TCSS143
Fundamentals of Object-Oriented Programming-Theory and Application
Programming Assignment 5

DUE: Tuesday, October 28, by 11:59pm

The purpose of this programming project is to reinforce the understanding of inheritance, polymorphism, and interfaces while illustrating the purpose of and including an Abstract class into an inheritance hierarchy.

REQUIREMENTS

You will submit all source code files (**8 in all**) as a **single zipped file named "Programming5.zip**" through the Programming Assignment 5 Submission link on Canvas.

Not only will you be graded on program correctness (Program executes correctly, proper use of methods, classes, inheritance, etc.) but also, good programming documentation techniques including javadoc, proper indentation, correct locations of braces, meaningful identifier names, general comments prior to methods, specific comments on complex code, etc. **Not adhering to our documentation requirements will receive a 10 point reduction on the grade**.

DETAILS

You will create an Interface that lists constants and abstract methods to be implemented by various Critter classes, an Abstract class which partially implements one of the Interface's abstract methods that is identical for any of the Critter classes as well as a related Instance field and Constructor, and finally, 6 classes that represent various Critters (actually, one is an inanimate Stone) that have basic movement characteristics. All 6 Critter classes will inherit (extend) the Abstract class mentioned. Like with the previous assignment, one major purpose of inheritance is to eliminate redundant code. For this reason you should take full advantage of super class code or methods when developing your subclasses. In this case, the super class will be this Abstract class. **Always remember to use final constants in place of any literal constants (anything other than 0 & 1) you would normally use in your code!** Greater detail of each class follows:

1. Critter.java

public interface Critter {
Decribes what methods to be implemented by Critter subclasses and includes several constants.

Finals: Here you will define 5 int constants (final) for the direction an animal can move. The names for each of these will **HAVE TO BE** as follows (the model expects these names):

NORTH
WEST
SOUTH
EAST
CENTER

You can assign any values to them as you see fit, as long as each is unique (may I suggest simply 0, 1, 2, 3, 4 respectively)

Fields: None. Interfaces can not be instantiated and thus, cannot contain instance fields.

Abstract Methods: (the only type of method that can exist in an Interface). These must have the following header signatures:

```
public char getChar();
public int getMove(CritterInfo theInfo);
```

(CritterInfo is an interface already defined and used by an inner class of the CritterModel class. 5 of the Critter classes will simply not use this parameter. However, the Frog class uses the getNeighbor method via the theInfo parameter, discussed in the Frog.java details)

2. AbstractCritter.java

public abstract class AbstractCritter implements Critter
Declares an Instance variable that all subclasses must have, a constructor to initialize the instance variable, and implements the getChar method (this way this method is only implemented once and to be inherited by all Critter subclasses.

Fields: A char field to hold the single character which represents a particular Critter and is used for display on the Critter Environment Grid (fancy name for what you see in the JPanel of the GUI).

Constructor – public AbstractCritter(final char theChar)
Assigns theChar to the Instance Field (This has to be called by all subclasses to set the correct character for the class).

Abstract Methods: None declared here. The responsibility to Implement the getMove method from Critter is passed down to the inheriting subclasses as getMove has to be unique within each Critter subclass.

None Abstract Methods/Implemented methods from the Critter Interface:

```
public char getChar() {
    // Your code goes here.
}
```

Returns the Instance Field.

3. Stone.java

public class Stone extends AbstractCritter
Stone is the only inanimate Critter. Several Stone objects are placed on the simulation grid during the initial instantiation and remain stationary throughout the simulation. All but one Critter dies when crashing into a Stone. Frog never moves to where a Stone is and waits until it moves elsewhere. Details to follow.

Constructor – public Stone( )
Calls the super constructor passing it an **UPPERCASE** S: 'S' char to initialize the super's instance field.

Implemented methods from the Critter Interface:
Implements the super getMove method:

```
public int getMove(CritterInfo theInfo) {
    // Since Stones never move this should simply return CENTER.
}
```

4. Bat.java

public class Bat extends AbstractCritter
Bats have rather irratic behavior as they flutter about catching flying insects. Their direction appears random, flying in one of the 4 constant directions (CENTER is not an option since they can't really hover).

Constructor – public Bat( )
Calls the super constructor passing it an uppercase B: 'B' char to initialize the super's instance field.

Implemented method from the Critter Interface:
Implements the super getMove method:

```
public int getMove(CritterInfo theInfo) {
    // Randomly chooses and returns one of the 4 directions: NORTH, WEST, SOUTH, EAST, each
    // having a 25% chance of being chosen.  .
}
```

5. Wolf.java

public class Wolf extends AbstractCritter

Because they are pack animals, their movement is somewhat consistent and is done in parallel with all wolves. However, they tend to travel in straight directions for some time before changing. This will require several fields to create a pattern of behavior:

Fields (all initializations are done in the Constructor):

myDirection: int to remember the current direction. Initialized to EAST.
myCount: int to count the number of moves before a change of direction. Initialized to 0.
myFirst: boolean toggles (true/false) whenever a change of direction takes place. Initialize to true.
myTarget: int Details of its use is decribed in getMove below. Initialized to 1.

Constructor – public Wolf( )

Calls the super constructor passing it an **UPPERCASE** W: 'W' char to initialize the super's instance field. Initialize the 4 fields to the values mentioned above.

