University of Nebraska – Kearney
CSIT 150: Object Oriented Programming
Fall 2017
HW3 – Recursion

Due: 10/30/2017
100 points possible

Objectives

1. Understand how to implement an interface
2. Understand how a subclass inherits the data and methods from the superclass
3. Understand how overriding works
4. Understand how constructor works in inheritance
5. Understand how polymorphism works

Requirements

1. The homework is expected to be submitted on time. Late work will be accepted; however you lose 10% of your points for each late day.
2. You can get help from the instructor, the tutor, or other students. However, the homework must be finished by yourself, and you should indicate it on the paper if you got any help.

Multiple Choice: Questions 1-10 are multiple-choice questions. Each one has one correct answer.

1. [2 points] This is the part of a problem that can be solved without recursion:

   A. recursive case B. base case C. solvable case D. iterative case

2. [2 points] This is the part of a problem that is solved with recursion:

   A. recursive case B. base case C. solvable case D. iterative case

3. [2 points] When a method explicitly calls itself, this is known as _____ recursion.

   A. explicit B. implicit C. indirect D. direct

4. [2 points] When method A calls method B, which calls Method A, this is known as ____ recursion.

   A. explicit B. implicit C. indirect D. direct

5. [2 points] An iterative algorithm will usually run slower than an equivalent recursive algorithm.

   A. True B. False

6. [2 points] Some problems can be solved through recursion only.

   A. True B. False

7. [2 points] In the base case, a recursive method calls itself with a smaller version of the original problem.

   A. True B. False

8. [4 points] What will be the output form the following programming segment?

```
public class Ques8{
  public static void main(String[] args){
              showMe(0);
  }
  public static void showMe(int num) {
        if (num<10)
              showMe(num+1);
        else
              System.out.println (num);
  }
}
```

9. [2 points] What is the base case for the recursive method in question 8 above?

10. [2 points] Is question 8 an example of direct recursion or indirect recursion?

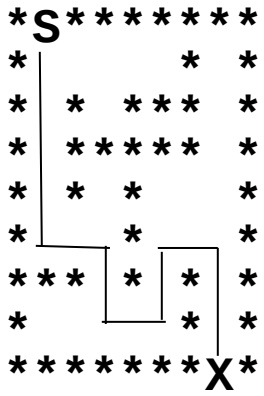
11. [4 points] Find and fix the error(s).

```
public class FindTheError{
  public static void main(String[] args){
              myMethod(0);
  }
  public static void myMethod(int num) {
        System.out.print(num+ " ");
        myMethod(num+1);
  }
}
```

Programming Problem: Robot Navigation Simulation

You are implementing a simulation of recursive robot that moves through the maze developed in HW2.

Picture a maze represented as an array of characters. A * symbol (asterisk) is used to represent a wall and a space is used to represent an open cell between walls. There are two holes in the outer wall, one labeled **S** for Start and one labeled **X** for Exit. Here is an example 9x9 maze with a traced trail through it:



There is a path from the Start (S) to the Exit (X). In this project you will write a variety of robots that will travel through a maze starting from the entrance and searching for the exit.

The MazeDriver program and the sample data file for a 9X9 maze is included with this program.

Data Input Files for this Project
Although you have one test maze file, you can see that the **`MazeDriver main`** method reads the name of a file that will be entered at runtime. The layout of each maze file will contain:

- In the first line: two integers (the number of rows and columns, respectively, in the maze)
- In the second line: two integers (the row and column locations, respectively, of the Start cell
- In the third line: two integers (the row and column locations, respectively, of the Exit cell
- Each line thereafter will contain characters appearing in one row of the maze.

The `Maze` Class – the same as HW2.
This is a "wrapper" class around a 2D array that contains the maze character data.
You *must* include all of the methods listed below, but you are free to add other private helper methods.

`public Maze (String filename)` The constructor will read the maze from the file.
`public int getRows()` returns the number of rows in the maze.
`public int getCols()` returns the number of columns in the maze.
`public int getStartRow()` returns the start cell's row.
`public int getStartCol()` returns the start cell's column.
`public int getExitRow()` returns the exit cell's row.
`public int getExitCol()` returns the exit cell's column.
`public char getCell(int row, int col)` returns the character stored at this cell.
`public boolean openCell(int row, int col)` returns true if the cell is open and the row and col values are valid for the given maze.
`public void setCell(int row, int col, char newCh)` sets the value stored in this cell.
`public String toString()` returns the maze as a string for output. The robot is shown as 'r', or by the direction it is facing – your choice. (This could depend on the kind of robot in the maze.)

