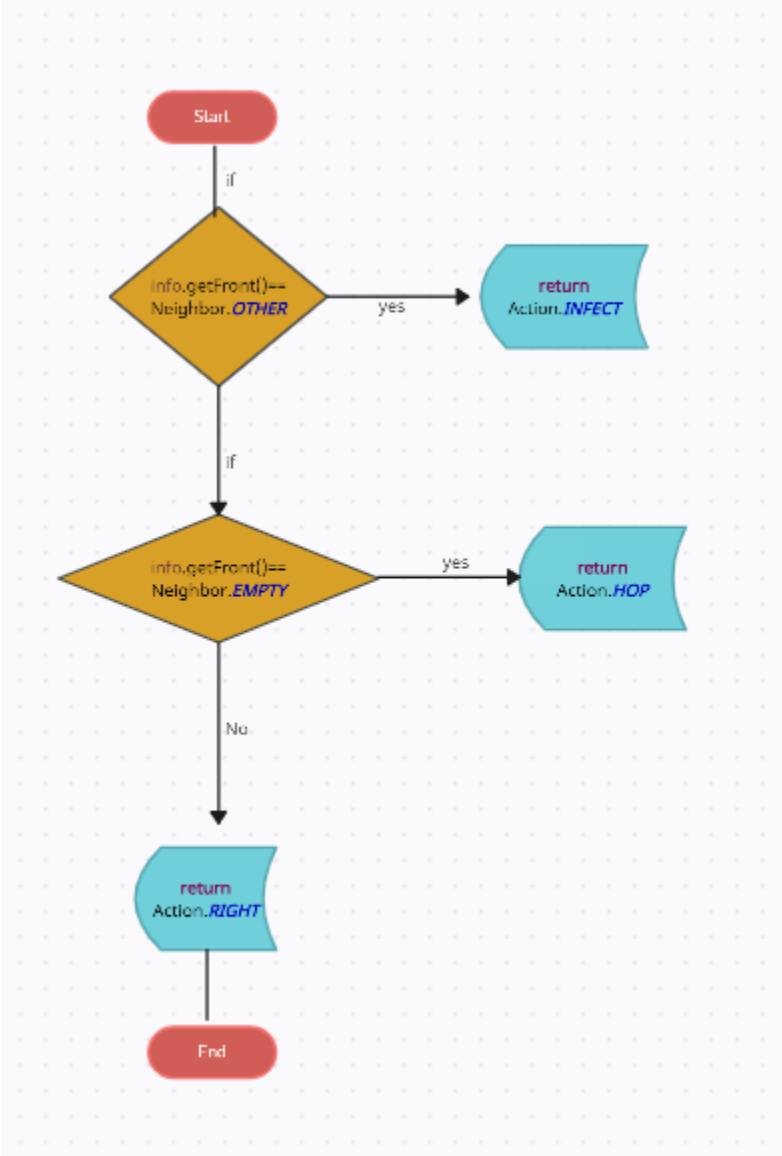
WhiteTiger

getMove()



