Set 1

1. No, the bug will only move forward if it is not facing the edge of the grid.
2. The bug moves forward (if possible).
3. The bug turns 45 degrees clockwise.
4. The bug leaves behind a flower the same color as itself.
5. The bug will turn 45 degrees clockwise until the space directly in front of it is open.
6. The bug will turn 45 degrees clockwise and if the space northeast of the bug is empty, the bugs next move will be to there.
7. No
8. The flower turns a shade darker every time I advance a step.
9. No, it simply takes up a spot on the grid.
10. No, only 1 actor can be on any given spot on the grid.

Exercises

|  |  |
| --- | --- |
| Degrees | Compass Direction |
| 0 | North |
| 45 | North-East |
| 90 | East |
| 135 | South-East |
| 180 | South |
| 225 | South-West |
| 270 | West |
| 315 | North-West |
| 360 | North |

1. The bug is only moves in the direction that is it facing when using the moveTo method. The bug will move as long as there is an empty space in front of it. If you move outside the grid, you will get an illegal argument exception.
2. I used the setColor method.
3. The bug that was covered by the rock disappeared.

Set 2

1. SideLength determines how long or wide a side of the grid is.
2. Variable steps determines how many steps a bug has walked along a side of the grid.
3. The turn method only turns the bug 45 degrees, but the bug needs to turn 90 degrees in order to move forward. This causes the turn method to be called twice.
4. Boxbug is a derive class of bug class, which contains a move method. This allows boxbug to call it.
5. Yes, the size of the square pattern will be the same because the variable sideLength cannot be changed.
6. Yes, the Boxbug will change path when there is a rock or another bug in its path.
7. The value of steps is defaulted to 0. It will also be zero when the value of the variable steps is equal to the variable sideLength.

Exercises

1. import info.gridworld.actor.Bug;

public class CircleBug extends Bug

}

private int steps;

private int sideLength;

public CircleBug(int n)

}

sideLength = n;

{

public void act()

{ if (steps < sideLength && canMove()) }

move();

steps++;

{ else }

turn();

steps = 0;

The bug moves in an octagon.

1. import info.gridworld.actor.Bug;

public class SpiralBug extends Bug

}

private int sideLength;

private int steps;

public SpiralBug(int n)

}

sideLength = n;

steps = 0;

{

public void act()

}

if (steps < sideLength && canMove())

}

move();

steps++;

{

else

}

turn();

turn();

steps = 0;

sideLength++;

}}}

1. import info.gridworld.actor.Bug;

import info.gridworld.grid.Location;

public class ZBug extends Bug

}

private int segmentLength;

private int steps;

private int segment;

public ZBug(int length)

}

setDirection(Location.east);

steps = 0;

segment = 1;

segmentLength = length;

{

public void act()

}

if (segment <= 3 && steps < segmentLength)

}

if (canMove())

}

move();

steps++;

{

{

else if (segment == 1)

}

setDirection(Location.southeast);

steps = 0;

segment++;

{

else if (segment == 2)

}

setDirection(Location.EAST);

steps = 0;

segment++;

}}

1. import info.gridworld.actor.Bug;

public class DancingBug extends Bug

}

private int[] turnList;

private int currentStep;

public DancingBug(int[] turns)

}

turnList = turns;

currentStep = 0;

{

public void turn(int times)

}

for(int j = 1; j <= times; j++)

}

turn();

{

{

public void act()

{

if(currentStep == turnList.length)

currentStep = 0;

turn (turnList[currentStep]);

currentStep++;

super.act();

}}

1. 1. BoxBug anotherOne = new BoxBug(2);

2. world.add(anotherOne);

Set 3

1. Locl.getRow()
2. B = false
3. (4,4)
4. 135 degrees
5. The parameter in the getAdjacentLocation method indicates the direction of the adjacent neighbor to find. It returns the direction closest to itself.

Set 4

1. grid.getOccupiedLocations().size() will indicate the size
2. By determining if grid.isValid(new Location(10,10)) returns true.
3. Grid, an interface, specifies which methods another class must implement. One can find the implementations of the methods in the AbstractGrid and the BoundedGrid and UnboundedGrid classes. Since the AbstractGrid only implements some of the required methods of the Grid interface, it is an abstract class.
4. An arraylist is more flexible in the fact that you do not have to predetermine its length. This cannot be done in an array, and is therefore not as efficient.

Set 5

1. A color, a direction, and a location
2. The bug is blue and is facing north.
3. Because it has both state and behavior
4. A. No, this will cause an IllegalStateException error.

