UNITAR
INTERNATIONAL
UNIVERSITY

FINAL PROJECT REPORT

Prepared by	:	Zakaria bin Ismail
Matric card no	:	MC210413437
Course code	:	ITWM5013
Course tittle	:	Software Design and Development
Tittle of assignment	:	Final Project – Animal Kingdom
Prepared for	:	Dr. Feras Zen Alden

Abstract

Animal Kingdom – is a final project developed to establish the level of understanding, skills and abilities based on knowledge acquired in Java Object Oriented Programming course which employs several functions such as inheritance, interface and polymorphism.

Introduction

Animal Kingdom – Final Project is a programming software developed using java programming language that implements object-oriented development. This project employs the functions of inheritance, interface and polymorphism simultaneously introducing alternative functions available in the Java programming language using an application programming interface (API) to create graphical user interface (GUI) generated via Abstract Window Toolkit as the output display.

Project Description

This final project is developed using several classes where each class needs to be programmed to determine the behaviour of each animal that has been arrange beforehand in which each conducts interacts differently in the program.

Several main and basic program codes are provided to help with this programming such as Critter.java, CritterInfo.java, CritterPanel.java, CritterModel.java, CritterMain.java, CritterFrame.java, Food.java and FlyTrap.java. In order to complete this program, there are five (5) new classes are added which are Bear.java, Tiger.java, WhiteTiger.java, Giant.java and Ninja Cat.java. Each of these classes needs to inherit from Critter.java which define actions via the getMove function, colours via the getColor function and identifiers via the toString function for each of the animals in the program.

Project Design

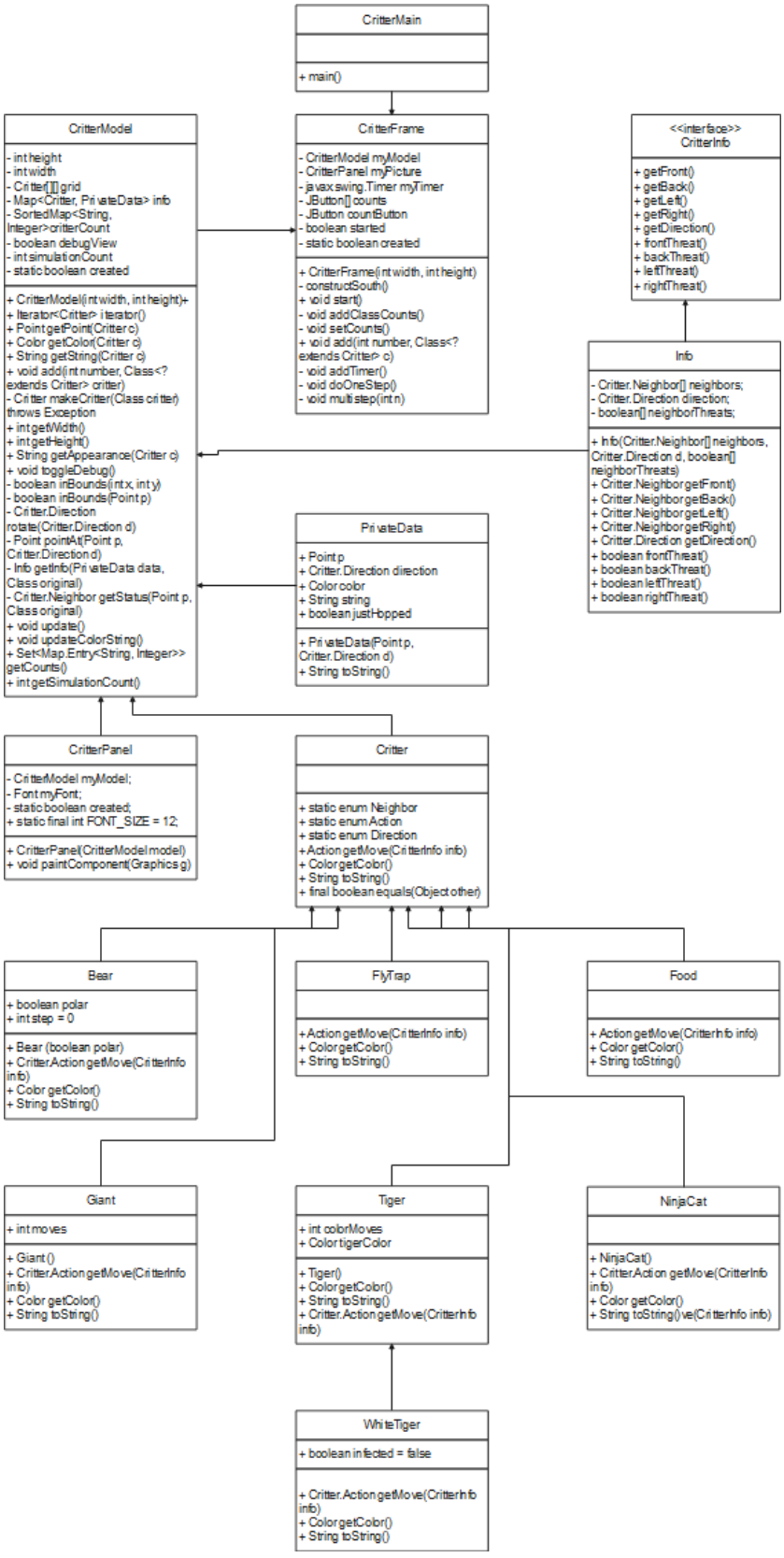

Figure 1: Project Design

This project is designed based on 15 interconnected java classes. The class begins with CritterMain.java that create frames via CritterFrame.java which then will call relevant classes to create frames, panels and animals in Animal Kingdom as shown in Figure 1.

Program workflow and logics

As a start, CritterMain.java that builds the frame will creates several objects such as Bear, Tiger, WhiteTiger, Giant, NinjaCat, FlyTrap and Food.

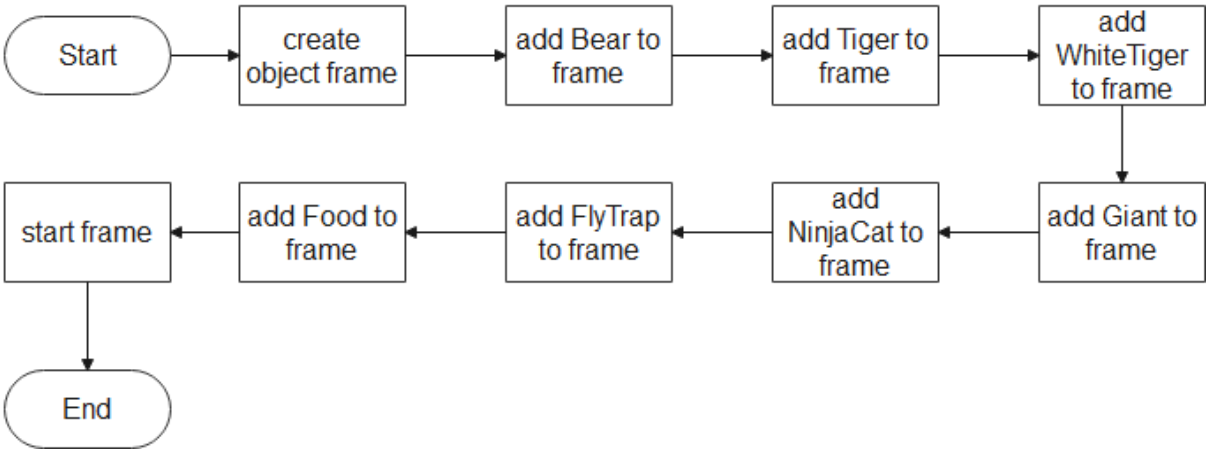

Figure 2: CritterMain.java

Figure 2 shows the workflow and logic in CritterMain.java class. As the main class that runs test on all classes will make sure the program is running efficiently. This class creates a frame for the output display as well as animals, food and flytraps within the frame.

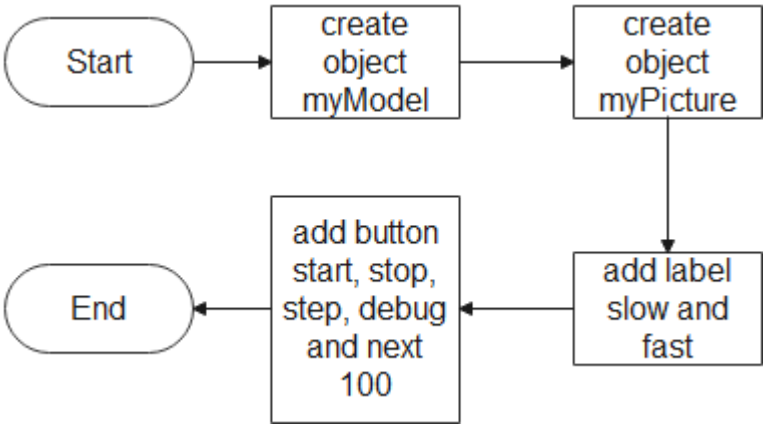

Figure 3: CritterFrame.java

Figure 3 shows the workflow and logic in CritterFrame.java. The class is programmed to create a GUI frame for the output display by creating models, pictures, labels and buttons. This class uses two additional classes, namely CritterModel.java and CritterPanel.java to create the output frame.

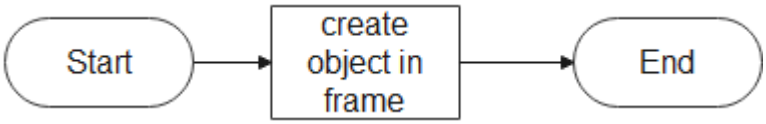

Figure 4: CritterModel.java

Figure 4 shows the workflow and logic in CritterModel.java. This class is use by CritterFrame.java to create object frames.

Figure 5: CritterPanel.java

Figure 5 shows the workflow and logic in CritterPanel.java. This class is use by CritterFrame.java to generate GUI for output display.