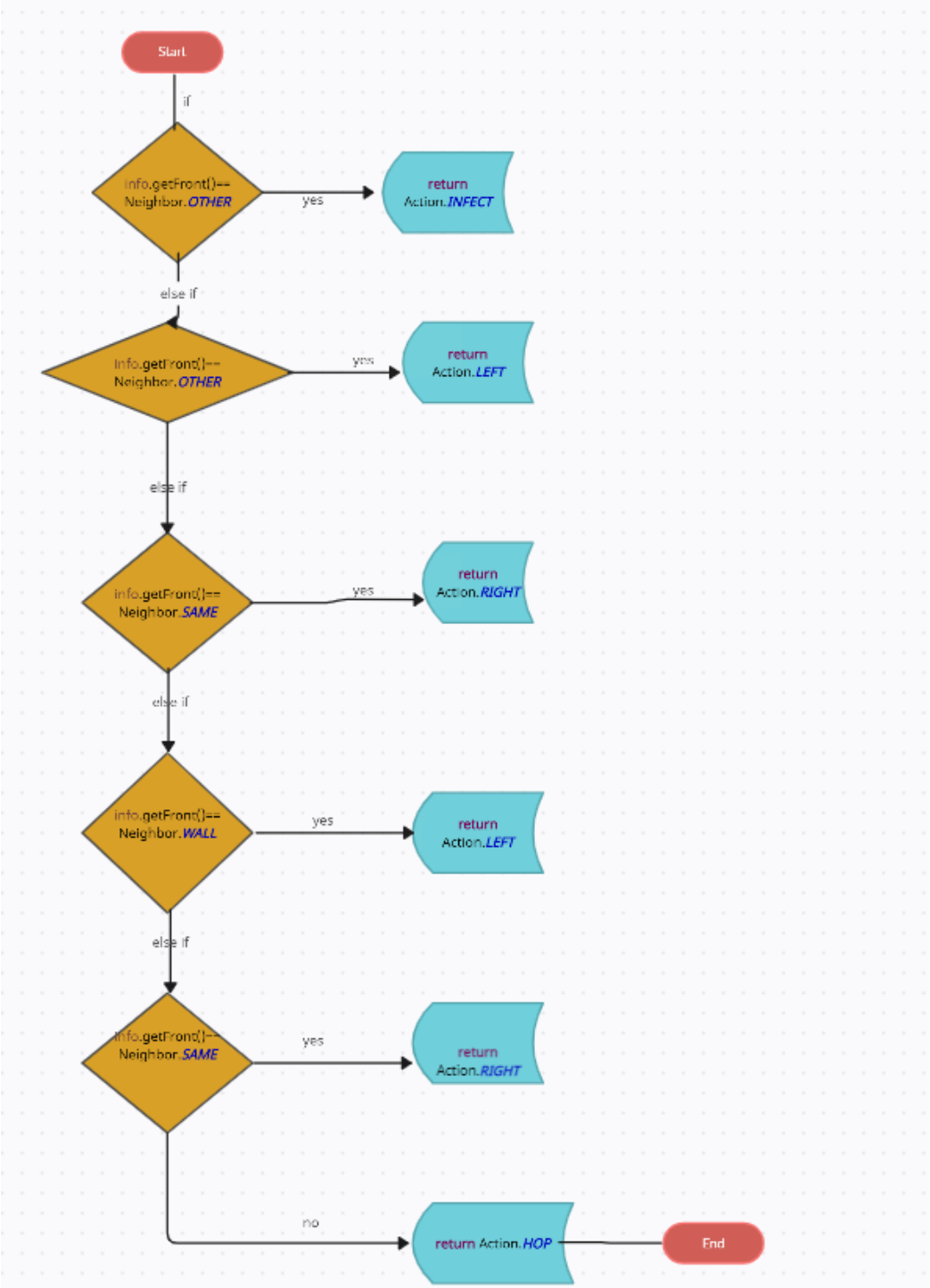
Giant

getMove()



NinjaCat

getMove()



6.0 DEVELOPMENT

A total of 5 classes were given to build the source code.

I have published the code in https://github.com/LS1719/Animal-Kingdom-Simulator.git

Bear.java

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.

```java
// * Final project | ITWM 5113 - Software Design and Development (Year 2023)
// * Author: Leela Shanti (MCC220919180)

import java.awt.*;

public class Bear extends Critter {
    private boolean polar;
    private int moves;

    public Bear(boolean polar){
        this.polar=polar;
        getColor();
    }

    public Color getColor() {
        //Color.WHITE for a polar bear (when polar is true),
        // Color.BLACK otherwise (when polar is false)
        if (this.polar){
            return Color.WHITE;
        } else {
            return Color.BLACK;
        }
    }

    public String toString(){
        //Should alternate on each different move between a slash character (/)
        // and a backslash character () starting with a slash.
        if (moves%2==0){
            return "/";
        } else {
            return "\\";
        }

    }

    public Action getMove(CritterInfo info){
        //always infect if an enemy is in front, otherwise hop if possible, otherwise turn left.
        moves++;
        if(info.getFront()==Neighbor.OTHER){
            return Action.INFECT;
        } else if (info.getFront()==Neighbor.EMPTY){
            return Action.HOP;
        } else {
            return super.getMove(info);
        }
    }

}

```

Tiger.java

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.

