Part 4

Set 7

1.act(), getGrid(), getActors(), processActors(), getMoveLocations(), selectMoveLocation(), makeMove(), getNeighbors(), getEmptyAdjacentLocations(), get(), removeSelfFromGrid(), moveTo();

2.getActors(), processActors(), getMoveLocations(), selectMoveLocation(), makeMove();

3.Yes.Once the subclass behaviors different from the Critter, it may override the getActors method.

4.A critter may get the actors around and store them into a list, choose one of them to eat.

5.getMoveLocations(), selectMoveLocation(), moveTo();

First it must get locations which it can move, and choose one of them to make a move ,at last move to the choosed locations.

6.Actor has a default constructor and Crritter extends Actor.

Set 8

1.Because ChameleonCritter behavior different from Critter, it calls getLocation(), setDirection() and makeMove() to set up its behaviors.

2.super.makeMove() is call the makeMove() in Critter class, not the makeMove() in ChameleonCritter class.And this.makeMove() is call the makeMove() in ChameleonCritter class.

3.Flower flower = new Flower(getColor());

flower.putSelfInGrid(gr, loc);

Add two statements above in makeMove() can make the ChameleonCritter drop flowers in its old location when it moves.

4.Because it can use the list in the Critter class.

5.The Actor class.

6.The Critter can call the method getGrid() in the Actor class.

Set 9

1.The CrabCritter will eat flowers in the front, to the right-front, or to the left-front of it.It is the same behavior in Critter class.

2.Firstly it will find the actors around it and store them into a list.Secondly it will judege if there is flower in the front, to the right-front, or to the left-front of it, and eat it.It doesn't eat all neighboring actors, only flowers in the front, to the right-front and to the left-front of it.

3.Because it need to find the flower in the front, to the right-front and to the left-front of it.So it need getLocationsInDirections() to find whether there is flower.

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

5.Similarities: They both no leave anything after they move.
Differences: The CrabCritter only can turn left or right, but the Critter can turn eight directions.The CrabCritter can eat flower in the front, to the right-front and to the left-front of it, but the
Critter can't.

6.If a CrabCritter cannot move the left or the right, then it turns 90 degrees, randomly to the left or right.

7.CrabCritter class extends Critter class.In critter, proccessActors() only get actor that aren't rock or critter.So CrabCritter can't each other.

Exercise 1

```
/*
 Modify the processActors method in ChameleonCritter so that if the list of actors to process is empty, the color of the ChameleonCritter will darken (like a flower)
*/
/* Add following code into ChameleonCritter.java */
import java.awt.Color;
private static final double DARKENING_FACTOR = 0.05;
public void processActors(ArrayList<Actor> actors) {
	int n = actors.size();
	if (n == 0) {
		Color c = getColor();
		int red = (int) (c.getRed() * (1 - DARKENING_FACTOR));
		int green = (int) (c.getGreen() * (1 - DARKENING_FACTOR));
		int blue = (int) (c.getBlue() * (1 - DARKENING_FACTOR));
		setColor(new Color(red, green, blue));
		return;
	}
	int r = (int) (Math.random() * n);
	Actor other = actors.get(r);
	setColor(other.getColor());
}
```

Exercise 2

```
/*
```

Create a class called ChameleonKid that extends ChameleonCritter as modified in exercise 1. A ChameleonKid changes its color to the color of one of the actors immediately in front or behind. If there is no actor in either of these locations, then the ChameleonKid darkens like the modified ChameleonCritter.
*/

```
import info.gridworld.actor.Actor;
import info.gridworld.actor.Critter;
import info.gridworld.grid.Location;
import info.gridworld.grid.Grid;
import java.util.ArrayList;
import java.awt.Color;
public class ChameleonKid extends Critter {
	private static final double DARKENING_FACTOR = 0.05;
	/**
	  * A ChameleonKid changes its color to the color of one of the actors
	  * immediately in front or behind.
	  */
	public void processActors(ArrayList<Actor> actors) {
		int n = actors.size();
		if (n == 0) {
			Color c = getColor();
			int red = (int) (c.getRed() * (1 - DARKENING_FACTOR));
			int green = (int) (c.getGreen() * (1 - DARKENING_FACTOR));
			int blue = (int) (c.getBlue() * (1 - DARKENING_FACTOR));
			setColor(new Color(red, green, blue));
			return;
		}
		int r = (int) (Math.random() * n);
		Actor other = actors.get(r);
		setColor(other.getColor());
	}
	/* get actors that ahead and behind of it and store in list */
	public ArrayList<Actor> getActors() {
		ArrayList<Actor> actors = new ArrayList<Actor>();
		int[] dirs =
			{ Location.AHEAD, Location.HALF_CIRCLE };
		for (Location loc : getLocationsInDirections(dirs)) {
			Actor a = getGrid().get(loc);
			if (a != null) {
				actors.add(a);
			}
		}
		return actors;
	}
```

```
    /**
     * Finds the valid adjacent locations of this ChameleonKid 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;
    }
    /**
     * Turns towards the new location as it moves.
     */
    public void makeMove(Location loc) {
        setDirection(getLocation().getDirectionToward(loc));
        super.makeMove(loc);
    }
}
```