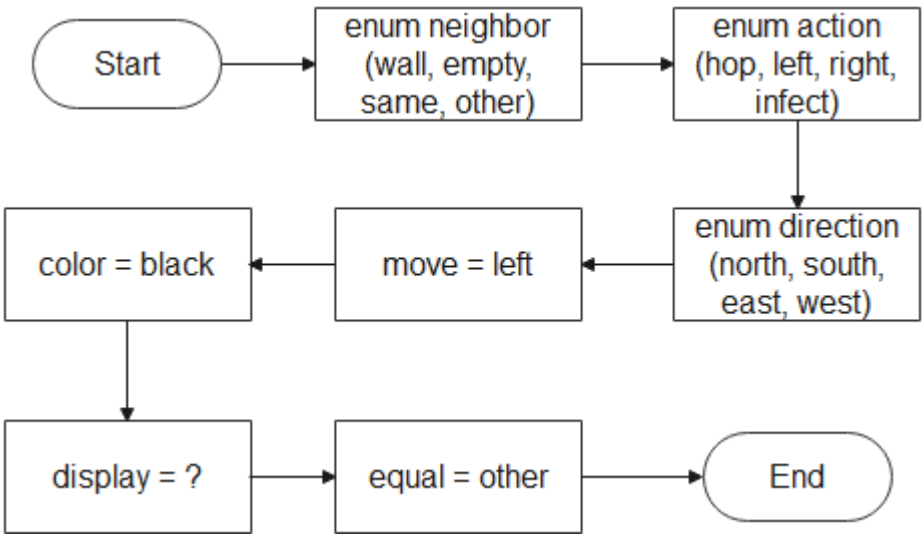

Figure 6: Critter.java

Figure 6 shows the workflow and logic in Critter.java. This superclass is inherited by Bear.java, FlyTrap.java, Food.java, Giant.java, NinjaCat.java and Tiger.java. This class has an enumeration for neighbour and action which correspond in the get animal move action function other than three complementary functions such as get animal color, display the animal and Boolean in which it is employed to differentiate between distinct animals species.

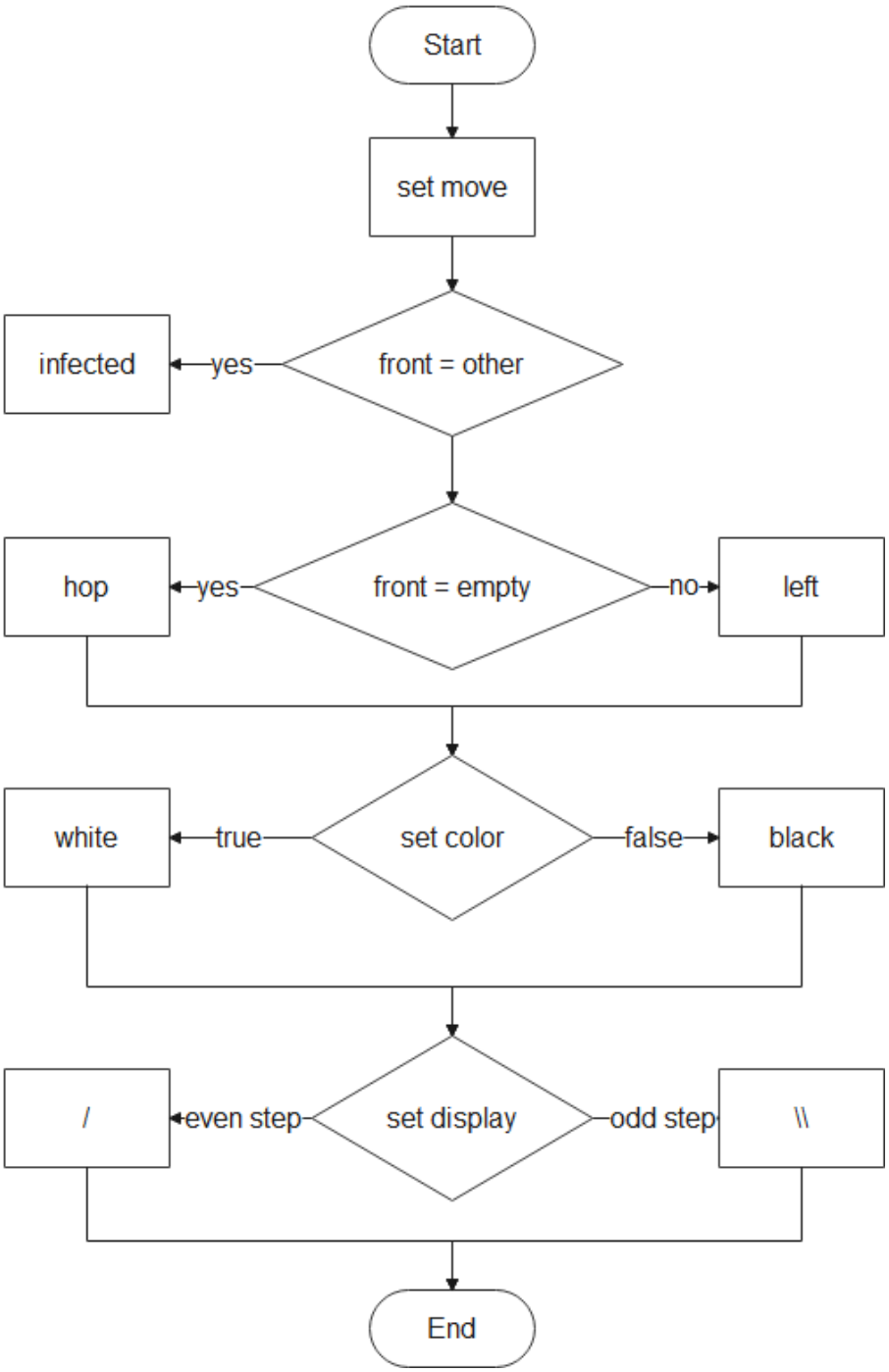

Figure 7: Bear.java

Figure 7 shows the workflow and logic in Bear.java that inherits from Critter.java. Bears are marked as infected if there is unspecified object in ahead. The bear has options to hop if there is nothing in front or move to the left if otherwise. White bear is generated if the program is correct and black bear if program is incorrect. For displays representing Bears, each even steps is marked as / while \\ is marked as odd steps.

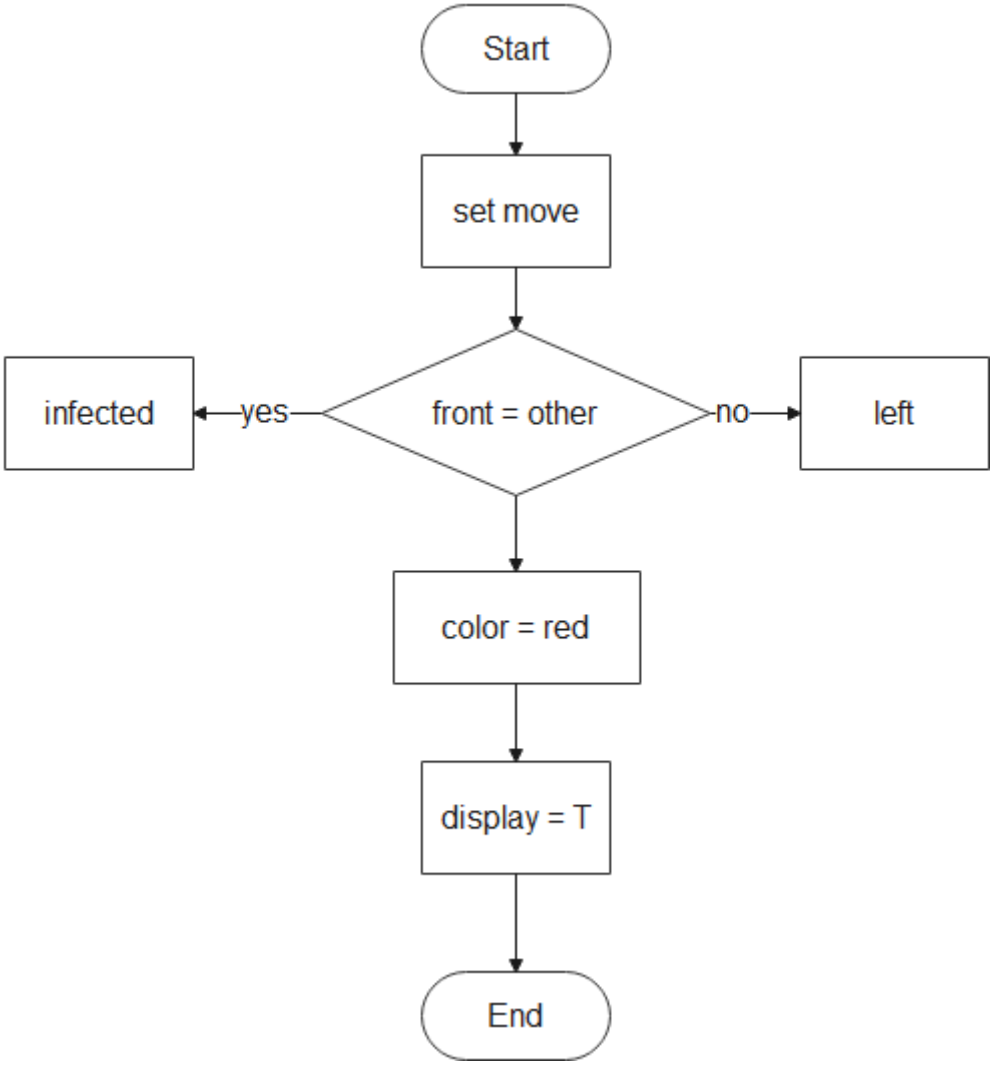

Figure 8: FlyTrap.java

Figure 8 shows the workflow and logic in FlyTrap.java inherited from Critter.java. FlyTrap is marked as infected if there is unspecified object ahead or move to the left if otherwise. FlyTrap is generated as red and displayed as a T.

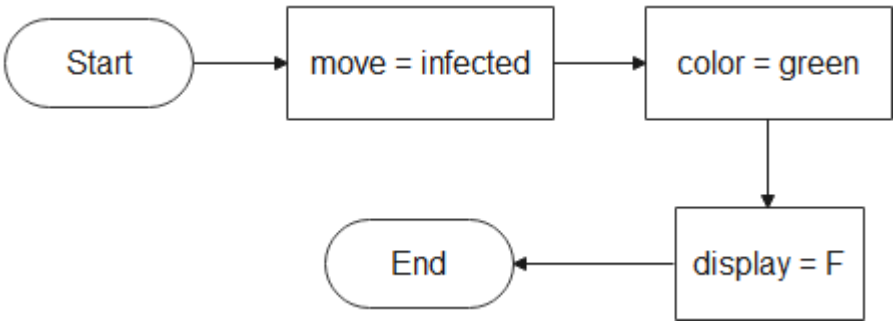

Figure 9: Food.java

Figure 9 shows the workflow and logic in Food.java inherited from Critter.java. Food is marked as infected, and generated green as well displayed as F.