Tiger.java

```
// * Final project | ITWM 5113 - Software Design and Development (Year 2023)
// * Author: Leela Shanti (MCC220919180)

import java.awt.*;

public class Tiger extends Critter {
    private int colorMoves;
    Color tigerColor;
    Random rand = new Random();

    public Tiger(){
        colorMoves=0;//1,2,3
        getColor();
    }

    public Color 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.
        if (colorMoves%3==0){ // set new color
            int x=0;
            while (x==0){
                int i=rand.nextInt(3); //0.Red 1.Green 2.Black
                if (i==0 && this.tigerColor!=Color.RED){
                    this.tigerColor= Color.RED;
                    x++;
                } if (i==1 && tigerColor!=Color.GREEN){
                    this.tigerColor=Color.GREEN;
                    x++;
                } if (i==2 && tigerColor!=Color.BLUE){
                    this.tigerColor=Color.BLUE;
                    x++;
                }
            }

        }
        return tigerColor;
    }
```

```
41      public String toString() {
42          return "TGR";
43      }
44
45      public Action getMove(CritterInfo info) {
46          //always infect if an enemy is in front,
47          // otherwise if a wall is in front or to the right, then turn left,
48          // otherwise if a fellow Tiger is in front, then turn right, otherwise hop.
49          colorMoves++;
50          if (info.getFront()==Neighbor.OTHER){
51              return Action.INFECT;
52          } else if (info.getFront()==Neighbor.WALL||info.getRight()==Neighbor.WALL){
53              return Action.LEFT;
54          } else if (info.getFront()==Neighbor.SAME){
55              return Action.RIGHT;
56          } else {
57              return Action.HOP;
58          }
59      }
60  }
61
```

WhiteTiger.java

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.java

```
// * Final project | ITWM 5113 - Software Design and Development (Year 2023)
// * Author: Leela Shanti (MCC220919180)

import java.awt.*;

public class WhiteTiger extends Tiger {
    boolean hasInfected;

    public WhiteTiger(){
        hasInfected=false;
    }


    public Color getColor() {
        //Always Color.WHITE.
        return Color.WHITE;
    }


    public String toString() {
        //"tgr" if it hasn't infected another Critter yet, "TGR" if it has infected.
        if (hasInfected){
            return super.toString();
        } else {
            return "tgr";
        }
    }


    public Action getMove(CritterInfo info) {
        //Same as a Tiger.
        // Note: you'll have to override this method to figure out if it has infected another Critter.
        if (info.getFront()==Neighbor.OTHER){
            hasInfected=true;
        }
        return super.getMove(info);

    }
}
```

Giant.java

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.

Giant.java

```
// * Final project | ITWM 5113 - Software Design and Development (Year 2023)
// * Author: Leela Shanti (MCC220919180)

import java.awt.*;

public class Giant extends Critter{
    private int moves;

    public Giant(){
        moves=1;
        getColor();
    }

    public Color getColor (){
        return Color.GRAY;
    }


    public String toString() {
        //"fee" for 6 moves, then "fie" for 6 moves, then "foe" for 6 moves, then "fum" for 6 moves, then repeat.
        if (moves<=6){
            return "fee";
        } else if (moves<=12){
            return "fie";
        } else if (moves<=18){
            return "foe";
        } else {
            return "fum";
        }
    }

    public Action getMove(CritterInfo info) {
        //always infect if an enemy is in front, otherwise hop if possible, otherwise turn right
        //track moves
        if (info.getFront()==Neighbor.OTHER){
            countMoves();
            return Action.INFECT;
        } else if(info.getFront()==Neighbor.EMPTY){
            countMoves();
            return Action.HOP;
        } else {
            countMoves();
            return Action.RIGHT;
        }
    }

    public void countMoves(){
        if (moves==24){
            moves=1;
        } else {
            moves++;
        }
    }
```

NinjaCat.java

Constructor	**public NinjaCat()**
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	"(=^.^=)"
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 NinjaCat is in front, then turn right, otherwise hop.

