University of Nebraska – Kearney
CSIT 150: Object Oriented Programming
Fall 2017
HW2 – Inheritance

Due: 10/06/2017
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 key word indicates that a class inherits another class
A. implements B. specialized C. extends D. protected

2. [2 points] Everything defined in the superclass will be inherited by its subclasses as long as the thing has an access modifier of public or protected. Is this statement true or false?
A. True B. False

3. [2 points] A super() statement will be put as the first statement of the constructor of a subclass by the compiler if this subclass has no explicit super call. Is this statement true or false?
A. True B. False

4. [2 points] A superclass can have more than one subclass, however a subclass can have only one superclass. Is this statement true or false?
A. True B. False

5. [2 points] Assume that we have classes A, B, and C. Is it possible that class B is the subclass of class A and is also the superclass of class C?
A. Yes B. No

6. [2 points] If a method in a subclass and a method in the superclass have the same name but different number of parameters, we call it:
A. overloading B. overriding C. composition D. an error

7. [2 points] Overriding can happen both between a superclass and a subclass and in the same class. Is this statement true or false?
A. True B. False

8. [2 points] Assume that classes A and B are in the same package and are unrelated to each other. If class A wants to prevent a field from being accessible to class B, which of the following access modifiers can be used?

A. public B. private C. protected D. No modifier E. Both B and C

9. [2 points] By default, fields in an interface are

A. final B. static C. not allowed D. Both A and B

10. [2 points] Abstract classes cannot:

A. be used as superclasses B. have abstract methods C. be instantiated D. have fields

11. [2 points] If a superclass is abstract, then its subclass must implement all of the abstract methods in the superclass. Is this statement true or false?

A. true B. false

12. [2 points] Assume `Super` is the name of a superclass and `Sub` is the name of a subclass. We can do:

```
Sub s = new Super();
```

Is this statement true or false? A. true B. false

13. [2 points] Assume `Super` is the name of a superclass and `Sub` is the name of a subclass. If we have:

```
Super s = new Sub();
```

we can only invoke the methods defined in `Super` class, and we cannot access the methods added by the `Sub` class. Is this statement true or false?

A. true B. false

14. [2 points] Assume `Super` is the name of a superclass and `Sub` is the name of a subclass, and `Sub` overrides the method of `demo`() defined by `Super`. If we have

```
Super s = new Sub();
s.demo();
```

Which `Demo`() method will be executed?

A. The one in the `Super` class

B. The one in the `Sub` class

15. [2 points] An interface is some special abstract class, in which all methods are abstract. Is this statement true or false?

A. true B. false

16. [2 points] In a subclass constructor, a call to the superclass constructor must:

A. appear as the first statement C. appear between the constructor's header and opening brace

B. appear as the last statement D. not appear

17. [2 points] All classes directly or indirectly inherit from this class:
   A. Root B. Object C. Super D. Java E. none of the above

18. [4 points] Find and fix the error(s).

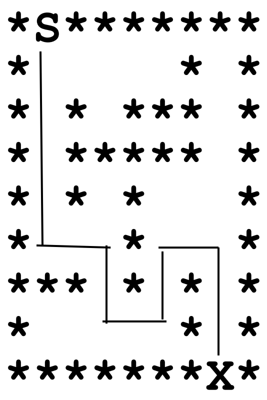
```
public class Vehicle {  //super class
      public abstract double getMilesPerGallon();
      //Other methods…
}

public class Car extends Vehicle {  //sub class
      private int mpg;
      public int getMilesPerGallon();
      {
            return mpg;
      }
      //Other methods…
}
```

Programming Problem: Robot Navigation Simulation

You are implementing a simulation of robot navigation. This project is borrowed from one similar in *Horstman, Big Java 3rd edition,* as well as from one similar used at U. Wisconsin- Parkside.

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
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 (File 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 **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 (For example, South=0, West=1, North = 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
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.

The RightHandRobot Class
Note that all robots have a current location. A **RightHandRobot** has both a location and a direction which it is facing. When initially constructed, we will assume the robot is facing South (because we will specify that all Start locations are in row 0 of the maze). After that, its direction is determined by its previous move. A **RightHandRobot**'s goal is to search for the exit while always trying to move right, if possible. If it cannot move right, it will try to move straight ahead. If it cannot move straight, it will try to move left. As a last resort, it will back up.

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'

For example, a **RightHandRobot** that is facing south, will first try to move west. If it cannot move west, it will then try to move south (straight). If it cannot move straight, it will try to move left (east). If it cannot move east, it will then go back north. Once the robot moves a certain direction, it then is facing in that direction. So – this kind of robot always needs to know the direction it is currently facing before it can decide which move to try first.

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

Bonus 10 points: FacingRobot Class and LeftHandRobot Class
The **FacingRobot** inherits from the **Robot**. The **FacingRobot** keeps track of the direction it is currently facing. Both the **LeftHandRobot** and the **RightHandRobot** inherit from the **FacingRobot**. The **LeftHandRobot** is just like the **RightHandRobot**, except it tries to move left first. If it cannot move left, it tries to move straight. If it cannot move straight, it will try to move right. As a last resort, it backs up.

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

Grading Policies
19. [50 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 listed below:
    - Use indentation to show the logical structure of your code.
    - Use blank lines to separate code blocks and give each code block a brief comment.
    - Give JavaDoc comments to each class and each method.
    - Include your name and algorithm in your program's heading documentation.
    - Output your name as part of the output.