Thursday, April 04, 2013
7:49 AM

'omputer Science GridWorld Quiz Review Name: ____________________

1. `Location` objects can be compared to each other and have a concept of sequencing. This is provided by the `compareTo` method of the `Location` class. Suppose a `MoveLowCritter` is to behave exactly like a `Critter`, except that it will move to the *smallest* adjacent location that is not blocked by another `Actor`. To achieve this behavior, which method should be overridden?

   a) `getActors`
   b) `processActors`
   c) `getMoveLocations`
   d) `selectMoveLocation`
   e) `makeMove`

2. Consider the following code segment.

```
Grid<Actor> grid = new BoundedGrid<Actor>(10, 10);

Bug bug1 = new Bug();
bug1.setDirection(Location.EAST);
bug1.putSelfInGrid(grid, new Location(3, 5));

Bug bug2 = new Bug();
bug2.setDirection(Location.SOUTH);
bug2.putSelfInGrid(grid, new Location(2, 6));

bug1.act();
bug2.act();
```

   What is the result of executing the code segment?

   a) `bug1` and `bug2` will both occupy location (3, 6).
   b) `bug1` will move into location (3, 6) and `bug2` will turn.
   c) `bug1` will move into location (3, 6) and `bug2` will do nothing.
   d) `bug1` will move into location (3, 6) and `bug2` will throw an exception.
   e) `bug1` will move into location (3, 6) but will be removed when `bug2` moves into the same location, (3, 6).

3. What is the result of the subsequent calls to `act` after a `BoxBug` encounters the edge of a bounded grid?
   a) The `BoxBug` stops tracing on the screen
   b) The `BoxBug` continues tracing past the edge of the grid, even though it will not be visible
   c) The `BoxBug` turns 90° clockwise and then stops tracing on the screen
   d) The `BoxBug` turns 90° clockwise and continues tracing even though the resulting trace may not be a perfect box shape
   e) A run-time error occurs.

4. A `RockingCritter` replaces each adjacent `Flower` with a `Rock` and then moves like a regular `Critter`. The following three implementations have been proposed.

I.

```
public class RockingCritter extends Critter {
    public ArrayList<Actor> getActors() {
        ArrayList<Actor> actors = new ArrayList<Actor>();
        for (Actor a : getGrid().getNeighbors(getLocation())) {
            if (a instanceof Flower)
                actors.add(a);
            }
            return actors;
        }

    public void processActors(ArrayList<Actor> actors) {
        for (Actor a : actors) {
            Location loc = a.getLocation();
            a.removeSelfFromGrid();
            Rock r = new Rock();
            r.putSelfInGrid(getGrid(), loc);
        }
    }
}
```

← makes an empty list

add all neighboring flowers to the list

remove the flower

add a rock

II.

```
public class RockingCritter extends Critter {
    public void processActors(ArrayList<Actor> actors) {
        for (Actor a : actors) {
            if (a instanceof Flower) {
                Location loc = a.getLocation();
                a.removeSelfFromGrid();
                Rock r = new Rock();
                r.putSelfInGrid(getGrid(), loc);
            }
        }
    }
}
```

Critter.getActors() sets all neighboring actors.

changes flowers to rocks

step 5

~~III~~.

```
public class RockingCritter extends Critter {
    public void makeMove(Location loc) {
        for (Actor a : getGrid().getNeighbors(getLocation())) {
            if (a instanceof Flower) {
                Location rLoc = a.getLocation();
                a.removeSelfFromGrid();
                Rock r = new Rock();
                r.putSelfInGrid(getGrid(), rLoc);
            }
        }

        super.makeMove(loc);
    }
}
```

goes through all neighbors

changes Flowers to Rocks.

Problem! Too late

All the Flowers would be deleted by steps 1 & 2 by the time step 5 runs.

Which of the implementations is (are) correct?