The `Robot` Class – the same as HW2.
The **Robot** class is the super class of several robots. We will not instantiate the robot class directly, but will make instances of its subclasses. The **Robot** class needs the following public constructors and methods. You are, of course, free to add any private or protected variables and helper methods that you desire. (The private/protected designation should be used appropriately. Hint: I made a protected method that returns the array of Booleans that indicate the directions the robot can legally move. I also made a protected method that checks a given spot in the maze and returns true if it is empty.)

`public Robot (Maze maze)`
Creates a robot and places it at the start location of the maze. Note that the robot stores a reference to the maze, so we won't need it as a parameter for any other method. Because the robot has to control its own movement, it needs to know where it can go. This is different than our HW1 where the user controlled the movement of the X and O players, which were then placed onto the board.

Appropriate getters and setters for the private variables (current row, current col, robot name).

`public int chooseMoveDirection()`
This method does nothing in this class. Hmmm… if it doesn't do anything, what kind of method might it be? I can tell you that this is the method that will be overridden in each of the subclasses. In all the implementations, the robot plans the direction of its next move from its present location (North=0, South=1, West = 2, East=3). (The strategy of the direction choice depends on the implementation.)

`public boolean move(int direction)`
This method also does nothing in this class but is implemented in the subclasses. In all the implementations in this homework, the robot moves from its present location by one cell (North, South, East or West) in the direction specified, if the location in the move direction is open.

`public boolean solved()`
Returns `true` if and only if the `Robot` has arrived at the Exit cell.
The following classes *inherit* from the **`Robot`** class. They are specialized robot classes differing only in how they plan their move. You need to override the **`chooseMoveDirection`** and **`move`** and methods. Adding additional private variables and methods may be useful.

The `RandomRobot` Class – the same as HW2.
A `RandomRobot` determines which of the four directions it can legally move in, and then chooses one of them at random. Note that a `RandomRobot` may move back to where it came from.

NEW HW3 REQUIIREMENT: `FacingRobot` Class
The **`FacingRobot`** inherits from the **`Robot`**. The **`FacingRobot`** keeps track of the direction it is currently facing. By itself, the **`FacingRobot`** does not have a move strategy. That is, it is an abstract type of robot.

Representing the direction the robot is facing
Obviously, you will need some sort of data representation for the direction the robot is facing. There are many ways to do this, but one simple method would be to use a character for 'N', 'S', 'W', and 'E'

NEW HW3 REQUIIREMENT The `RightHandRobot` Class inherits from `FacingRobot`
The move and chooseMoveDirection methods are defined here, but the methods and variables used to keep track of the direction the robot is facing are defined in and inherited from **`FacingRobot.`**

NEW HW3 REQUIIREMENT: The `LookAheadRobot` Class
A **`LookAheadRobot`** has both a location and a direction which it is facing. A **`LookAheadRobot's`** goal is to search for the exit while always trying to **recursively** move as far as it can in one direction. On a move, it goes in the same direction until it cannot go any further. When planning its move direction, it will try to choose to go straight. If it cannot go straight, it will try to turn right. If it cannot turn right, it will try to turn left. As a last resort, it will turn around. Once the robot moves as far as it can in certain direction, its move is finished, and it stays facing in that direction. This kind of robot always needs to know the direction it is currently facing before it can decide which move to try first. It also inherits from the **`FacingRobot.`**

NEW HW3 REQUIIREMENT: Test the all of the robots (`RandomRobot`, `RightHandRobot`, `LookAheadRobot`, and `LeftHandRobot`, if you created it) on both the 9X9 testmaze.txt and the biggermaze.txt.

Bonus 10 points: Use an enumeration (from chapter 9) to keep track of the direction the **`FacingRobot`** robot is facing. (We will cover enumerations later, but they would be useful here.)

More Bonus: A `Robot that uses your own strategy`
Define your own strategy for deciding how to move through the maze.

Grading Policies
19. [52 points] Correctness: Your code should be able to function correctly according to the description of the above requirements.

20. [10 points] Style and Documentation: You code should meet the style and documentation requirements established for this course, including **JavaDoc comments to each class and each method; with heading documentation for each program.**