PROJECT (40%)

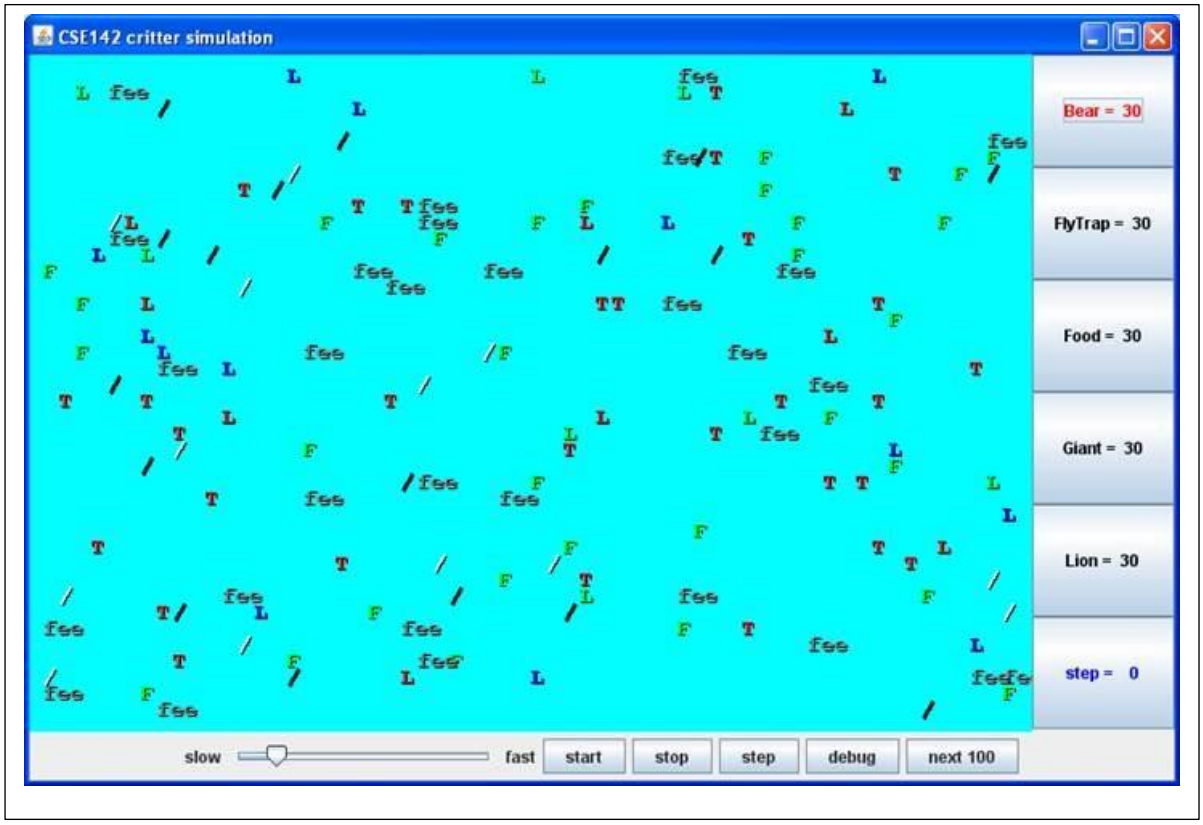
Capstone Project

PROJECT DESCRIPTION

Final Project - Animal Kingdom

This assignment will give you practice with defining classes. You are to write a set of classes that define the behavior of certain animals. You will be 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 you are defining those differences.

For this assignment, you will be given a lot of supporting code that runs the simulation. While it is running, the simulation will look something like this:



Download each of the following files (downloadable files will be made available in the course) and make sure they are included in the same project/folder:

1. Critter.java
2. CritterInfo.java
3. CritterPanel.java
4. CritterModel.java
5. CritterMain.java
6. CritterFrame.java
7. Food.java
8. FlyTrap.java

PROJECT (40%)

Every object in this world is a "critter", where Critter is the super class with default behavior defined. You will be writing five classes, each representing a different type of Animal: Bear, Tiger, WhiteTiger, Giant and NinjaCat. All of the classes you right should be sub classes of Critter. On each round of the simulation, each critter is asked for 3 pieces of information:

1. How should it act?
2. What color is it?
3. What string represents that critter?

These 3 pieces of information are provided by 3 methods present in each Critter class. You will be responsible for overriding these methods and programming their appropriate behavior:

```
public Action getMove(CritterInfo info) {

...

}

public Color getColor() {

...

}

public String toString() {

...

}
```

For example, here is a Food class who would always appear as a green “F” that would try to infect whatever critter is in front of it:

```
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";

}

}
```

Notice that it begins with an import declaration to be able to access the Color class. All of your Critter classes will have the basic format shown above.