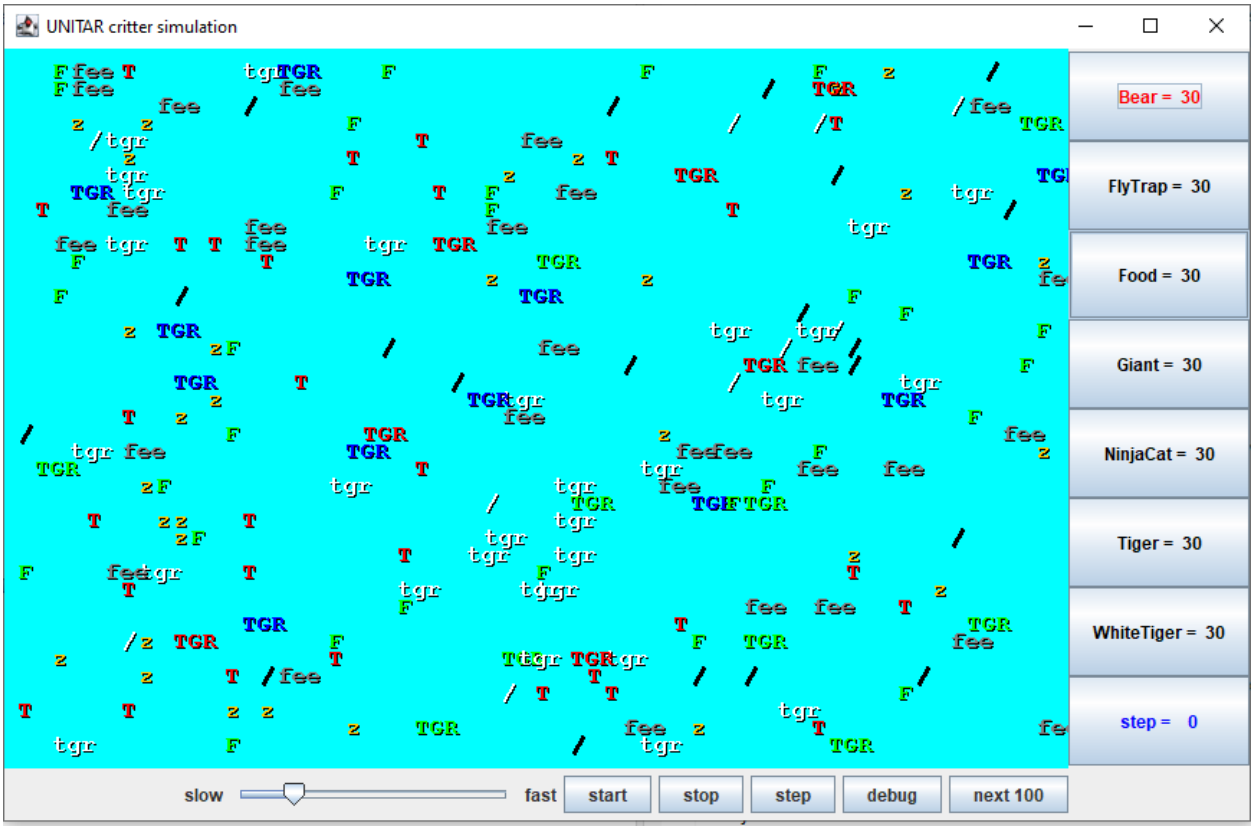
NinjaCat.java

```
// * Final project | ITWM 5113 - Software Design and Development (Year 2023)
// * Author: Leela Shanti (MCC220919180)

import java.awt.*;

public class NinjaCat extends Tiger {

    public boolean hasInfected;

    public NinjaCat (){
        hasInfected=false;
    }

    public Color getColor() {
        if (hasInfected){
            return Color.MAGENTA;
        } else {
            return Color.orange;
        }

    }


    public String toString() {
        if (hasInfected){
            return "Z";
        } else {
            return "z";
        }

    }


    public Action getMove(CritterInfo info) {
        //same as Tiger, but changes color when has infected
        if (info.getFront()==Neighbor.OTHER){
            hasInfected=true;
        }
        return super.getMove(info);

    }
}
```

7.0 RESULTS

Running in simulation mode for the purpose of testing.

All classes have been built and successfully running. The life status is shown on the right-hand side of the dashboard. The layout is as below.