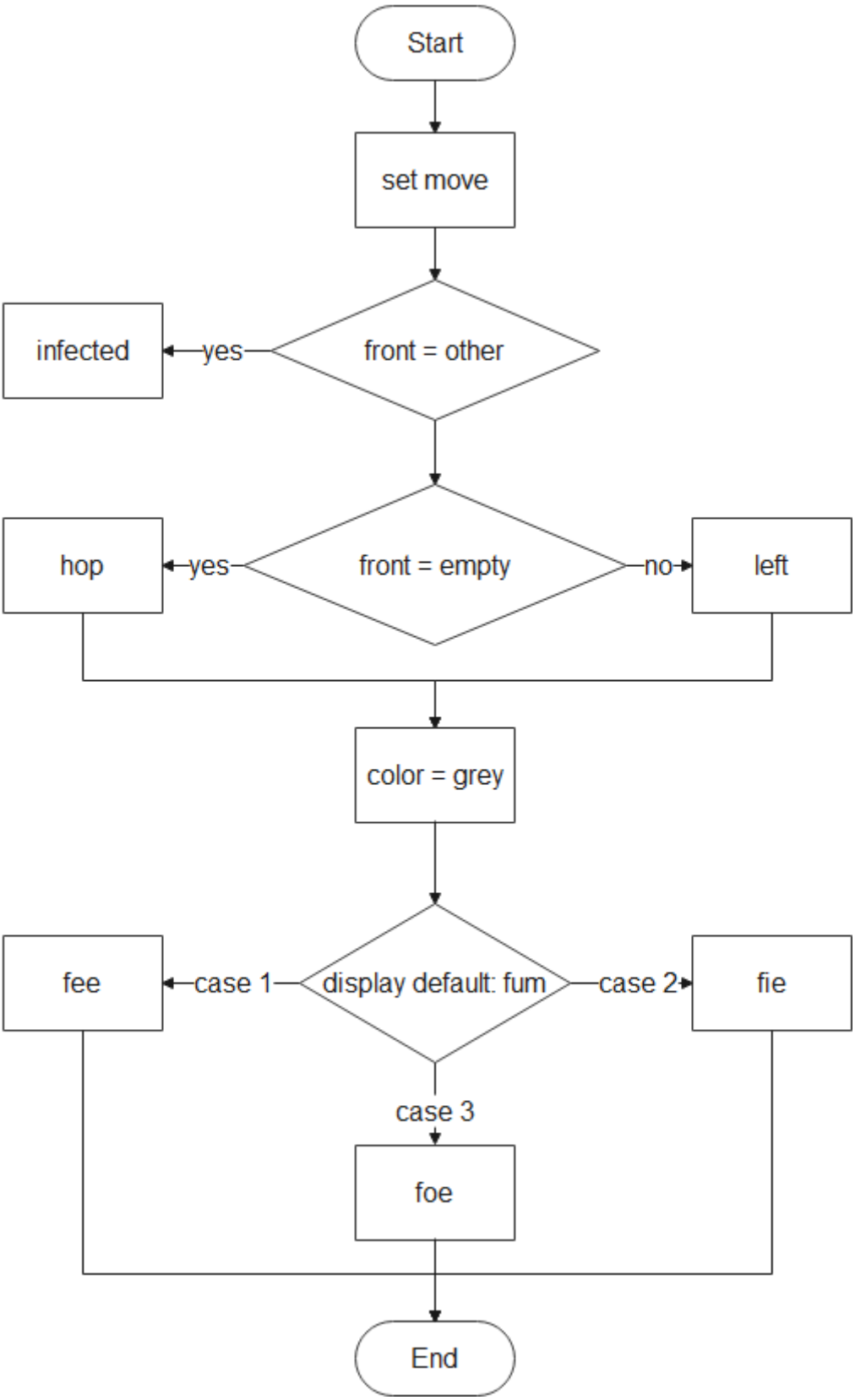

Figure 10: Giant.java

Figure 10 shows the workflow and in Giant.java that inherits from Critter.java. Giant is marked as infected if there is unspecified object ahead. Giant has options to hop if there is nothing in front or move to the left if otherwise. Giant is generated as grey and displayed as fum and changes to fee in the first six steps, subsequently changes to fie in the next six steps and foe for the third six steps. Displays are repeated in every six step changes.

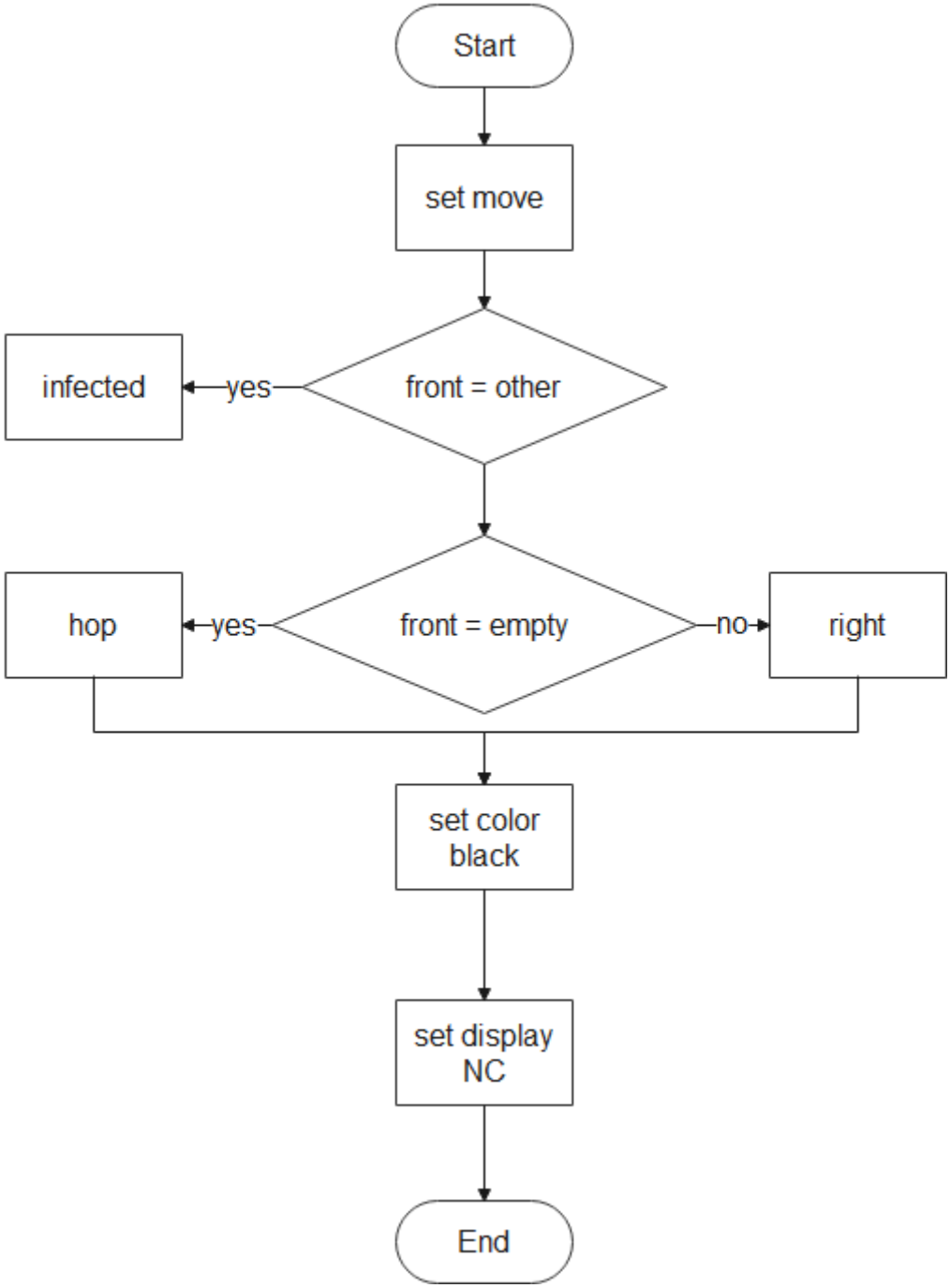

Figure 11: NinjaCat.java

Figure 11 shows the workflow and found in NinjaCat.java that inherited from Critter.java. NinjaCat is marked as infected if there is unspecified object ahead. NinjaCat has option to hop if there is nothing in front or move to the right if otherwise and generated as black and displayed as NC.

