Grid World Part 3 - CS 30

By Eric George

**Part 3**

**Set 3**

1. You can access the row value for loc1 by using the getRow method on itself i.e. loc1.getRow().
2. The value of b after the statement: boolean b = loc1.equals(loc2) is false.
3. After the statement, loc3 has a row value of 4 and a column value of 4.
4. The value of dir after the statement is 135;
5. The getAdjacentLocation method knows which adjacent location to return based on the int argument.

**Set 4**

1. You can obtain a count of the objects in a grid by first using the getOccupiedAdjacentLocations method on the grid, and then using the size method on the resulting ArrayList. For empty locations, you can use the getEmptyAdjacentLocations method instead of the getOccupiedAdjacentLocations.
2. You can check if the location (10,10) is valid by making a location object with those coordinates and using the isValid method on it.
3. No code is supplied in the methods because due to the nature of an interface, all the methods must be implemented in the classes that use the interface. Therefore, the implementations for these methods can be found in the class that implement or use the interface.
4. No. The major pro for ArrayList is that its size is dynamic, which is necessary due to the unknown nature of the amount of objects to return. As well, the major con, that it can only store objects, is not relevant in the situation as there are only objects to return.

**Set 5**

1. Every Actor has a direction, colour, and location.
2. A newly constructed actor has its directions set north and its colour set to blue from awt.Color.
3. An actor has defaults that do not need to be modified with every extension of it, such as its act method. Thus, it is not necessarily better to make it an interface.
4. An actor can not put itself into a grid twice without first removing itself and it cannot remove itself from a grid twice without putting itself in it first as it will cause an exception due to violating the conditions required for the method to run correctly.
5. An actor can turn 90 degrees to the right by setting the actors direction to its previous direction plus 90 i.e. setDirection(this.getDirection() + 90)

**Set 6**

1. The statement: if (gr == null) return false.
2. The statement: return (neighbor == null) || (neighbor instanceof Flower).
3. The isValid method is used to determine whether the next grid square is in grid space and the get method is used to determine which Actor subclass is in the adjacent grid space.
4. The getAdjacentLocation method is used to determine the square that the actor is trying to move to.
5. The getLocation method, getDirection method, and the getGrid method.
6. The bug removes itself from the grid using the removeSelfFromGrid method.
7. The variable loc is needed to keep the reference of where the bug was after it moved to place a Flower object there.
8. The flower has the same color as the bug because when the flower object is created, the color of the bug is passed as an argument into the flowers constructor.
9. A bug does not inherently place a flower into its past location with the removeSelfFromGrid. However, when the bug removes itself as a result of the move method, a flower will be placed into its past location due to the move method.
10. The following statements places the flower into the bugs previous location:

Flower flower = new Flower(getColor());

flower.putSelfInGrid(gr, loc);

1. The bug should call the turn method four times

**Group Activity Members:(Eric George, Jake Yeo)**

1. Specify:
   1. The jumper will remove itself from the grid.
   2. The jumper will remove itself from the grid.
   3. The jumper will remove itself from the grid.
   4. The jumper will remove itself from the grid.
   5. If either jumper is in front of the other by one or two cells, there are two possible outcomes, that depend on which jumper acts first. If the jumper behind acts first only one jumper will remain as the behind jumper will be removed from the grid. If the ahead jumper acts first, then it will appear as if the jumpers are moving in unison. Any encounter otherwise does not affect the jumpers.
   6. How should the jumper react to eternal flowers? Should the jumper be able to ignore the precedence of movement?
2. Design:
   1. Jumper should extend the actor class.
   2. The bug class is similar to the jumper in that many of its methods are similar to ones required for the jumper class.
   3. A default constructor is necessary but there are no necessary parameters for the constructor.
   4. A method that should be overridden by the jumper class is the act method.
   5. A method to determine whether the jumper object can move and a method to move the jumper object should be added.
   6. The jumper should be tested on the edge of the grid facing towards it, one cell away from the grid, before a rock, before a flower, before the cell before a rock, and before the cell before a flower.

