13331231 孙圣 Part4

Set7

1. What methods are implemented in Critter?

   act(); getActors(); processActors(); getMoveLocations(); selectMoveLocation(); makeMove();

2. What are the five basic actions common to all critters when they act?

   ArrayList<Actor> actors = getActors();
   processActors(actors);
   ArrayList<Location> moveLocs = getMoveLocations();
   Location loc = selectMoveLocation(moveLocs);
   makeMove(loc);

3. Should subclasses of Critter override the getActors method? Explain.

   Yes. Different critters have different ways to find their neighbors (e.g. Some find all their neighbors while others just find the neighbors to the front.), so some subclasses should override the method to implement different fuctions.

4. Describe the way that a critter could process actors.

   Change the color of the actors.

5. What three methods must be invoked to make a critter move? Explain each of these methods.

   1) g*etMoveLocations();* Get all the neighboring locations that a critter can move to.
   2) selectMoveLocation(); From all the locations, select just one location randomly for the critter to move to.
   3) makeMove(); Perform the move action.

6. Why is there no Critter constructor?

   Because Critter extends Actor, Actor has a constructor and there is no need to change the default color and direction of a critter set by the constructor of class Actor. So even there is no constructor for Critter, it still can call the constructor of Actor to initialize.

Set8

1. Why does act cause a ChameleonCritter to act differently from a Critter even though ChameleonCritter does not override act?

   Because ChameleonCritter overrides *processActors();* and *makeMove();* and act() calls these two methods, so it can act differently.

2. Why does the makeMove method of ChameleonCritter call super.makeMove?

   This can avoid duplicates, i.e. writing the code for many times. ChameleonCritter first need to change its direction so that it heads the location which it will move to, then it calls *super,makeMove()* to finish the moving action.

3. How would you make the ChameleonCritter drop flowers in its old location when it moves?

```
1       public void makeMove(Location loc)
2       {
3           ocation oldLoc = getLocation();
4           setDirection(getLocation().getDirectionToward(loc));
5           super.makeMove(loc);
6           Flower flower = new Flower(getColor());
7           flower.putSelfInGrid(gr, oldLoc);
8       }
```

4. Why doesn't ChameleonCritter override the getActors method?

   Because it differs with Critter in the way how it processes actors, not how it gets actors. So getActors(); stays the same without being overridden.

5. Which class contains the getLocation method?

   Class Actor.

6. How can a Critter access its own grid?

   By calling getGrid(); which is defined in class Actor.

Set 9

1. Why doesn't CrabCritter override the processActors method?

   Because CrabCritter also eats the returned actors as defined in *processActors();* in Critter.

2. Describe the process a CrabCritter uses to find and eat other actors. Does it always eat all neighboring actors? Explain.

   It gets the actors in three specific locations(AHEAD, HALF_LEFT, HALF_RIGHT), then actors other than rocks and critters in these locations are eaten by calling the method *processActors();*

   No. For example, it does not eat actors that are to the right or to the left and it does not eat rocks and other critters either.

3. Why is the getLocationsInDirections method used in CrabCritter?

   For *getActors();* it needs to get three different locations; for *getMoveLocations();* it needs to get two different locations. So in order to avoid duplicate, *getLocationsInDirections();* is used to get different valid locations.

4. If a CrabCritter has location (3, 4) and faces south, what are the possible locations for actors that are returned by a call to the getActors method?

   (4, 3), (4, 4), (4, 5).

5. What are the similarities and differences between the movements of a CrabCritter and a Critter?

   Similarities:

   They all move randomly to the location that is not occupied.

   Differences:

   CrabCritter only moves to the left or to the right, and if it cannot move, it will turn left of turn right. But Critter can move to eight different places and it does not turn when it cannot move.

6. How does a CrabCritter determine when it turns instead of moving?

   In the method *makeMove();* if a CrabCritter's location equals the location returned by *selectMoveLocation();* (i.e there is no place for it to move to), the CrabCritter will turn instead of moving.

7. Why don't the CrabCritter objects eat each other?

   Because in method *processActors();* when the actor is an instance of Critter, it will not be removed.