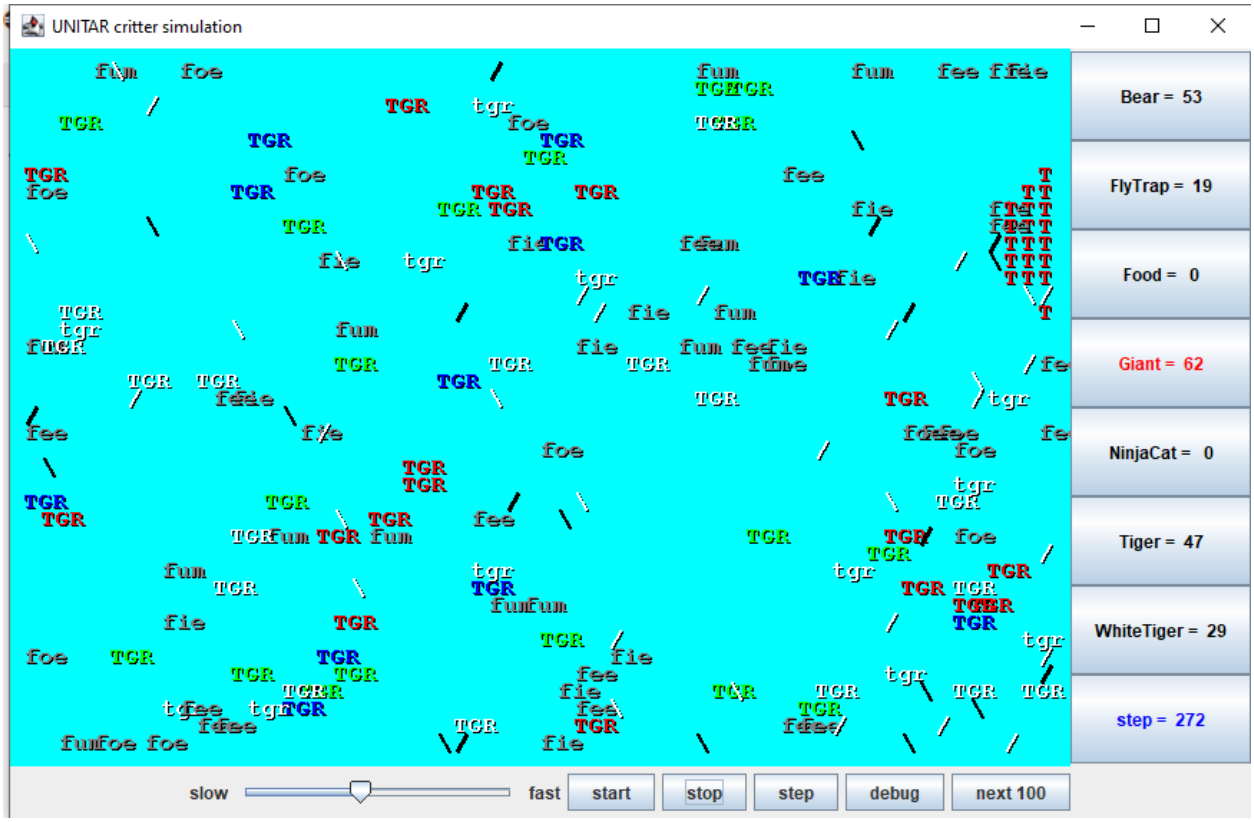


Before start running

TESTING

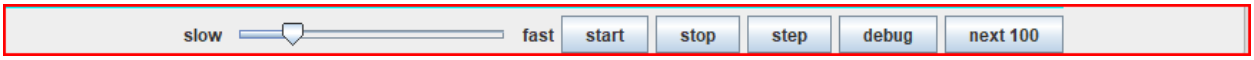
Below are some suggestions for how you can test your critters: • Bear: Try running the simulator with just 30 bears in the world. You should see about half of them being white and about half being black. Initially they should all be displayed with slash characters.

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.

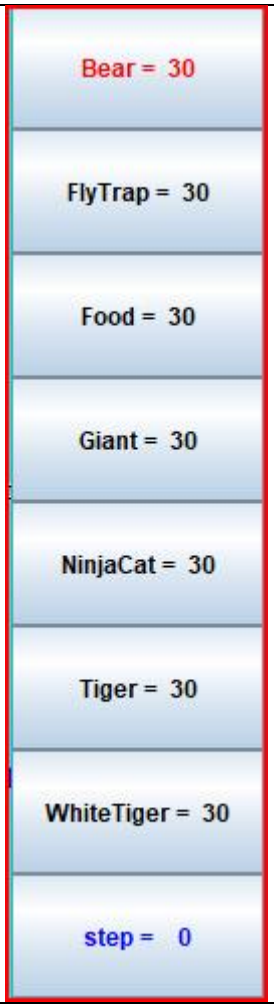


At Running

The game has slow and fast function to run the animal functions slower and faster . Below is the screenshot.