YOUR FIVE CLASSES' BEHAVIOR

Bear

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.

Tiger

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.

WhiteTiger

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.

PROJECT (40%)

Giant

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.

NinjaCat

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

THE METHODS

getColor()

As the simplest method, we suggest you start writing the getColor() method first. For getColor you should return whatever color you want the simulator to use when drawing your critter. Colors are represented like "Color.WHITE". For the random colors, each possible choice must be equally likely. You may use either a Random object or the Math.random() method. If your color changes based on moves, you will want some way to count how many moves a critter has made. This state should only be updated in the getMove method, and simply referenced in others (like getColor or toString).

toString()

For the toString method, you should return whatever text you want the simulator to use when displaying your critter (normally a single character). Remember, that if your String changes based on moves you might need to keep track of the critter's number of moves. For this assignment, do not worry about integer overflow (we won't run the simulation that long).

getMove()

The getMove method can return only 4 different things:

Constant	Description
Action.HOP	Move forward one square in its current direction
Action.LEFT	Turn left (rotate 90 degrees counter-clockwise)
Action.RIGHT	Turn right (rotate 90 degrees clockwise)
Action.INFECT	Infect the critter in front of you

PROJECT (40%)

This is why your return type is "Action". You will need to use information about the area around your Critter to determine which Action to return. For example, Bear, Tiger, WhiteTiger and NinjaCat should all return Action.INFECT if an enemy is in front of them. You can tell what is around your critter based on the CritterInfo object parameter:

CritterInfo Method	Description	Possible Return Values
public Neighbor getFront()	returns neighbor in front of you	Neighbor.WALL, Neighbor.EMPTY, Neighbor.SAME, Neighbor.OTHER
public Neighbor getBack()	returns neighbor in back of you	Neighbor.WALL, Neighbor.EMPTY, Neighbor.SAME, Neighbor.OTHER
public Neighbor getLeft()	returns neighbor to your left	Neighbor.WALL, Neighbor.EMPTY, Neighbor.SAME, Neighbor.OTHER
public Neighbor getRight()	returns neighbor to your right	Neighbor.WALL, Neighbor.EMPTY, Neighbor.SAME, Neighbor.OTHER
public Direction getDirection()	returns direction you are facing	Direction.NORTH, Direction.SOUTH, Direction.EAST, Direction.WEST
public boolean frontThreat()	whether there is an enemy facing you in front of you	TRUE or FALSE
public boolean backThreat()	whether there is an enemy facing you in back of you	TRUE or FALSE
public boolean leftThreat()	whether there is an enemy facing you to your left	TRUE or FALSE
public boolean rightThreat()	whether there is an enemy facing you to your right	TRUE or FALSE

FAQ

- You are not writing the main method; your code will not be in control. Instead, you are defining a series of objects that become part of a larger system. For example, you might find that you want to have one of your critters make several moves all at once. You won't be able to do that. The only way a critter can move is to wait for the simulator to ask it for a move. The simulator is in control, not your critters.
- Critters move around in a world of finite size that is enclosed on all four sides by walls.
- You should include a constructor for each class you write as most classes need to keep track of number of moves or other variables.
- For classes that always use the same Strings/Colors, you should declare them as class constants.
- The simulator has several supporting classes that are included (CritterModel, CritterFrame, etc). You can in general ignore these classes. When you compile CritterMain, these other classes will be compiled. The only classes you will have to modify and recompile are CritterMain (if you change what critters to include in the simulation) and your own individual Critter classes.
- Do not implement any Critter behavior in the CritterMain class. The only classes that are graded are your 5 Critter classes, so if the behavior is not in that class, it is not graded.

PROJECT (40%)

TESTING

You will notice that CritterMain has lines of code like the following:

```
// frame.add(30, Tiger.class);
```

You should uncomment these lines of code as you complete the various classes you have been asked to write. Then critters of that type will be included in the simulation.

The simulator provides great visual feedback about where critters are, so you can watch them move around the world. But it doesn't give great feedback about what direction critters are facing. The simulator has a "debug" button that makes this easier to see. When you request debug mode, your critters will be displayed as arrow characters that indicate the direction they are facing.