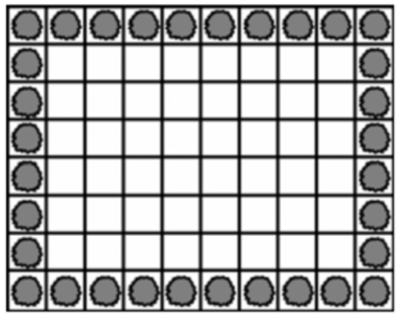
a) I only
b) II only
c) III only
d) I and II only
e) I, II, and III

5. Consider the following instance variable and method.

```
private BoundedGrid<Actor> grid;

public void placeRock(int row, int col) {
     new Rock().putSelfInGrid(grid, new Location(row, col));
}
```

Consider the problem of marking the borders of the grid with rocks as shown in the following picture.



The following code segments are proposed as solutions to the problem.

I.

```
for (int j = 0; j < grid.getNumRows(); j++) {
    placeRock(j, 0);
    placeRock(j, grid.getNumCols() - 1);
}
for (int k = 0; k < grid.getNumCols(); k++) {
    placeRock(0, k);
    placeRock(grid.getNumRows() - 1, k);
}
```

II.

```
for (int j = 0; j < grid.getNumRows() - 1; j++) {
    placeRock(j, 0);
    placeRock(j + 1, grid.getNumCols() - 1);
}
for (int k = 0; k < grid.getNumCols() - 1; k++) {
    placeRock(0, k + 1);
    placeRock(grid.getNumRows() - 1, k);
}
```

III.

```
for (int j = 0; j < grid.getNumRows(); j++) {
    for (int k = 0; k < grid.getNumCols(); k++) {
        Location loc = new Location(j, k);
        if (grid.getValidAdjacentLocations(loc).size() < 8)
            placeRock(j, k);
    }
}
```

Which of the code segments will correctly place rocks along the borders of the grid?

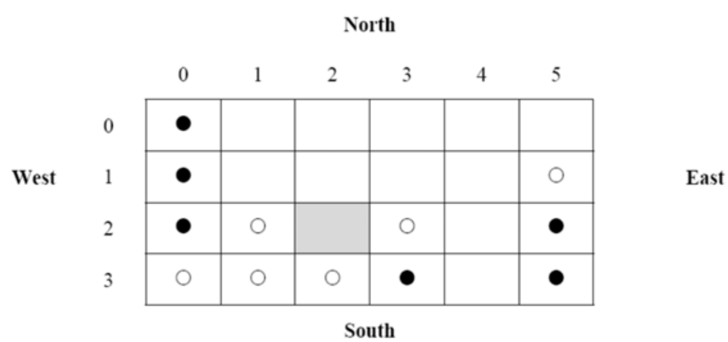
a) I only
b) II only
c) III only
d) I and III only
e) I, II, and III

ree Response Questions:

1. Consider using the `BoundedGrid` class from the GridWorld case study to model a game board.

DropGame is a two-player game that is played on a rectangular board. The players — designated as BLACK and WHITE — alternate, taking turns dropping a colored piece in a column. A dropped piece will fall down the chosen column until it comes to rest in the empty location with the largest row index. If the location for the **newly dropped** piece has **at least four** neighbors that match its color, the player that dropped this piece wins the game.

The diagram below shows a sample game board on which several moves have been made.



The following chart shows where a piece dropped in each column would land on this board.

Column	Location for Piece Dropped in the Column
0	No piece can be placed, since the column is full
1	(1, 1)
2	(2, 2)
3	(1, 3)
4	(3, 4)
5	(0, 5)

Note that a WHITE piece dropped in column 2 would land in the shaded cell at location (2, 2) and result in a win for WHITE because the four neighboring locations — (2, 1), (3, 1), (3, 2), and (2, 3) — contain WHITE pieces. This move is the only available winning move on the above game board.

The `Piece` class is defined as follows.

```
    public class Piece {
        /** @return the color of this Piece
        */
        public Color getColor() {
            /* implementation not shown */
        }

        // There may be instance variables, constructors, and methods that
        // are not shown.
}
```