Exercises

1. Modified ChameleonCritter

```
31  public class ChameleonCritter extends Critter
32  {
33      /**
34       * Randomly selects a neighbor and changes this critter's color to be the
35       * same as that neighbor's. If there are no neighbors,
36       * the color of the ChameleonCritter will darken.
37       */
38
39      private static final double DARKENING_FACTOR = 0.05;
40      // lose 5% of color value in each step
41
42      public void processActors(ArrayList<Actor> actors)
43      {
44          int n = actors.size();
45          if (n == 0)
46          {
47              Color c = getColor();
48              int red = (int) (c.getRed() * (1 - DARKENING_FACTOR));
49              int green = (int) (c.getGreen() * (1 - DARKENING_FACTOR));
50              int blue = (int) (c.getBlue() * (1 - DARKENING_FACTOR));
51
52              setColor(new Color(red, green, blue));
53          }
54          else
55          {
56              int r = (int) (Math.random() * n);
57
58              Actor other = actors.get(r);
59              setColor(other.getColor());
60          }
61      }
62
63      /**
64       * Turns towards the new location as it moves.
65       */
66      public void makeMove(Location loc)
67      {
68          setDirection(getLocation().getDirectionToward(loc));
69          super.makeMove(loc);
70      }
71  }
72
```

2. ChameleonKid

```java
public class ChameleonKid extends ChameleonCritter {
    /**
     * Randomly selects a neighbor immediately in front or behind and
     * changes this critter's color to be the same as that neighbor's.
     * If there are no neighbors, the color of the ChameleonCritter will darken.
     */

    public ArrayList<Actor> getActors()
    {
        ArrayList<Actor> actor = new ArrayList<Actor>();
        int[] dirs =
            { Location.AHEAD, 180 };

        for (Location loc : getLocationsInDirections(dirs))
        {
            Actor a = getGrid().get(loc);
            if (a != null)
            {
                actor.add(a);
            }
        }

        return actor;
    }

    /**
     * Finds the valid adjacent locations of this critter in different
     * directions.
     * @param directions - an array of directions (which are relative to the
     * current direction)
     * @return a set of valid locations that are neighbors of the current
     * location in the given directions
     */
    public ArrayList<Location> getLocationsInDirections(int[] directions)
    {
        ArrayList<Location> locs = new ArrayList<Location>();
        Grid gr = getGrid();
        Location loc = getLocation();

        for (int d : directions)
        {
            Location neighborLoc = loc.getAdjacentLocation(getDirection() + d);
            if (gr.isValid(neighborLoc))
            {
                locs.add(neighborLoc);
            }
        }
        return locs;
    }
}
```

3. RockHound

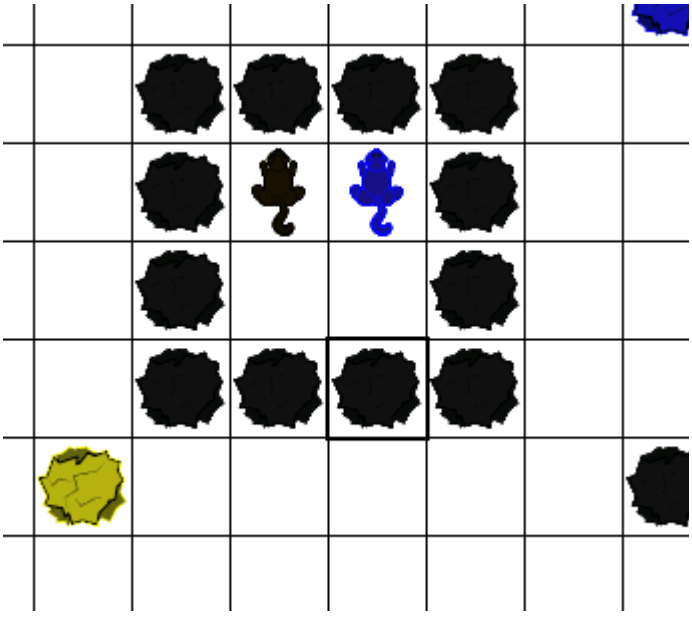
```
30  public class RockHound extends Critter
31  {
32
33      public void processActors(ArrayList<Actor> actors)
34      {
35          for (Actor a : actors)
36          {
37              if ( a instanceof Rock )
38              {
39                  a.removeSelfFromGrid();
40              }
41          }
42      }
43  }
44
```