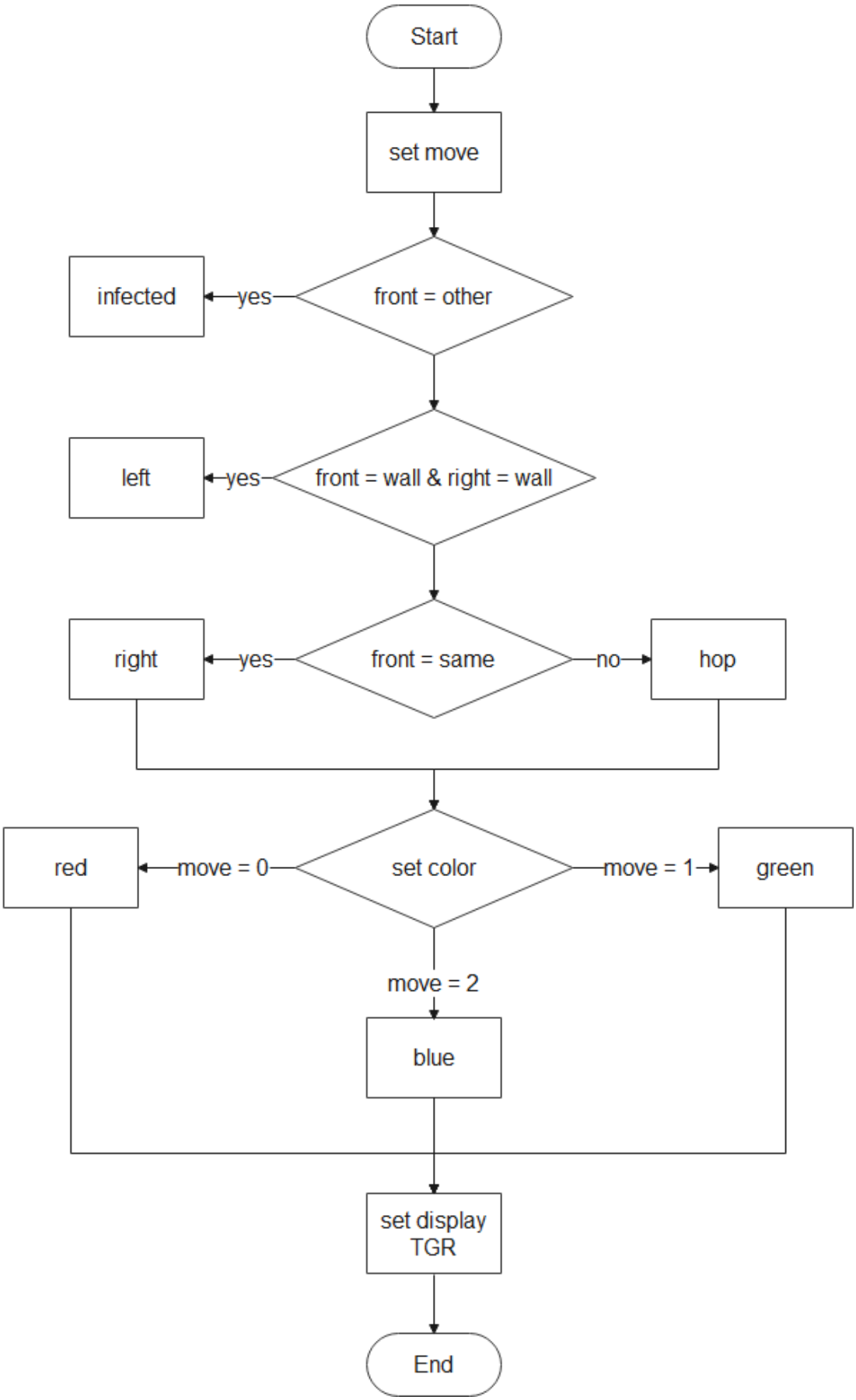

Figure 12: Tiger.java

Figure 12 shows the workflow and logic in Tiger.java that inherits from Critter.java. Tiger is marked as infected if there is unspecified object ahead. Tiger has option to move to the left if there is a wall in front or move to the right if there is another Tiger in front. Otherwise, Tiger may hop to the right. Random colours are generated for Tiger such as red, green or blue and may switch to choose new random colours after moving three steps. Tiger is displayed as TGR.

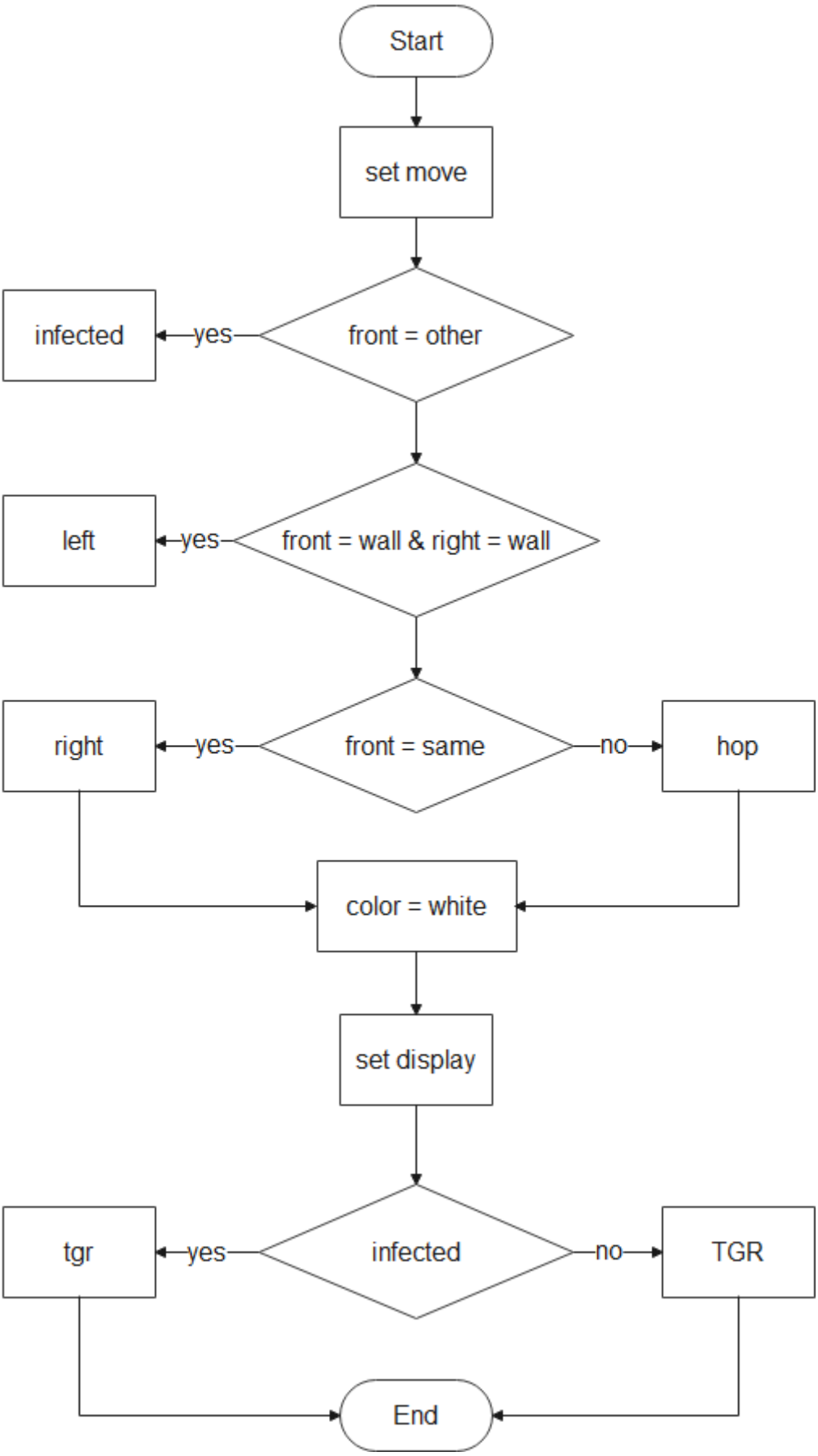

Figure 13: WhiteTiger.java

Figure 12 shows the workflow and logic in WhiteTiger.java that inherited from Tiger.java. WhiteTiger is marked as infected if there is unspecified object ahead. WhiteTiger has option to move to the left action if there is a wall in front or to the right. If there is another WhiteTiger in front, the WhiteTiger has the option to move to the right or hop to the right if otherwise. WhiteTiger is generated as white and displayed as tgr if infected and TGR if otherwise.

Development details

This program is developed using 13 java classes as follows:

```
// CSE 142 Homework 9 (Critters)
// Authors: Stuart Reges and Marty Stepp
//          modified by Kyle Thayer
//
// CritterMain provides the main method for a simple simulation program.  Alter
// the number of each critter added to the simulation if you want to experiment
// with different scenarios.  You can also alter the width and height passed to
// the CritterFrame constructor.

public class CritterMain {
    public static void main(String[] args) {
        CritterFrame frame = new CritterFrame( width: 60,  height: 40);

        // uncomment each of these lines as you complete these classes
        frame.add( number: 30, Bear.class);
        frame.add( number: 30, Tiger.class);
        frame.add( number: 30, WhiteTiger.class);
        frame.add( number: 30, Giant.class);
        frame.add( number: 30, NinjaCat.class);

        frame.add( number: 30, FlyTrap.class);
        frame.add( number: 30, Food.class);

        frame.start();

    }
}
```

Figure 14: CritterMain.java

Figure 14 shows the CritterMain.java program as the main program use to run this project. The program will create a frame and subsequently adds other objects such as animals, food and traps into the animal kingdom frame.

```java
// Class CritterFrame provides the user interface for a simple simulation
// program.

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.util.*;

public class CritterFrame extends JFrame {
    private CritterModel myModel;
    private CritterPanel myPicture;
    private javax.swing.Timer myTimer;
    private JButton[] counts;
    private JButton countButton;
    private boolean started;
    private static boolean created;

    public CritterFrame(int width, int height) {
        // this prevents someone from trying to create their own copy of
        // the GUI components
        if (created)
            throw new RuntimeException("Only one world allowed");
        created = true;

        // create frame and model
        setTitle("CSE142 critter simulation");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        myModel = new CritterModel(width, height);
```

```
        // set up critter picture panel
        myPicture = new CritterPanel(myModel);
        add(myPicture, BorderLayout.CENTER);

        addTimer();

        constructSouth();

        // initially it has not started
        started = false;
    }

    // construct the controls and label for the southern panel
    private void constructSouth() {
        // add timer controls to the south
        JPanel p = new JPanel();

        final JSlider slider = new JSlider();
        slider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                double ratio = 1000.0 / (1 + Math.pow(slider.getValue(), 0.3));
                myTimer.setDelay((int) (ratio - 180));
            }
        });
        slider.setValue(20);
        p.add(new JLabel( text: "slow"));
        p.add(slider);
        p.add(new JLabel( text: "fast"));
```

```
        JButton b1 = new JButton( text: "start");
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                myTimer.start();
            }
        });
        p.add(b1);
        JButton b2 = new JButton( text: "stop");
        b2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                myTimer.stop();
            }
        });
        p.add(b2);
        JButton b3 = new JButton( text: "step");
        b3.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                doOneStep();
            }
        });
        p.add(b3);

        // add debug button
        JButton b4 = new JButton( text: "debug");
        b4.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                myModel.toggleDebug();
                myPicture.repaint();
            }
        });
        p.add(b4);
```

```
        // add 100 button
        JButton b5 = new JButton( text: "next 100");
        b5.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                multistep( n: 100);
            }
        });
        p.add(b5);

        add(p, BorderLayout.SOUTH);
    }

    // starts the simulation...assumes all critters have already been added
    public void start() {
        // don't let anyone start a second time and remember if we have started
        if (started) {
            return;
        }
        // if they didn't add any critters, then nothing to do
        if (myModel.getCounts().isEmpty()) {
            System.out.println("nothing to simulate--no critters");
            return;
        }
        started = true;
        addClassCounts();
        myModel.updateColorString();
        pack();
        setVisible(true);
    }
```

```java
// add right-hand column showing how many of each critter are alive
private void addClassCounts() {
    Set<Map.Entry<String, Integer>> entries = myModel.getCounts();
    JPanel p = new JPanel(new GridLayout( rows: entries.size() + 1,  cols: 1));
    counts = new JButton[entries.size()];
    for (int i = 0; i < counts.length; i++) {
        counts[i] = new JButton();
        p.add(counts[i]);
    }

    // add simulation count
    countButton = new JButton();
    countButton.setForeground(Color.BLUE);
    p.add(countButton);

    add(p, BorderLayout.EAST);
    setCounts();
}
```

```java
private void setCounts() {
    Set<Map.Entry<String, Integer>> entries = myModel.getCounts();
    int i = 0;
    int max = 0;
    int maxI = 0;
    for (Map.Entry<String, Integer> entry: myModel.getCounts()) {
        String s = String.format("%s =%4d", entry.getKey(),
                                 (int) entry.getValue());
        counts[i].setText(s);
        counts[i].setForeground(Color.BLACK);
        if (entry.getValue() > max) {
            max = entry.getValue();
            maxI = i;
        }
        i++;
    }
    counts[maxI].setForeground(Color.RED);
    String s = String.format("step =%5d", myModel.getSimulationCount());
    countButton.setText(s);
}
```

```java
// add a certain number of critters of a particular class to the simulation
public void add(int number, Class<? extends Critter> c) {
    // don't let anyone add critters after simulation starts
    if (started) {
        return;
    }
    // temporarily turning on started flag prevents critter constructors
    // from calling add
    started = true;
    myModel.add(number, c);
    started = false;
}

// post: creates a timer that calls the model's update
//       method and repaints the display
private void addTimer() {
    ActionListener updater = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            doOneStep();
        }
    };
    myTimer = new javax.swing.Timer( delay: 0, updater);
    myTimer.setCoalesce(true);
}

// one step of the simulation
private void doOneStep() {
    myModel.update();
    setCounts();
    myPicture.repaint();
}
```

```java
    // advance the simulation until step % n is 0
    private void multistep(int n) {
        myTimer.stop();
        do {
            myModel.update();
        } while (myModel.getSimulationCount() % n != 0);
        setCounts();
        myPicture.repaint();
    }
}
```