Exercise 3

```
/*
```

Create a class called RockHound that extends Critter. A RockHound gets the actors to be processed in the same way as a Critter. It removes any rocks in that list from the grid. A RockHound moves like a Critter.

```
*/
import info.gridworld.actor.Actor;
import info.gridworld.actor.Rock;
import info.gridworld.actor.Critter;
import info.gridworld.grid.Location;
import java.util.ArrayList;
/**
```

Create a class called RockHound that extends Critter. A RockHound gets the actors to be processed in the same way as a Critter. It removes any rocks in that list from the grid. A RockHound moves like a Critter.

```
 */
public class RockHound extends Critter {
	public void processActors(ArrayList<Actor> actors) {
		for (Actor a : actors) {
			if (a instanceof Rock) {
			a.removeSelfFromGrid();
			}
		}
	}
	/**
	 * Turns towards the new location as it moves.
	 */
	public void makeMove(Location loc) {
		setDirection(getLocation().getDirectionToward(loc));
		super.makeMove(loc);
	}
}
```

Exercise 4

```
/*
```

Create a class BlusterCritter that extends Critter. A BlusterCritter looks at all of the neighbors within two steps of its current location. (For a BlusterCritter not near an edge, this includes 24 locations). It counts the number of critters in those locations. If there are fewer than c critters, the BlusterCritter's color gets brighter (color values increase). If there are c or more critters, the BlusterCritter's color darkens (color values decrease). Here, c is a value that indicates the courage of the critter. It should be set in the constructor.

```
*/

import info.gridworld.actor.Actor;
import info.gridworld.actor.Critter;
import info.gridworld.grid.Location;
import java.util.ArrayList;
import java.awt.Color;
public class BlusterCritter extends Critter {
	private int cc;
	private static final int CH = 5;
	private static final int GR = 128;
	private static final int MA = 255;
	public BlusterCritter(Color bccolor, int c) {
		setColor(bccolor);
		cc =    c;
	}
	/* get all of the neighbors within two steps of its current location */
	public ArrayList<Actor> getActors() {
```

```
        ArrayList<Actor> actors = new ArrayList<Actor>();
        Location loc = getLocation();
        for(int r = loc.getRow() - 2; r <= loc.getRow() + 2; r++ ) {
            for(int c = loc.getCol() - 2; c <= loc.getCol() + 2; c++) {
                Location tempLoc = new Location(r,c);
                if(getGrid().isValid(tempLoc)) {
                    Actor a = getGrid().get(tempLoc);
                    if(a != null && a != this) {
                        actors.add(a);
                    }
                }
            }
        }
        return actors;
    }
    /* Count all of the neighbors within two steps of its current location */
    /* If there are fewer than c critters, the BlusterCritter's color gets brighter (color values increase).*/
    /* If there are c or more critters, the BlusterCritter's color darkens (color values decrease).*/
    public void processActors(ArrayList<Actor> actors) {
        int count = 0;
        for (Actor a: actors) {
            if (a instanceof Critter) {
                count++;
            }
        }
        /* make the color-value have a initial value */
        int red = 0;
        int green = 0;
        int blue = 0;
        if (count < cc) {
        /* make a size of color-value and set their values in two cases */
            if (red > 0 && red < MA && green > 0 && green < MA && blue > 0 && blue < MA) {
                red = GR + (cc - count)*CH;
                green = GR + (cc - count)*CH;
                blue = GR + (cc- count)*CH;
            }
        } else {
            if (red > 0 && red < MA && green > 0 && green < MA && blue > 0 && blue < MA) {
                red = GR - (count - cc)*CH;
                green = GR - (count - cc)*CH;
                blue= GR - (count - cc)*CH;
```

```
                }
            }
            setColor(new Color(red, green, blue));
        }


        /**
         * Turns towards the new location as it moves.
         */
        public void makeMove(Location loc) {
            setDirection(getLocation().getDirectionToward(loc));
            super.makeMove(loc);
        }
}
```