4. BlusterCritter

```java
public class BlusterCritter extends Critter
{
    private static final double FACTOR = 0.05;
    private int courage;

    public BlusterCritter(int cour)
    {
        // Call the constructor of Actor.(Must be first)
        super();
        courage = cour;
    }

    // Get neighboring critter within two steps.
    public ArrayList<Actor> getActors()
    {
        ArrayList<Actor> actors = new ArrayList<Actor>();
        Grid<Actor> grid = getGrid();
        Location curLoc = getLocation();
        int row = curLoc.getRow();
        int column = curLoc.getCol();

        for ( int i = row - 2; i <= row + 2; i++ ) {
            for ( int j = column - 2; j <= column + 2; j++ ) {
                if ( i == row && j == column ) {
                    continue;
                }
                Location loc = new Location(i, j);
                if ( grid.isValid(loc) ) {
                    Actor a = grid.get(loc);
                    if ( a instanceof Critter ) {
                        actors.add(a);
                    }
                }
            }
        }
        return actors;
    }
```

```
70      /** If the number of critters within two steps is fewer than courage, it will brighten,
71       *  otherwise, it will darken.
72       */
73      public void processActors(ArrayList<Actor> actors)
74      {
75          Color c = getColor();
76          int red = c.getRed();
77          int green = c.getGreen();
78          int blue = c.getBlue();
79          System.out.println(courage);
80          System.out.println(actors.size());
81
82          if ( actors.size() < courage ) {
83              red = (int) (red * (1 + FACTOR));
84              green = (int) (green * (1 + FACTOR));
85              blue = (int) (blue * (1 + FACTOR));
86              if ( red > 255 ) {
87                  red = 255;
88              }
89              if ( green > 255 ) {
90                  green = 255;
91              }
92              if ( blue > 255 ) {
93                  blue = 255;
94              }
95
96          } else if ( actors.size() > courage ) {
97              red = (int) (red * (1 - FACTOR));
98              green = (int) (green * (1 - FACTOR));
99              blue = (int) (blue * (1 - FACTOR));
100         }
101
102         setColor(new Color(red, green, blue));
103     }
104 }
105
```

Result:



The BlusterCritter with courage 0 darkens while another with courage 2 brightens.

5. QuickCrab

```
32  public class QuickCrab extends CrabCritter
33  {
34      /** A QuickCrab moves to one of the two locations, randomly selected,
35       *  that are two spaces to its right or left,
36       *  if that location and the intervening location are both empty.
37       *  Otherwise, a QuickCrab moves like a CrabCritter.
38       */
39      public ArrayList<Location> getMoveLocations()
40      {
41          ArrayList<Location> locs = new ArrayList<Location>();
42          Grid grid = getGrid();
43          Location curLoc = getLocation();
44          int dir = getDirection();
45          Location locLeft = curLoc.getAdjacentLocation(dir + Location.LEFT);
46          Location locRight = curLoc.getAdjacentLocation(dir + Location.RIGHT);
47
48          if ( grid.isValid(locLeft) && grid.get(locLeft) == null )
49          {
50              Location locLeftTwo = locLeft.getAdjacentLocation(dir + Location.LEFT);
51              if ( grid.isValid(locLeftTwo) && grid.get(locLeftTwo) == null ) {
52                  locs.add(locLeftTwo);
53              } else {
54                  locs.add(locLeft);
55              }
56          }
57          if ( grid.isValid(locRight) && grid.get(locRight) == null )
58          {
59              Location locRightTwo = locRight.getAdjacentLocation(dir + Location.RIGHT);
60              if ( grid.isValid(locRightTwo) && grid.get(locRightTwo) == null ) {
61                  locs.add(locRightTwo);
62              } else {
63                  locs.add(locRight);
64              }
65          }
66          return locs;
67      }
68  }
69
```

6. KingCrab

```
32  public class KingCrab extends CrabCritter
33  {
34      /**
35       * A KingCrab causes each actor that it processes to move one location further away from the KingCrab.
36       * If the actor cannot move away, the KingCrab removes it from the grid.
37       */
38      public void processActors(ArrayList<Actor> actors)
39      {
40          Location loc = getLocation();
41          int row = loc.getRow();
42          int col = loc.getCol();
43          for (Actor a : actors)
44          {
45              Location aLoc = a.getLocation();
46              int aRow = aLoc.getRow();
47              int aCol = aLoc.getCol();
48              int newRow = row + 2 * (aRow - row);
49              int newCol = col + 2 * (aCol - col);
50              Location newLoc = new Location(newRow, newCol);
51              if ( getGrid().isValid(newLoc) ) {
52                  a.moveTo(newLoc);
53              } else {
54                  a.removeSelfFromGrid();
55              }
56          }
57      }
58  }
59
```