An incomplete definition of the `DropGame` class is shown below. The class contains a private instance variable `theGrid` to refer to the `Grid` that represents the game board. Players will add `Piece` objects to this grid as they take turns. You will implement two methods for the `DropGame` class.

```
public class DropGame {
    private Grid<Piece> theGrid;

    /** @param column a column index in the grid
        * Precondition: 0 ≤ column < theGrid.getNumCols()
        * @return null if no empty locations in column;
        * otherwise, the empty location with the largest row index within
        * column
    */
    public Location dropLocationForColumn(int column) {
        /* implementation not shown */
    }

    /** @param column a column index in the grid
        * Precondition: 0 ≤ column < theGrid.getNumCols()
        * @param pieceColor the color of the piece to be dropped
        * @return true if dropping a piece of the given color into the
        * specified column matches color
        * with at least four neighbors;
        * false otherwise
    */
    public boolean dropMatchesNeighbors(int column, Color pieceColor) {
        /* to be implemented */
    }

    // There may be instance variables, constructors, and methods that are
    // not shown.
}
```

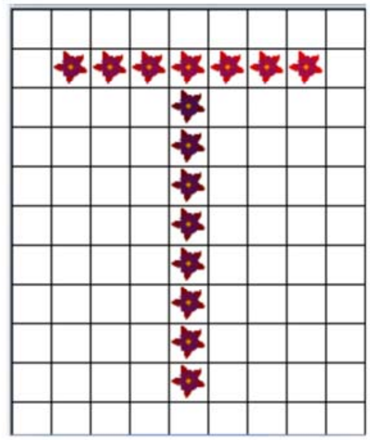
Write the `DropGame` method `dropMatchesNeighbors`, which returns `true` if dropping a piece of a given color into a specific column will match the color of at least four of its neighbors. The location to be checked for matches with its neighbors is the location identified by method `dropLocationForColumn`. If there are no empty locations in the column, `dropMatchesNeighbors` returns `false`.

Complete method `dropMatchesNeighbors` below.

```
/** @param column a column index in the grid
 *      Precondition: 0 ≤ column < theGrid.getNumCols()
 * @param pieceColor the color of the piece to be dropped
 * @return true if dropping a piece of the given color into the specified
 *              column matches color with at least four neighbors;
 *          false otherwise
 */
public boolean dropMatchesNeighbors(int column, Color pieceColor)
```

```
Location loc = dropLocationForColumn(column);
if (loc == null)
    return false;
ArrayList<Piece> pieces = theGrid.getNeighbors(loc);
int count = 0;
for ( Piece p : pieces) { // for each Piece p in pieces
    if (p.getColor().equals(pieceColor))
        count ++;
}
if (count >= 4)
    return true;
else
    return false;
```

2. In this question, you will consider implementing the design of a bug that produces a T-shaped pattern of flowers. You may assume that there are no other actors in the grid and that there is enough room for the T to be placed in the grid with a row of empty locations surrounding the area filled by the T. Here is a pattern in which the height of the T has length 9.



The bug starts at the top of the T in the center, facing south. When it finishes tracing the up-down part of the T, it jumps to the upper-left hand part of the T and traces the top. The top of the T is calculated to be 80% of the height of the T and made into an odd number if that result is even.

The declaration of `TBug` is as follows:

```
public class TBug extends Bug {
    private int steps;          // current number of steps that have been taken
    private int height;         // height of the T
    private int width;          // width of the top of the T
    private Location topLeft;   // location of the top left of the T

    public TBug(int h) {
        height = h;
        width = (int)(h * .8);
        if (width % 2 == 0)
            width++;
        setDirection(Location.SOUTH);
        steps = 0;
    }

    public void putSelfInGrid(Grid<Actor> g, Location l) {
        /* puts the bug in the grid and initializes topLeft location */
    }

    public void act() {
        /* to be implemented */
    }
}
```

```
if (steps < height) {
    move();
    steps++;
}
else if (steps == height) {
    moveTo(topLeft);  setDirection(Location.East); 90
    steps++;
}
else if (steps < (height+width+1)) {
    move();
    step++
}
else {
    removeSelfFromGrid();
}
```


Write the `TBug` `act` method. You may assume that the instance variables have been initialized prior to the first call of the `act` method.

move();
step++
}
else {

Write the TBug act method. You may assume that the instance variables have been initialized prior to the first call of the act method.

It starts in at the top-center point of the T and moves south. When it reaches the bottom of the T, it calls moveTo to move to the top left position of the T. It then moves west. When the pattern is completed, the TBug removes itself from the grid in the next call to the act method.

Complete method act below.

```
public void act()
```