Button	Description of the Layout Button
Speed	Toggle between slow/fast slider
Start	Begin the Game
Stop	Pause/Stop Completely
Debug	Stop game for checking/debugging
Next 100	Display result of next 100 steps

Count of Life Remaining	Description of the Layout Button	
30	Bear	
30	FlyTrap	
30	Food	
30	Giant	
30	NinjaCat	
30	Tiger	
30	White Tiger	
0	Steps	

8.0 DISCUSSION

The Java programme that simulates the animal kingdom utilizing the classes (Critter.java, Critter.Frame.java, CritterInfo, CritterMain.java, CritterModel.java, CritterPanel.java, FlyTrap.java, Food.java, Giant.java, NinjaCat.java, Tiger.java, and WhiteTiger.java) is crucial in terms of the animal kingdom. The final project allows for the creation of new versions in addition to encouraging analytical thinking in the design and development of comprehensive software. By utilizing both private and public methods or classes, the project results in good programming abilities, solid coding standards, adequate documentation, and minimal code.

Here are a few important things that have been discovered, comprehended, and accomplished: -

Using OOP concepts

The nomenclature for inheritance varies slightly between computer languages. Java refers to the parent class as the superclass and the descendant class as the subclass. Superclasses can alternatively be referred to by developers as base, parent, or child classes, and subclasses as derived from them.

The extends keyword is used in the definition of subclasses to relate them to superclasses. Subclasses can create new, local methods or fields to utilise or call inherited methods or the super function Object() { [native code] } using the super keyword.

The Java programming language provides a foundation for simulating and modelling the various characteristics and behaviours of animals. Using Java concepts like classes, objects, inheritance, polymorphism, and encapsulation, the programme enables the depiction of animal behaviours like locomotion, interaction, and reproduction. As a result, we now have a better understanding of how members of the Animal Kingdom might behave in their natural habitats, advancing the study of biology and animal behaviour.

The use of simulations and predictive modelling

In contrast, simulation is model-centric. You begin by building a model of the system in which the issue operates using human knowledge of cause and effect. Then, you use that model to estimate the future using the data you already have. For instance, to forecast future sales, you would model its primary causal components, such as the expertise of the sales personnel, the caliber of the product, other market circumstances, and how they all interact. In other words, the more knowledgeable the people engaged, the more accurate the predictions will be.

Pattern recognition relies on correlation, whereas simulation depends on human understanding of causation, which is the key distinction between the two methods.

Prospects for Future Change

The Java software can be updated and upgraded to support more complex animal behaviors, various animal species, or fresh research questions. It might act as a springboard for the creation of additional simulations, educational materials, or intricate research models. The programme could be enhanced and utilised in a variety of sectors connected to the animal kingdom because of Java's adaptability and extensibility.

In conclusion, the Java programme that mimics the Animal Kingdom has a big influence on assisting us in understanding the range of features and behaviours that animals display. Numerous methods, including simulations, teaching aids, or scientific models, can be used to study animal behaviors, predict results, and provide insights into how the Animal Kingdom functions.

9.0 CONCLUSION

Through the use of Java, this project will promote Object Oriented Programming (OOP). By using Java Classes and Methods and OOP approaches to build the Animal Kingdom Game, the project seeks to advance programming abilities.

The Animal Kingdom is a large and complex set of living beings that are incredibly significant in many academic fields and are essential to ecosystems. Java models of the animal world that take advantage of concepts from object-oriented programming such as classes, objects, inheritance, polymorphism, and encapsulation provide a robust framework for defining the diverse characteristics and behaviours of animals.

We emphasised the richness and diversity of the animal kingdom as well as the benefits of simulating it in Java, such as the ability to capture the various components of animal biology, ecology, and behaviour in a systematic and organised way. The modular and upkeep-friendly object-oriented philosophy of Java, which includes components like classes, objects, inheritance, polymorphism, and encapsulation, provides a strong and flexible framework for defining the varied characteristics and behaviours of animals.

We can develop intricate and precise simulations, teaching tools, or research models that assist us in understanding and appreciating the complexity and diversity of the animal kingdom by exploiting the strengths of Java programming techniques. Java modelling can be used to study the animal kingdom in a variety of fields.

This project has provided an opportunity to work on object oriented programming, Java and understanding how the animal kingdom simulator works.