BlueBugBased code

@Override

public void act() {

this.turnToFlower();

if (!this.canMove()) {

//flower in front?? never occurs because it can move when flower there

removeSelfFromGrid();

} else {

// Grid<Actor> grid = this.getGrid();

//// ArrayList<Actor> neighbours = grid.getNeighbors(this.getLocation());

//// Iterator neighborI = neighbours.iterator();

//// while(neighborI.hasNext()) {

////// Actor a = (Actor) neighborI.next();

////// if(neighborI.next() instanceof Flower){

//////

////// }

//// }

// ArrayList<Location> locList = grid.getOccupiedLocations();

// //getting closest flower

// Iterator i = locList.iterator();

//// Location locComp = (Location) i.next();

//// Location locNext = null;

//// int rCurr = this.getLocation().getRow();

//// int cCurr = this.getLocation().getCol();

//// boolean flowerFound = false;

// int dir = 0;

// for (; i.hasNext() /\*&& !flowerFound\*/;) {

// Location l = (Location) i.next();

// if (grid.get(l) instanceof Flower) {

// dir = this.getLocation().getDirectionToward(l);

// if (this.getLocation().getAdjacentLocation(dir).equals(grid.get(l))) {

// grid.get(l).removeSelfFromGrid();

//// flowerFound = false;

// } else {

//// flowerFound = true;

// }

// }

// }

// this.setDirection(dir);

this.move();

// locComp = (Location) i.next();

// int r = locComp.getRow();

// int c = locComp.getCol();

// if ()

//

}

// randomTurn();

// if (!canMove()) {

// return;

// }

// move();

//

// Location loc = getLocation();

// Grid<Actor> grid = getGrid();

// ArrayList<Actor> neighbors = grid.getNeighbors(loc);

//

// for (int i=0; i<neighbors.size(); i++) {

// Actor actor = neighbors.get(i);

//// System.out.println(actor);

// }

//

// for (Actor actor: neighbors) {

// if (actor instanceof Flower) {

//

// }

// }

}

/\*

\* Turns left or right 45 degrees at random.

\*/

public void randomTurn() {

if (Math.random() < 0.5) {

turn(-45);

} else {

turn(45);

}

}

public void turnToFlower() {

Grid<Actor> grid = this.getGrid();

ArrayList<Location> locList = grid.getOccupiedLocations();

//getting closest flower

Iterator i = locList.iterator();

// Location locComp = (Location) i.next();

// Location locNext = null;

// int rCurr = this.getLocation().getRow();

// int cCurr = this.getLocation().getCol();

// boolean flowerFound = false;

int dir = 0;

for (; i.hasNext() /\*&& !flowerFound\*/;) {

Location l = (Location) i.next();

if (grid.get(l) instanceof Flower) {

dir = this.getLocation().getDirectionToward(l);

if (this.getLocation().getAdjacentLocation(dir).equals(grid.get(l))) {

grid.get(l).removeSelfFromGrid();

// flowerFound = false;

}

// else {

//// flowerFound = true;

// }

}

}

this.setDirection(dir);

}

/\*

\* Turns the given number of degrees.

\*/

public void turn(int degrees) {

setDirection(getDirection() + degrees);

}

@Override

public void move() {

Grid<Actor> gr = getGrid();

if (gr == null) {

return;

}

Location loc = getLocation();

Location next = loc.getAdjacentLocation(getDirection());

if (gr.isValid(next)) {

this.moveTo(next);

} else {

removeSelfFromGrid();

}

}

Jumper code

public Jumper() {

}

@Override

public void act() {

if (this.canJump()) {

this.jump();

}else {

this.removeSelfFromGrid();

// this.setDirection(this.getDirection() + 45);

}

}

public boolean canJump() {

Grid<Actor> gr = this.getGrid();

if (gr == null) {

return false;

}

Location loc = this.getLocation();

Location next = loc.getAdjacentLocation(getDirection()).getAdjacentLocation(getDirection());

if (!gr.isValid(next)) {

return false;

}

if (!(gr.get(loc.getAdjacentLocation(getDirection())) instanceof Rock)){

return false;

}

Actor neighbor = gr.get(next);

return (neighbor == null) || (neighbor instanceof Flower);

}

public void jump() {

Grid<Actor> gr = this.getGrid();

if (gr == null) {

return;

}

Location loc = this.getLocation();

Location next = loc.getAdjacentLocation(getDirection()).getAdjacentLocation(getDirection());

if (gr.isValid(next)) {

this.moveTo(next);

} else {

removeSelfFromGrid();

}

}