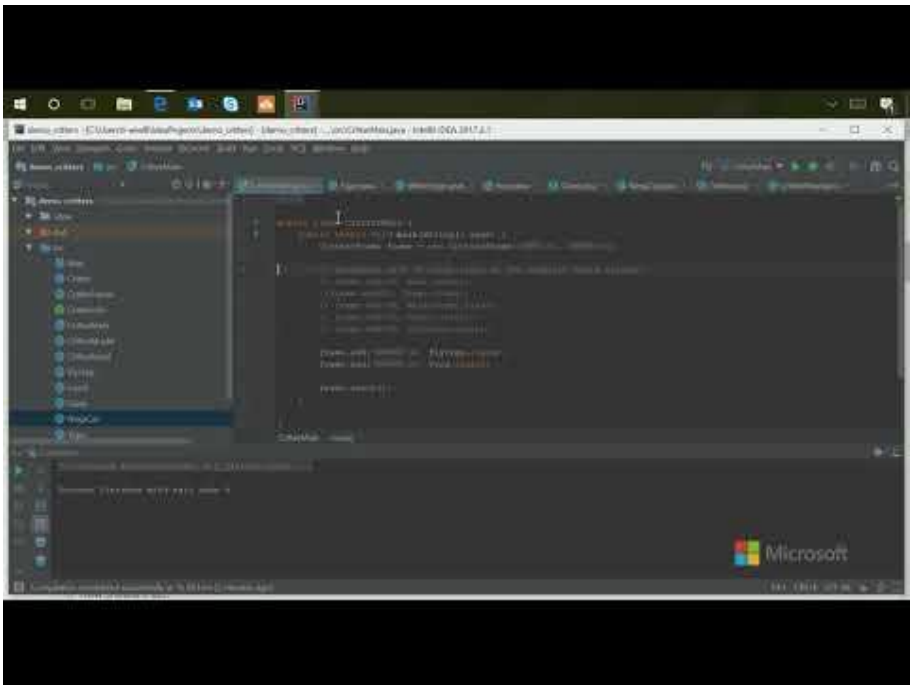
The simulator also indicates the "step" number as the simulation proceeds (initially displaying a 0).

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.

PROJECT (40%)

PROJECT HELP



ADDITIONAL INSTRUCTIONS:

- Please make sure that the code is fully documented
- When project is complete, you need to publish it on ***github***
- Apply all professional coding principles you have learnt during this course.
- You are required to write a report (10-15 pages, Times New Roman, 12 Font Size,1.5 Spacing), with following outlines:
    - Abstract
    - Introduction
    - Project Description
    - Project Design
    - Program workflow and logics
    - Development details
    - Results
    - Discussion
    - Conclusion

PROJECT (40%)

RUBRIC:

Your submission will be graded based on the following criterias:

	Unsatisfactory (25%)	Fair (50%)	Good (75%)	Excellent (100%)
Delivery and report presentation **10 Points**	• Completed less than 50% of the requirements. • Not delivered on time or not in correct format (disk, email, Canvas, printout etc.) • Does not comply with requirements (does something other than requirements).	• Completed between 50-64% of the requirements. • Delivered on time, and in correct format.	• Completed between 65-89% of the requirements.	• Completed at least 90% of requirements
Coding Design Standards **5 Points**	• There is no software design at all • No programmer name included • Poor use of white space (indentation, blank lines) making code hard to read. • Disorganized and messy • Ambiguous identifiers.	• There is a short description of the software project • Includes name, and assignment title. • White space makes program fairly easy to read. • Organized work. • Good use of variables.	• There is a comprehensive description of the software project • Good use of white space. • Organized work. • Good use of variables and constants • Minimum line-wrap	• There is a comprehensive description of the software project with UML diagrams • Excellent use of white space. • Creatively organized work. • Excellent use of variables and constants. • No magic numbers. • Correct identifiers for constants. • No line-wrap
Documentation **15 Points**	• No documentation included.	• Basic documentation has been completed including a summary of requirements. • Purpose is noted for each function.	• Purpose is noted for each function and control structure. • One sample run included	• Specific purpose is noted for each function, control structure, input requirements, and output results.

PROJECT (40%)

Development and Runtime **10 Points**	• The code doesn't do what it is supposed to do • Does not execute due to syntax errors. • Does not execute due to runtime errors (endless loop, crashes etc.) • User prompts are misleading or non-existent. • No testing has been completed.	• The code does what it is supposed to do, but with unnecessary instructions and potential improvements to the logic • Executes without errors. • User prompts contain little information, poor design. • Some testing has been completed.	• The code does what it is supposed to do • Executes without errors. • User prompts are understandable, minimum use of symbols or spacing in output. • Thorough testing has been completed	• The code does what it is supposed to do and also it is optimized to run quickly • Executes without errors excellent user prompts, good use of symbols, spacing in output. • Thorough and organized testing has been completed and output from test cases is included.
Efficiency **10 Points**	• A difficult and inefficient solution.	• A logical solution that is easy to follow but it is not the most efficient.	• Solution is efficient and easy to follow (i.e. no confusing tricks).	• Solution is efficient, easy to understand, and maintain.

End of Lesson