Implemented methods from the Critter Interface:

Implements the super getMove method:

```
    public int getMove(CritterInfo theInfo) {
/*
```

Begin by setting a local int variable (let's say direction) to myDirection. This will be returned as is and not changed until the next time getMove is called. Next, increment the myCount, then:
when myCount reaches myTarget, several things will happen:

if myFirst is false, myTarget should be incremented.
toggle myFirst to the opposite state (i.e.false to true, true to false)
myCount is reset to 0
myDirection is changed (in a counterclockwise direction, i.e. if myDirection is EAST, set it to NORTH, if NORTH, set to WEST, etc.

Of course, this change is what will be assigned the local direction variable next time getMove is called.
return the local direction variable value. */

```
    }
```

6. Mouse.java

public class Mouse extends AbstractCritter

Mice move in a zigzag (right-left-right-left, etc.) motion. Their overall direction then is NORTH-WEST-NORTH-WEST, or WEST-SOUTH-WEST-SOUTH, or SOUTH-EAST... etc.. The general direction of this zigzag motion should be chosen at random intervals. How you go about accomplishing this will be up to you.

Fields (all initializations are done in the Constructor):

You will have to decide just what you need here in order to fulfill the zigzag/random direction of a Mouse.

Constructor – public Mouse( )

Calls the super constructor passing it an **UPPERCASE** M: 'M' char to initialize the super's instance field. Initialize all other Mouse class instance fields.

Implemented methods from the Critter Interface:

Implements the super getMove method:

```
    public int getMove(CritterInfo theInfo) {
/*
```

Like the Wolf and Bat, the movement of the mouse will be determined by what getMove returns (NORTH, WEST, SOUTH, EAST) to the calling model object (the program using your Critter classes.
*/

```
    }
```

7. Turtle.java

public class Turtle extends AbstractCritter

As we all know, turtles move rather slowly. As such, they will only move 1/3 as many times as all other Critters (excluding the Stone, of course, which doesn't move at all). The Turtle getMove method can return a direction of CENTER as needed, which will keep it from moving.

Also, each Turtle object will make its first (following instantiation) actual move (none CENTER) based on some randomly chosen value (NORTH, WEST, SOUTH, EAST).

When a Turtle does move (every 3 calls to getMove), if one randomly chosen boolean value is true and a second randomly chosen int value of:

0 -move NORTH
1 -move EAST
2 -move SOUTH
3 -move WEST

If this randomly chosen boolean value is false, then the direction based on the second random (int) value is:

0 -move SOUTH
1 -move WEST
2 -move NORTH
3 -move EAST

Fields (all initializations are done in the Constructor):

Whatever you might need to satisfy the movement of a Turtle.

Constructor – public Turtle( )

Calls the super constructor passing it an **UPPERCASE** T: 'T' char to initialize the super's instance field. Initialize all necessary instance fields of this class.

Implemented methods from the Critter Interface:

Implements the super getMove method:

```
    public int getMove(CritterInfo theInfo) {
/*
This is based on the rules for a Turtle mentioned above.
*/
    }
```

8. Frog.java

public class Frog extends AbstractCritter

The movement of a frog is a bit less complicated than some of our other Critters. Frogs never jump into Stones and thus, never die unless encountering another Critter. Generally, a frog will continue in the same direction unless a counter (one of the fields) is zero. If the counter is zero then a random value should be chosen (in this case, use the Math.random() which returns a double in the range of 0..1.0 (1.0 exclusive). If the result is < 0.25, your direction field should be set to NORTH, else if < 0.5, go SOUTH, < .75, go EAST otherwise go WEST (Remember, this change of direction only occurs when the counter is zero).

All other random number generation (for the other Critters) can use the Random class of which we are most familiar.

After the test for counter == 0 (and the above mentioned direction change), the counter should be incremented, then checked to see if it equals 3. If so, reset it to zero.

Next, if the neighbor of the Frog (based on the current direction) is the Stone (see next page for example of using theInfo class getNeighbor method), getMove should set the returning value to CENTER (thus keeping the frog from encountering a Stone). Otherwise, the return value should be set to the current direction (This suggests a local variable used to avoid multiple return statements. Finally, return this value.

Your code inside getMove of Frog based on the previous discussion and the use of getNeighbor:

```
...
        if (theInfo.getNeighbor(myDirection) == 'S') {
            result =  CENTER;  // If a Stone, sit still.
         }else {
            result = myDirection;
         }
      return result;
   }
```

Fields (all initializations are done in the Constructor):

myDirection: int to remember the current direction.
myCount: int to count the number of moves before a change of direction.

Constructor – public Frog( )

Calls the super constructor passing it an **UPPERCASE** F: 'F' char to initialize the super's instance field.
myDirection: Initialized to CENTER.
myCount: Initialized to 0.

Implemented method from the Critter Interface:

Implements the super getMove method:

```
    public int getMove(CritterInfo theInfo) {
/*
This is based on the discussion above (previous page).
*/
    }
```

Whenever Critters collide, one will pass on and will be elliminated from the simulation. All Critters except Frogs will die when they run into a Stone. These unformtunate occurances are implemented in the Model portion of the simulation and should not be a concern of the Critter related classes.

The Driver, Model, and View files are included in the zip file on Canvas. To test your classes, make sure you have unzipped the following files and compile Critter.java:

CritterMain.java
CritterInfo.java
CritterPanel.java
CritterFrame.java
CritterModel.java

You should create the Interface and Abstract class first (Critter.java & AbstractCritter.java). Then, create one Critter class at a time and test it. To do so, document out all all the frame.add statements in CritterMain except for the class you're testing. The simulator will run with just the selected Critter.

There is also a separate jar file which, when downloaded, can be double clicked to execute a full working version of CritterMain. This file is named CritterMain.jar. I can't guarantee its execution on anything other than a windows based machine.