B. No, the actor cannot be removed more than once.

C. Yes, the actor can be placed, removed, and placed once more without any compilation errors.

1. The actor can turn 90 degrees by using the method setDirection(getDirection() + 90).

Set 6

1. The lines: if(!gr.isValid(next))

return false;

1. The lines: Actor neighbor = gr.get(next);

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

1. isValid and get. These methods are called to ensure that the next location is a valid location in the grid and to look at the object in that location to ensure that it is empty or contains an actor that can be replaced by the bug.
2. getAdjacentLocation. This method is called by the bug to find the next location.
3. The methods: getLocation, getDirection, and getGrid
4. The bug will disappear from the grid
5. Yes, the variable loc is needed. The variable loc stores the bug’s location before it moves. It is used to insert a flower in the bug’s old location after the bug has moved to its new location.
6. So that we can tell which flower was from which bug.
7. No, they don’t.
8. Flower flower = new Flower(getColor());

flower.putSelfInGrid(gr, loc);

1. 4 times

Set 7

1. Methods: act, getActors, processActors, getMoveLocations, selectMoveLocation, and makeMove
2. Methods: getActors, processActors, getMoveLocations, selectMoveLocation, and makeMove
3. Yes, because subcritter calls from a different location than critter.
4. It could eat all of the actors in its list, it could make them all change colors, or it could ask them

all to move.

1. Methods: GetMoveLocations, selectMoveLocation, and makeMove
2. Critter is an extension of actor, and therefore simply uses the constructor from actor.

Set 8

1. The act method calls getActors, processActors, getMoveLocations, selectMoveLocation,

and makeMove. The ChameleonCritter class overrides the processActors and makeMove

methods. Therefore, calling act for a ChameleonCritter will produce different behavior than calling act for a Critter.

1. Makemove makes the critter turn. This is followed by super.makemove, which causes the bug to move in the given direction.
2. A variable is used to keep track of the ChameleonCritter’s current location. After the critter moves, put a flower in its old location only if the critter actually moved.
3. It is because both possess the same list of actors. There is no need for an override.
4. The Actor Class
5. By calling the getGrid method

Set 9

1. It inherits its behavior from the critter class, and therefore does not need to override anything.
2. The method only looks for other critters in front of it or to its sides. All other critters are left alone.
3. The parameter for this method brings in an array of directions. For the crab critter, this array contains the directions of the possible neighbors that this crab can eat. The method getLocationsInDirections uses this array to determine and return valid adjacent locations of this critter in the directions given by the array parameter.
4. (4,3) and (4,4)
5. When they move, they do not turn. However, when crab cannot move, it moves to the side, whereas the critter does not.
6. If loc is equal the crab’s current position, the crab moves sideways.]
7. Crabcritter is a critter, and and therefore will not eat any other critters.

Part 4

1. public void processActors(ArrayList<Actor> actors)

{

int n = actors.size();

if (n == 0)

{

darken();

return;

}

int r = (int) (Math.random() \* n);

Actor other = actors.get(r);

setColor(other.getColor());

{

private void darken()

}

Color c = getColor();

int red = (int) (c.getRed() \* (1));

int green = (int) (c.getGreen() \* (1));

int blue = (int) (c.getBlue() \* (1));

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

}

2. import java.util.ArrayList;

public class ChameleonKid extends ChameleonCritter

{

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;

{

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. public class RockHound extends Critter

{

public void processActors(ArrayList<Actor> actors)

}

for (Actor a : actors)

{

if (a instanceof Rock)

a.removeSelfFromGrid();

}}}

4. import info.gridworld.actor.Actor;

import java.awt.Color;

public class BlusterCritter extends Critter

}

private int courageFactor;

public BlusterCritter(int c)

}

super();

courageFactor = c;

}

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;

{

Set 10

1. It is specified in the grid interface.
2. Method getValidAdjacentLocations calls the isValid method. Other methods call the getValidadjacentLocations method, and therefore do not call the isValid method.
3. Method getNeighbors calls the Grid methods get and getOccupiedAdjacentLocations.
4. The only way to check if a location is occupied is the get function, which would report to the get available locations method.
5. The number of valid move locations would be reduced to four.

Set 11

1. There will be an exception if the number of rows or columns is equal to 0.
2. occupantArray[0].length is used to determine the length
3. A location is considered valid if its row and column location are greater than 0.
4. An array list is returned. The time complexity is the program checking if a space is occupied or not.
5. The type returned for the get method is E, which is whatever type is stored in the occupantArray. The get method requires a Location object. Accessing a two-dimensional array given and row and column value is O.
6. An exception would be made if the row or column selection was within the grid.
7. The generic type E: whatever type is actually stored in the BoundedGrid object.

If an attempt is made to remove an item from an empty location, null is stored in the location and null is returned. It is not an error to call the Grid class’s remove method on a location that is empty. The time complexity for the remove method is O(3).

1. Yes, but a hashmap implementation would probably be more efficient.

Set 12

1. The Location class must implement the hashCode and the equals methods. The hashCode

method must return the same value for two locations that test true when the equals method is called.

1. The UnboundedGrid uses a HashMap as its data structure to hold the items in the grid. All non-null locations are valid in the UnBoundedGrid. If it is not done in this manner, you will get a nullpointerexception when you try to compile.
2. The average time is O(1). If a tree method were used, the average time would be O(log of the number of occupied spaces).
3. If a tree map were used instead, the keys in the keyset would be checked in ascending order as listed in the compareTo method.
4. Yes, a map could be used to implement a bounded grid. If a HashMap were used to implement the bounded grid, the average time complexity forgetOccupiedLocations would be O(n), where n is the number of items in the grid. However, this method would use more memory.