Figure 15: CritterFrame.java

Figure 15 shows the CritterFrame.java class used to create frames that inherit from JFrame and by using CritterModel and CritterPanel classes.

```java
// Class CritterModel keeps track of the state of the critter simulation.

import java.util.*;
import java.awt.Point;
import java.awt.Color;
import java.lang.reflect.*;

public class CritterModel {
    // the following constant indicates how often infect should fail for
    // critters who didn't hop on their previous move (0.0 means no advantage,
    // 1.0 means 100% advantage)
    public static final double HOP_ADVANTAGE = 0.2; // 20% advantage

    private int height;
    private int width;
    private Critter[][] grid;
    private Map<Critter, PrivateData> info;
    private SortedMap<String, Integer>critterCount;
    private boolean debugView;
    private int simulationCount;
    private static boolean created;
```

```java
public CritterModel(int width, int height) {
    // this prevents someone from trying to create their own copy of
    // the GUI components
    if (created)
        throw new RuntimeException("Only one world allowed");
    created = true;

    this.width = width;
    this.height = height;
    grid = new Critter[width][height];
    info = new HashMap<Critter, PrivateData>();
    critterCount = new TreeMap<String, Integer>();
    this.debugView = false;
}

public Iterator<Critter> iterator() {
    return info.keySet().iterator();
}

public Point getPoint(Critter c) {
    return info.get(c).p;
}

public Color getColor(Critter c) {
    return info.get(c).color;
}

public String getString(Critter c) {
    return info.get(c).string;
}
```

```java
public void add(int number, Class<? extends Critter> critter) {
    Random r = new Random();
    Critter.Direction[] directions = Critter.Direction.values();
    if (info.size() + number > width * height)
        throw new RuntimeException("adding too many critters");
    for (int i = 0; i < number; i++) {
        Critter next;
        try {
            next = makeCritter(critter);
        } catch (IllegalArgumentException e) {
            System.out.println("ERROR: " + critter + " does not have" +
                               " the appropriate constructor.");
            System.exit( status: 1);
            return;
        } catch (Exception e) {
            System.out.println("ERROR: " + critter + " threw an " +
                               " exception in its constructor.");
            System.exit( status: 1);
            return;
        }
        int x, y;
        do {
            x = r.nextInt(width);
            y = r.nextInt(height);
        } while (grid[x][y] != null);
        grid[x][y] = next;

        Critter.Direction d = directions[r.nextInt(directions.length)];
        info.put(next, new PrivateData(new Point(x, y), d));
    }
```

```
        String name = critter.getName();
        if (!critterCount.containsKey(name))
            critterCount.put(name, number);
        else
            critterCount.put(name, critterCount.get(name) + number);
    }

    /unchecked/
    private Critter makeCritter(Class critter) throws Exception {
        Constructor c = critter.getConstructors()[0];
        if (critter.toString().equals("class Bear")) {
            // flip a coin
            boolean b = Math.random() < 0.5;
            return (Critter) c.newInstance(new Object[] {b});
        } else {
            return (Critter) c.newInstance();
        }
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }
```

```java
public String getAppearance(Critter c) {
    // Override specified toString if debug flag is true
    if (!debugView)
        return info.get(c).string;
    else {
        PrivateData data = info.get(c);
        if (data.direction == Critter.Direction.NORTH) return "^";
        else if (data.direction == Critter.Direction.SOUTH) return "v";
        else if (data.direction == Critter.Direction.EAST) return ">";
        else return "<";
    }
}

public void toggleDebug() {
    this.debugView = !this.debugView;
}

private boolean inBounds(int x, int y) {
    return (x >= 0 && x < width && y >= 0 && y < height);
}

private boolean inBounds(Point p) {
    return inBounds(p.x, p.y);
}

// returns the result of rotating the given direction clockwise
private Critter.Direction rotate(Critter.Direction d) {
    if (d == Critter.Direction.NORTH) return Critter.Direction.EAST;
    else if (d == Critter.Direction.SOUTH) return Critter.Direction.WEST;
    else if (d == Critter.Direction.EAST) return Critter.Direction.SOUTH;
    else return Critter.Direction.NORTH;
}
```

```
    private Point pointAt(Point p, Critter.Direction d) {
        if (d == Critter.Direction.NORTH) return new Point(p.x,  y: p.y - 1);
        else if (d == Critter.Direction.SOUTH) return new Point(p.x,  y: p.y + 1);
        else if (d == Critter.Direction.EAST) return new Point( x: p.x + 1, p.y);
        else return new Point( x: p.x - 1, p.y);
    }

    private Info getInfo(PrivateData data, Class original) {
        Critter.Neighbor[] neighbors = new Critter.Neighbor[4];
        Critter.Direction d = data.direction;
        boolean[] neighborThreats = new boolean[4];
        for (int i = 0; i < 4; i++) {
            neighbors[i] = getStatus(pointAt(data.p, d), original);
            if (neighbors[i] == Critter.Neighbor.OTHER) {
                Point p = pointAt(data.p, d);
                PrivateData oldData = info.get(grid[p.x][p.y]);
                neighborThreats[i] = d == rotate(rotate(oldData.direction));
            }
            d = rotate(d);
        }
        return new Info(neighbors, data.direction, neighborThreats);
    }

    private Critter.Neighbor getStatus(Point p, Class original) {
        if (!inBounds(p))
            return Critter.Neighbor.WALL;
        else if (grid[p.x][p.y] == null)
            return Critter.Neighbor.EMPTY;
        else if (grid[p.x][p.y].getClass() == original)
            return Critter.Neighbor.SAME;
        else
            return Critter.Neighbor.OTHER;
    }
```

```
/unchecked/
public void update() {
    simulationCount++;
    Object[] list = info.keySet().toArray();
    Collections.shuffle(Arrays.asList(list));

    // This keeps track of critters that are locked and cannot be
    // infected this turn. The happens when:
    // * a Critter is infected
    // * a Critter hops
    Set<Critter> locked = new HashSet<~>();

    for (int i = 0; i < list.length; i++) {
        Critter next = (Critter)list[i];
        PrivateData data = info.get(next);
        if (data == null) {
            // happens when creature was infected earlier in this round
            continue;
        }
        // clear any prior setting for having hopped in the past
        boolean hadHopped = data.justHopped;
        data.justHopped = false;
        Point p = data.p;
        Point p2 = pointAt(p, data.direction);
```

```
        // try to perform the critter's action
        Critter.Action move = next.getMove(getInfo(data, next.getClass()));
        if (move == Critter.Action.LEFT)
            data.direction = rotate(rotate(rotate(data.direction)));
        else if (move == Critter.Action.RIGHT)
            data.direction = rotate(data.direction);
        else if (move == Critter.Action.HOP) {
            if (inBounds(p2) && grid[p2.x][p2.y] == null) {
                grid[p2.x][p2.y] = grid[p.x][p.y];
                grid[p.x][p.y] = null;
                data.p = p2;
                locked.add(next); //successful hop locks a critter from
                                  // being infected for the rest of the
                                  // turn
                data.justHopped = true;  // remember a successful hop
            }
        } else if (move == Critter.Action.INFECT) {
            if (inBounds(p2) && grid[p2.x][p2.y] != null
                && grid[p2.x][p2.y].getClass() != next.getClass()
                && !locked.contains(grid[p2.x][p2.y])
                && (hadHopped || Math.random() >= HOP_ADVANTAGE)) {
                Critter other = grid[p2.x][p2.y];
                // remember the old critter's private data
                PrivateData oldData = info.get(other);
                // then remove that old critter
                String c1 = other.getClass().getName();
                critterCount.put(c1, critterCount.get(c1) - 1);
                String c2 = next.getClass().getName();
                critterCount.put(c2, critterCount.get(c2) + 1);
                info.remove(other);
                // and add a new one to the grid
```

```
                    try {
                        grid[p2.x][p2.y] = makeCritter(next.getClass());
                        // This critter has been infected and is now locked
                        // for the rest of this turn
                        locked.add(grid[p2.x][p2.y]);
                    } catch (Exception e) {
                        throw new RuntimeException("" + e);
                    }
                    // and add to the map
                    info.put(grid[p2.x][p2.y], oldData);
                    // but it's new, so it didn't just hop
                    oldData.justHopped = false;
                }
            }
        }
        updateColorString();
    }

    // calling this method causes each critter to update the stored color and
    // text for toString; should be called each time update is performed and
    // once before the simulation begins
    public void updateColorString() {
        for (Critter next : info.keySet()) {
            info.get(next).color = next.getColor();
            info.get(next).string = next.toString();
        }
    }

    public Set<Map.Entry<String, Integer>> getCounts() {
        return Collections.unmodifiableSet(critterCount.entrySet());
    }
```

```
    public int getSimulationCount() {
        return simulationCount;
    }

    private class PrivateData {
        public Point p;
        public Critter.Direction direction;
        public Color color;
        public String string;
        public boolean justHopped;

        public PrivateData(Point p, Critter.Direction d) {
            this.p = p;
            this.direction = d;
        }

        public String toString() {
            return p + " " + direction;
        }
    }

    // an object used to query a critter's state (neighbors, direction)
    private static class Info implements CritterInfo {
        private Critter.Neighbor[] neighbors;
        private Critter.Direction direction;
        private boolean[] neighborThreats;

        public Info(Critter.Neighbor[] neighbors, Critter.Direction d,
                    boolean[] neighborThreats) {
            this.neighbors = neighbors;
            this.direction = d;
            this.neighborThreats = neighborThreats;
        }
```

```
    public Critter.Neighbor getFront() {
        return neighbors[0];
    }

    public Critter.Neighbor getBack() {
        return neighbors[2];
    }

    public Critter.Neighbor getLeft() {
        return neighbors[3];
    }

    public Critter.Neighbor getRight() {
        return neighbors[1];
    }

    public Critter.Direction getDirection() {
        return direction;
    }

    public boolean frontThreat() {
        return neighborThreats[0];
    }

    public boolean backThreat() {
        return neighborThreats[2];
    }

    public boolean leftThreat() {
        return neighborThreats[3];
    }

    public boolean rightThreat() {
        return neighborThreats[1];
    }
  }
}
```