Exercise 5

```
/*
Create a class QuickCrab that extends CrabCritter. A QuickCrab processes actors the same way a CrabCritter does. A QuickCrab moves to one of the two locations, randomly selected, that are two spaces to its right or left, if that location and the intervening location are both empty. Otherwise, a QuickCrab moves like a CrabCritter.
*/
import info.gridworld.actor.Actor;
import info.gridworld.actor.Critter;
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;
import java.awt.Color;
import java.util.ArrayList;
public class QuickCrab extends Critter {
    private static final double HALF = 0.5;
    public QuickCrab() {
        setColor(Color.RED);
    }
    /**
     * A crab gets the actors in the three locations immediately in front, to its
     * front-right and to its front-left
     * @return a list of actors occupying these locations
     */
    public ArrayList<Actor> getActors() {
        ArrayList<Actor> actors = new ArrayList<Actor>();
        int[] dirs =
            { Location.AHEAD, Location.HALF_LEFT, Location.HALF_RIGHT };
        for (Location loc : getLocationsInDirections(dirs)) {
            Actor a = getGrid().get(loc);
```

```
            if (a != null) {
                actors.add(a);
            }
        }
        return actors;
    }
    /**
     * select one of two locations on the left or right
     * @return list of empty locations immediately to the right and to the left
     */
    public ArrayList<Location> getMoveLocations() {
        ArrayList<Location> locs = new ArrayList<Location>();
        quickMove(locs,getDirection() + Location.LEFT);
        quickMove(locs,getDirection() + Location.RIGHT);
        if (locs.size() == 0) {
            return locs;
        }
        int[] dirs =
            { Location.LEFT, Location.RIGHT };
        for (Location loc : getLocationsInDirections(dirs)) {
            if (getGrid().get(loc) == null) {
                locs.add(loc);
            }
        }
        return locs;
    }
    /* check if the QucikCrab canmake a    qickMove */
    private void quickMove(ArrayList<Location> locs, int dir) {
        Grid gr = getGrid();
        Location loc = getLocation();
        Location first = loc.getAdjacentLocation(dir);
        /* check if the first location is valid and null that the QucikCrab can move */
        if(gr.isValid(first) && gr.get(first) == null) {
            Location second = first.getAdjacentLocation(dir);
            /* check if the second location is valid and null the QucikCrab can move */
            if(gr.isValid(second) && gr.get(second)== null) {
                locs.add(second);
            }
        }
    }
    /**
     * If the crab critter doesn't move, it randomly turns left or right.
     */
    public void makeMove(Location loc) {
```

```
            if (loc.equals(getLocation())) {
                double r = Math.random();
                int angle;
                if (r < HALF) {
                    angle = Location.LEFT;
                } else {
                    angle = Location.RIGHT;
                }
                setDirection(getDirection() + angle);
            } else {
                super.makeMove(loc);
            }
        }
        /**
         * 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;
        }
}
```

Exercise 6

```
/*
Create a class KingCrab that extends CrabCritter. A KingCrab gets the actors to be processed in the same way a CrabCritter does. A KingCrab causes each actor that it processes to move one location further away from the KingCrab. If the actor cannot move away, the KingCrab removes it from the grid. When the KingCrab has completed processing the actors, it moves like a CrabCritter.
*/
import info.gridworld.actor.Actor;
```

```
import info.gridworld.actor.Critter;
import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;
import java.awt.Color;
import java.util.ArrayList;
public class KingCrab extends Critter {
	private static final double HALF = 0.5;
	public KingCrab() {
		setColor(Color.RED);
	}
	/*
	* Computes the distance between two given locations.
	*/
	public double getDistance(Location a, Location b) {
		double xx = Math.abs(a.getRow() - b.getRow());
		double yy = Math.abs(a.getCol() - b.getCol());
		double dist = Math.sqrt(xx*xx + yy*yy) + HALF;
		return (int)Math.floor(dist);
	}
	/*
	* Judge if the actor can move one location further away from the KingCrab.
	*/
	private boolean moveAway(Actor a) {
		ArrayList<Location> locs =
		getGrid().getEmptyAdjacentLocations(a.getLocation());
		for(Location loc:locs) {
			if(getDistance(getLocation(), loc) > 1) {
				a.moveTo(loc);
				return true;
			}
		}
		return false;
	}
	/*
	* The actor have to move one location further away from the KingCrab.
	* If there is no location for it to further away, it'll removed itself from grid.
	*/
	public void processActors(ArrayList<Actor> actors) {
		for (Actor a : actors) {
			if (!moveAway(a)) {
				a.removeSelfFromGrid();
			}
		}
	}
```

```
/**
 * A crab gets the actors in the three locations immediately in front, to its
 * front-right and to its front-left
 * @return a list of actors occupying these locations
 */
public ArrayList<Actor> getActors() {
	ArrayList<Actor> actors = new ArrayList<Actor>();
	int[] dirs =
		{ Location.AHEAD, Location.HALF_LEFT, Location.HALF_RIGHT };
	for (Location loc : getLocationsInDirections(dirs)) {
		Actor a = getGrid().get(loc);
		if (a != null) {
			actors.add(a);
		}
	}
	return actors;
}
/**
 * @return list of empty locations immediately to the right and to the left
 */
public ArrayList<Location> getMoveLocations() {
	ArrayList<Location> locs = new ArrayList<Location>();
	int[] dirs =
		{ Location.LEFT, Location.RIGHT };
	for (Location loc : getLocationsInDirections(dirs)) {
		if (getGrid().get(loc) == null) {
			locs.add(loc);
		}
	}
	return locs;
}
/**
 * If the crab critter doesn't move, it randomly turns left or right.
 */
public void makeMove(Location loc) {
	if (loc.equals(getLocation())) {
		double r = Math.random();
		int angle;
		if (r < HALF) {
			angle = Location.LEFT;
		} else {
			angle = Location.RIGHT;
		}
		setDirection(getDirection() + angle);
```

```
            } else {
                super.makeMove(loc);
            }
        }
        /**
         * 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;
        }
    }
```