Figure 16: CritterModel.java

Figure 16 shows CritterModel.java class used to create myModel objects

```java
// Class CritterPanel displays a grid of critters

import javax.swing.*;
import java.awt.*;
import java.awt.Point;
import java.util.*;

public class CritterPanel extends JPanel {
    private CritterModel myModel;
    private Font myFont;
    private static boolean created;

    public static final int FONT_SIZE = 12;

    public CritterPanel(CritterModel model) {
        // this prevents someone from trying to create their own copy of
        // the GUI components
        if (created)
            throw new RuntimeException("Only one world allowed");
        created = true;

        myModel = model;
        // construct font and compute char width once in constructor
        // for efficiency
        myFont = new Font( name: "Monospaced", Font.BOLD,  size: FONT_SIZE + 4);
        setBackground(Color.CYAN);
        setPreferredSize(new Dimension( width: FONT_SIZE * model.getWidth() + 20,
                                        height: FONT_SIZE * model.getHeight() + 20));
    }
```

```
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setFont(myFont);
        Iterator<Critter> i = myModel.iterator();
        while (i.hasNext()) {
            Critter next = i.next();
            Point p = myModel.getPoint(next);
            String appearance = myModel.getAppearance(next);
            g.setColor(Color.BLACK);
            g.drawString( str: "" + appearance,  x: p.x * FONT_SIZE + 11,
                          y: p.y * FONT_SIZE + 21);
            g.setColor(myModel.getColor(next));
            g.drawString( str: "" + appearance,  x: p.x * FONT_SIZE + 10,
                          y: p.y * FONT_SIZE + 20);
        }
    }
}
```

Figure 17: CritterPanel.java

Figure 17 shows CritterPanel.java class used to create myPicture objects. This class inherit JPanel to generate picture required.

```
// The CritterInfo interface defines a set of methods for querying the
// state of a critter simulation.  You should not alter this file.  See
// the documentation in the Critter class for a more detailed explanation.

public interface CritterInfo {
    public Critter.Neighbor getFront();
    public Critter.Neighbor getBack();
    public Critter.Neighbor getLeft();
    public Critter.Neighbor getRight();
    public Critter.Direction getDirection();
    public boolean frontThreat();
    public boolean backThreat();
    public boolean leftThreat();
    public boolean rightThreat();
}
```

Figure 18: CritterInfo.java

Figure 18 shows CritterInfo.java class interface which is use by other classes to animal, food and flytrap.

```
// This is the superclass of all of the Critter classes.  Your class should
// extend this class.  The class provides several kinds of constants:
//
//    type Neighbor  : WALL, EMPTY, SAME, OTHER
//    type Action    : HOP, LEFT, RIGHT, INFECT
//    type Direction : NORTH, SOUTH, EAST, WEST
//
// Override the following methods to change the behavior of your Critter:
//
//     public Action getMove(CritterInfo info)
//     public Color getColor()
//     public String toString()
//
// The CritterInfo object passed to the getMove method has the following
// available methods:
//
//     public Neighbor getFront();           neighbor in front of you
//     public Neighbor getBack();            neighbor in back of you
//     public Neighbor getLeft();            neighbor to your left
//     public Neighbor getRight();           neighbor to your right
//     public Direction getDirection();      direction you are facing
//     public boolean frontThreat();         threatening critter in front?
//     public boolean backThreat();          threatening critter in back?
//     public boolean leftThreat();          threatening critter to the left?
//     public boolean rightThreat();         threatening critter to the right?

import java.awt.*;

public class Critter {
    public static enum Neighbor {
        WALL, EMPTY, SAME, OTHER
    };
```

```
    public static enum Action {
        HOP, LEFT, RIGHT, INFECT
    };

    public static enum Direction {
        NORTH, SOUTH, EAST, WEST
    };

    // This method should be overriden (default action is turning left)
    public Action getMove(CritterInfo info) {
        return Action.LEFT;
    }

    // This method should be overriden (default color is black)
    public Color getColor() {
        return Color.BLACK;
    }

    // This method should be overriden (default display is "?")
    public String toString() {
        return "?";
    }

    // This prevents critters from trying to redefine the definition of
    // object equality, which is important for the simulator to work properly.
    public final boolean equals(Object other) {
        return this == other;
    }
}
```

Figure 19: Critter.java

Figure 19 shows a SuperClass Critter.java that is inherited by Bear.java, FlyTrap.java, Food.java, Giant.java, NinjaCat.java and Tiger.java.

```java
// This defines a simple class of critters that sit around waiting to be
// taken over by other critters.

import java.awt.*;
import java.sql.Struct;

public class Bear extends Critter {

    boolean polar;
    int step = 0;

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

    public Critter.Action getMove(CritterInfo info) {
        step++;
        if (info.getFront() == Neighbor.OTHER) {
            return Action.INFECT;
        }

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

        else {
            return Action.LEFT;
        }
    }
```

```
    public Color getColor() {

        if (true)
        return Color.white;
        else
        return Color.black;
    }

    public String toString() {

        if (step%2 == 0) {
            return "/";
        }
        else{
            return  "\\";
        }
    }
}
```

Figure 20: Bear.java

Figure 20 shows Bear.java that inherited Critter.java produces bear in the frame in which is generated by the program.

```java
// This defines a simple class of critters that infect whenever they can and
// otherwise just spin around, looking for critters to infect.  This simple
// strategy turns out to be surpisingly successful.

import java.awt.*;

public class FlyTrap extends Critter {
    public Action getMove(CritterInfo info) {
        if (info.getFront() == Neighbor.OTHER) {
            return Action.INFECT;
        } else {
            return Action.LEFT;
        }
    }

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

    public String toString() {
        return "T";
    }
}
```

Figure 21: FlyTrap.java

Figure 21 shows FlyTrap.java that inherited Critter.java produces flytrap in the frame in which is generated by the program.

```java
// This defines a simple class of critters that sit around waiting to be
// taken over by other critters.

import java.awt.*;

public class Food extends Critter {
    public Action getMove(CritterInfo info) {
        return Action.INFECT;
    }

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

    public String toString() {
        return "F";
    }
}
```

Figure 22: Food.java

Figure 22 shows Food.java that inherited Critter.java produces food in the frame in which is generated by the program.

```java
// This defines a simple class of critters that sit around waiting to be
// taken over by other critters.

import java.awt.*;

public class Giant extends Critter {

    int moves;
    public Giant (){
        getColor();
    }

    public Critter.Action getMove(CritterInfo info) {
        moves++;

        if (moves > 24){
            moves = 0;
        }
        if (info.getFront() == Neighbor.OTHER) {
            return Action.INFECT;
        }

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

        else {
            return Action.LEFT;
        }

    }
```

```
    public Color getColor() {
        return Color.gray;
    }

    public String toString() {

        switch (moves/6){
            case 1:
                return "fee";
            case 2:
                return "fie";
            case 3:
                return "foe";
            default:
                return "fum";
        }
    }
}
```

Figure 23: Giant.java

Figure 23 shows Giant.java that inherited Critter.java to create giants called fee, fie, foe and fum which changes at every six (6) steps in the frame, are generated by the program.

```java
// This defines a simple class of critters that sit around waiting to be
// taken over by other critters.
import java.awt.*;

public class NinjaCat extends Critter {

    public NinjaCat(){
        getColor();
    }

    public Critter.Action getMove(CritterInfo info) {

        if (info.getFront() == Neighbor.OTHER) {
            return Action.INFECT;
        }

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

        else {
            return Action.RIGHT;
        }
    }

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

    public String toString() {
        return "NC";
    }
}
```

Figure 24: NinjaCat.java

Figure 24 shows NinjaCat.java that inherited Critter.java to create ninjacat in the frame is generated by this program.

```java
// This defines a simple class of critters that sit around waiting to be
// taken over by other critters.

import java.awt.*;
import java.util.*;

public class Tiger extends Critter {

    int colorMoves;
    Color tigerColor;
    Random rand = new Random();

    public Tiger(){
        getColor();
    }
```

```java
    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( bound: 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;
    }
```

```
    public String toString() {
        return "TGR";
    }

    public Action getMove(CritterInfo info) {

        colorMoves++;
        if (info.getFront()==Neighbor.OTHER){
            return Action.INFECT;
        } else if (info.getFront()==Neighbor.WALL||info.getRight()==Neighbor.WALL){
            return Action.LEFT;
        } else if (info.getFront()==Neighbor.SAME){
            return Action.RIGHT;
        } else {
            return Action.HOP;
        }
    }
}
```

Figure 25: Tiger.java

Figure 25 shows Tiger.java that inherited Critter.java to create tigers at random colours mainly, red, green and blue. These three colours change every three steps continuously in the frame that is generated by the program.

```java
// This defines a simple class of critters that sit around waiting to be
// taken over by other critters.
import java.awt.*;

public class WhiteTiger extends Tiger {

    boolean infected = false;

    public Critter.Action getMove(CritterInfo info) {
        if (info.getFront() == Neighbor.OTHER) {
            infected = true;
            return Action.INFECT;
        } else if (info.getFront() == Neighbor.WALL || info.getRight() == Neighbor.WALL) {
            return Action.LEFT;
        } else if (info.getFront() == Neighbor.SAME) {
            return Action.RIGHT;
        } else {
            return Action.HOP;
        }
    }

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

    public String toString() {
        if (infected == true) {
            return "tgr";
        } else {
            return "TGR";
        }
    }
}
```

Figure 26: WhiteTiger.java

Figure 26 shows WhiteTiger.java that inherited Tiger.java. represents as TGR symbol if not infected while the tgr symbol if infected. WhiteTiger in the frame is generated by the program. The following are the output results when the program:

Results

Output of this program are as follows:

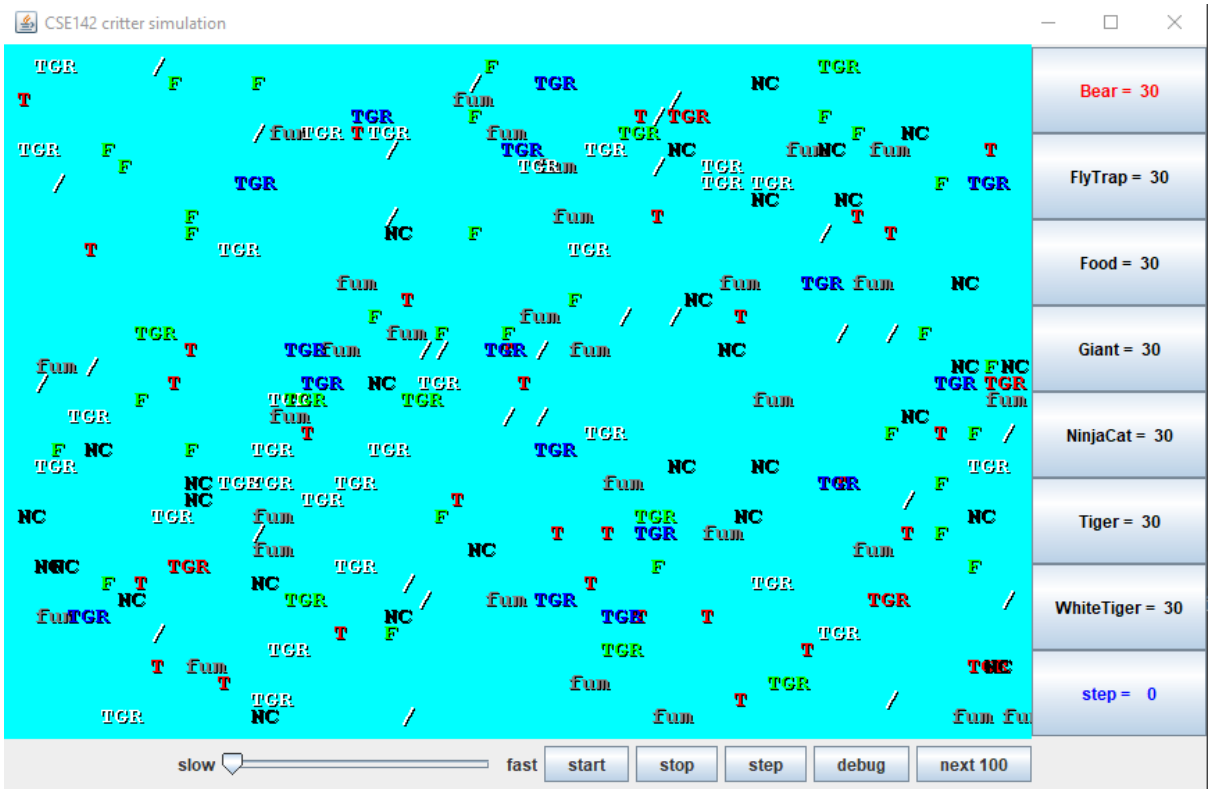

Figure 27: Output 1

Figure 27 shows the display when the program is run. The display output is different each time the program is run. Objects displayed in the frame are randomly place at any location coordinates. At the beginning of the program, all objects are arranged with the starting value of 30.

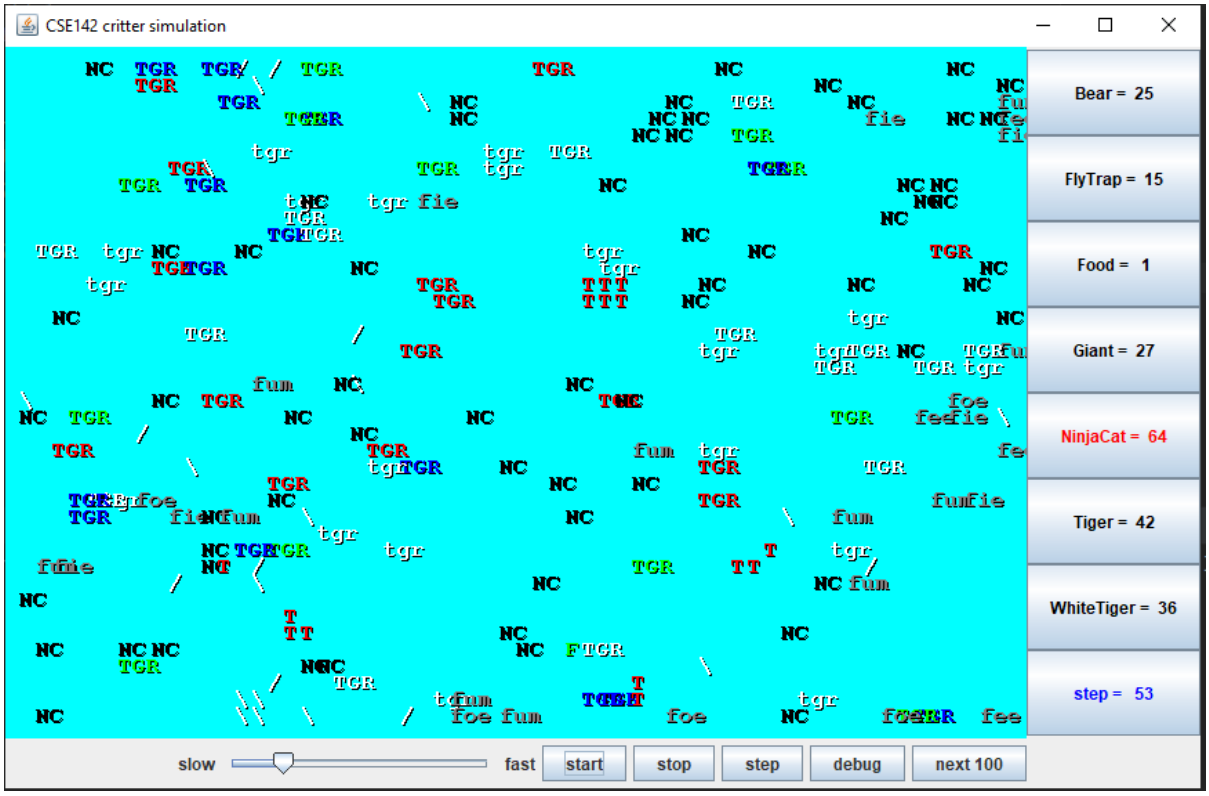

Figure 28: Output 2

Figure 28 shows the display after the program is played until step 53. The display shows that each object in the frame move according to the program that has established based on the workflow and logic mentioned above. Observations indicates that objects may decrease or increase from the number starting at 30.

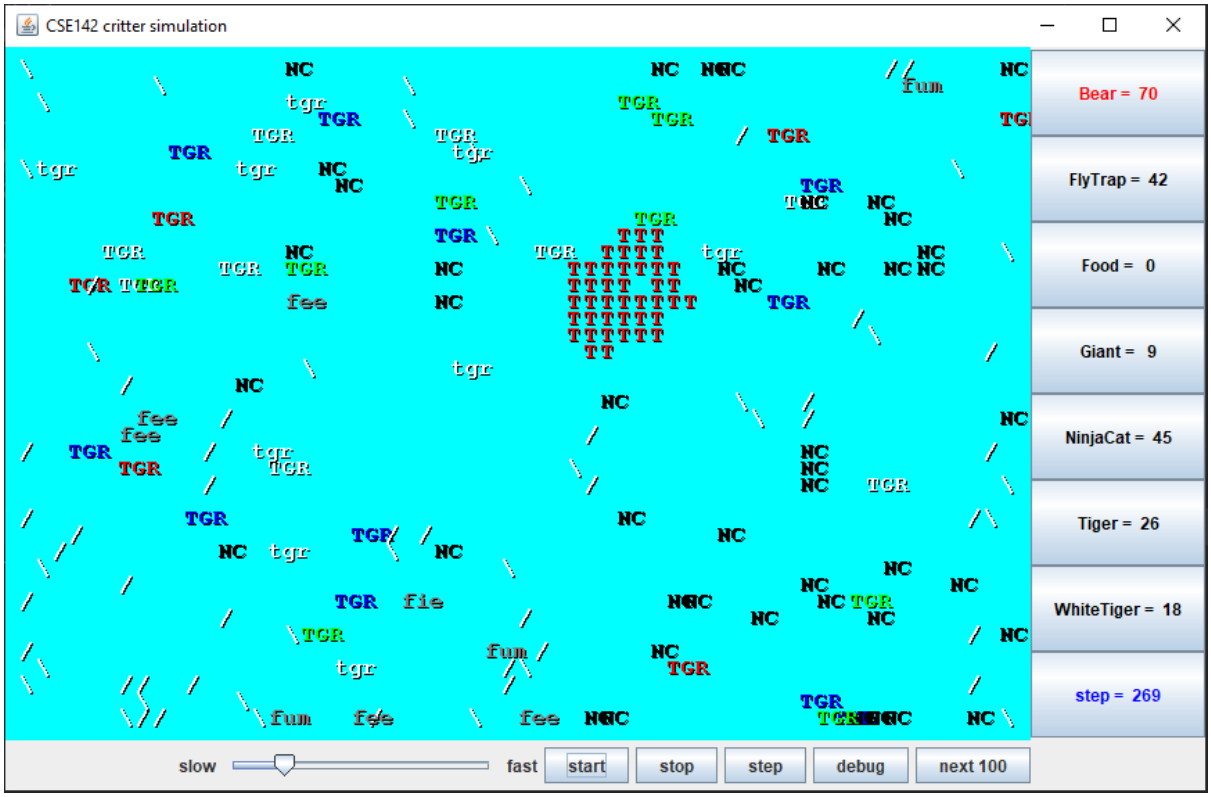

Figure 29: Output 3

Figure 29 shows the display after the program is played until step 269. This display shows a significant change compared to Figure 28 where objects decrease or increase drastically when Food object has become 0.

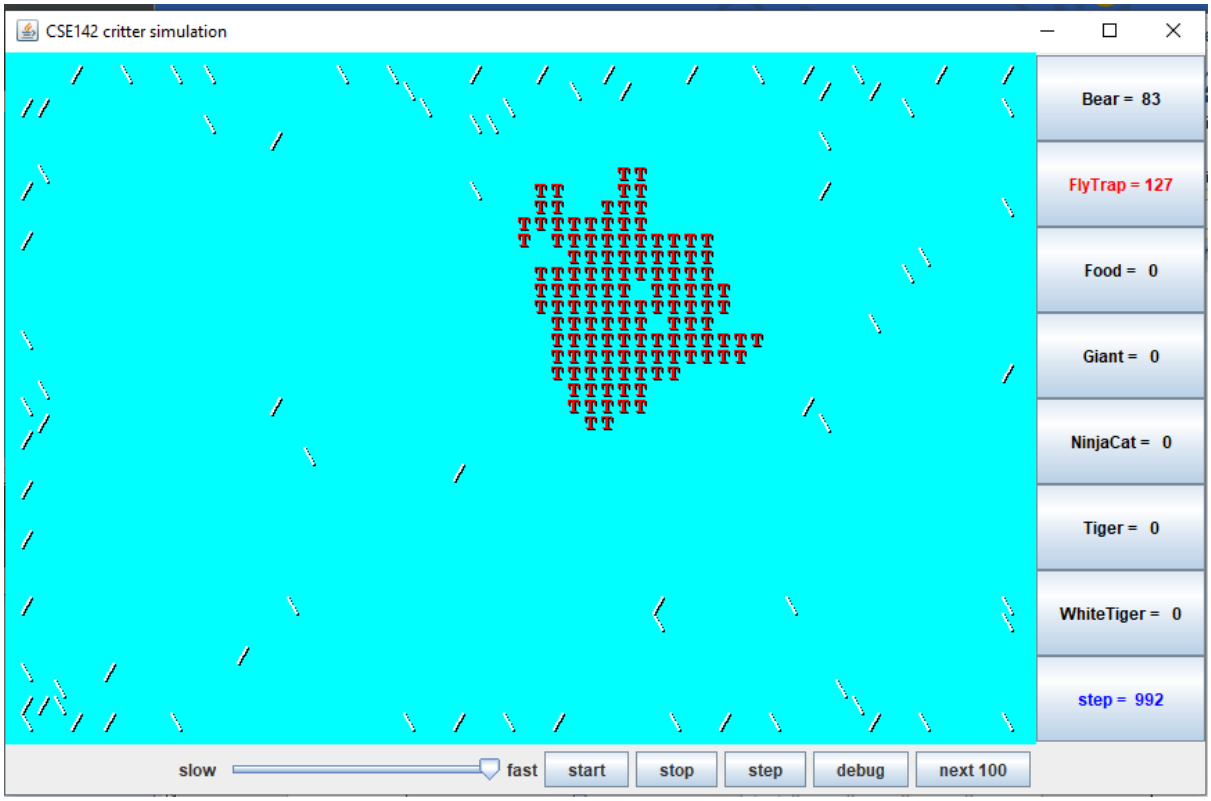

Figure 30: Output 4

Figure 30 shows the display after the program is played until step 992. Display shows only Bear living with FlyTrap.

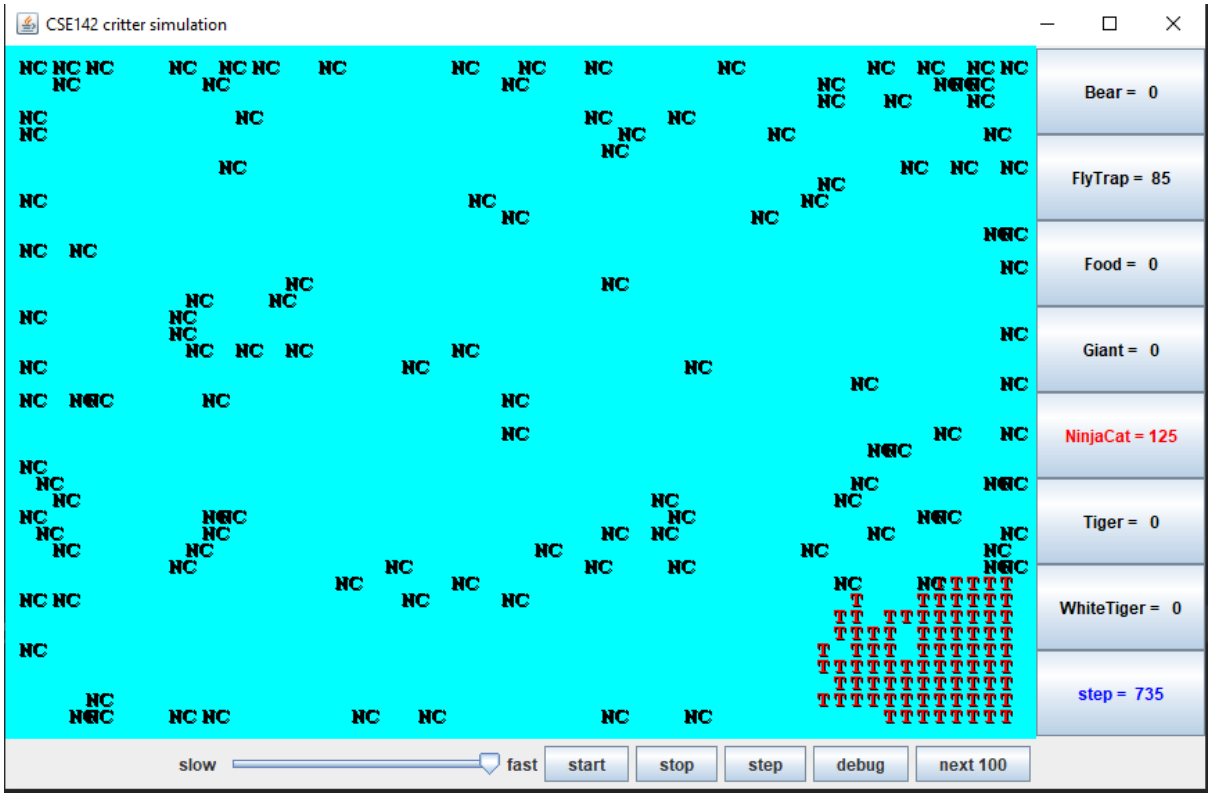

Figure 31: Output 5

Figure 31 shows the display after the program is played until step 735 when the program is run for the second time. Display shows only NinjaCat living with FlyTrap.

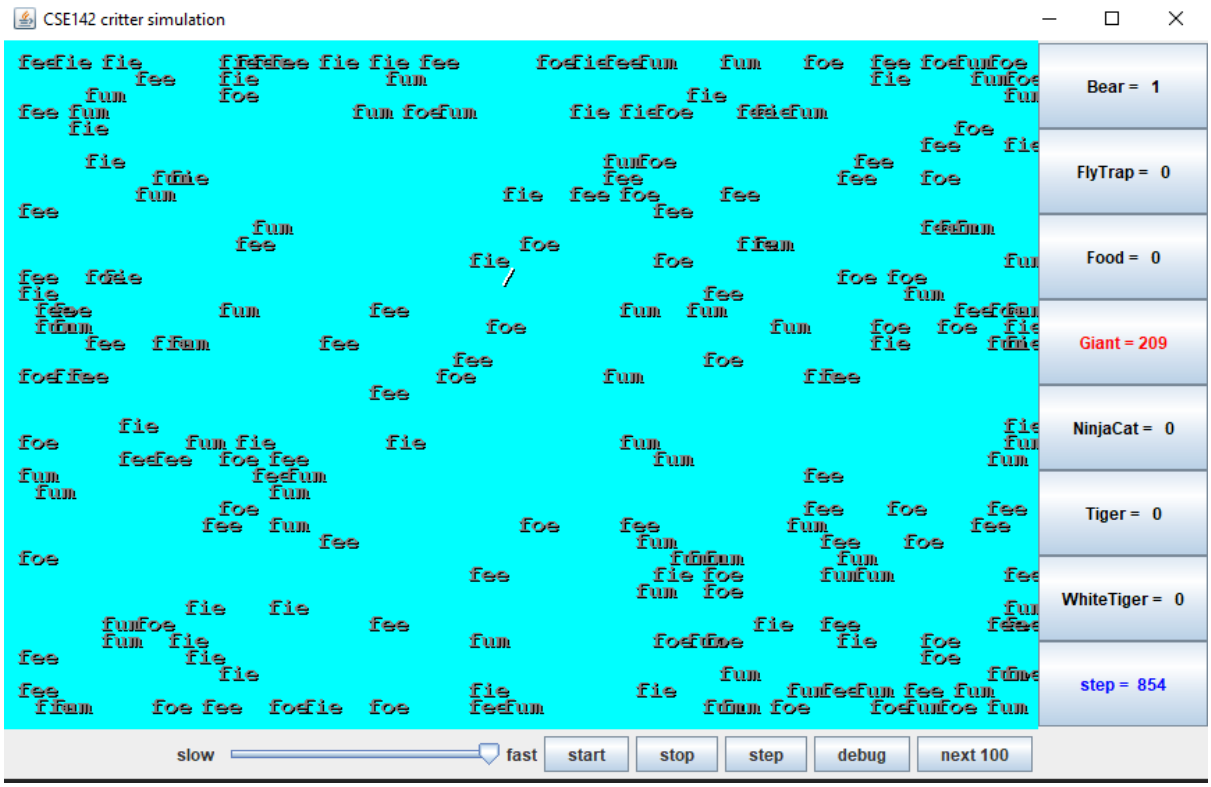

Figure 32: Output 6

Figure 31 shows the display after this program is played until step 854 when the program is run for the third time. Display shows only Giants and one Bear

.

Discussion

Based on the results obtained each time the program is run, we find that the program produces objects according to the initial number arranged of 30 units each for each class and are displayed at random position in the frame. The differences are seen each time the program started.

At the start of press button, the object in the frame moves accordingly to the program generated. We find that these movements are rendered randomly and differently each time they conducted.

When the program runs continuously, there is an increasing and decreasing number of objects. This number changes each time the program is run. If allow to continue until the end, only one or two objects will remain and the objects are generally dissimilar each time the program is run as shown in Figure 30, 31 and 32.

Conclusion

As conclusion, this program provides more in -depth exposure on how to use a class in java not just using one inheritance but several inheritances to accomplish in a program. The class can also be broken down into module forms i.e., GUI modules and objects in frame display.

This project introduces a way to produce output using the GUI facilities available in the Java Object Oriented Programming (OOP) Language. With such exposure, it can enhance knowledge, experience and skills of students to explore Java programming language broadly especially in OOP that are used as one of the alternative programming languages which can be offered to produce programs specifically requested by users.