CSE 11 **START EARLY!**
Winter 2015
Programming/Homework Assignment #7

Due: 13 March 2015, 11:59pm (electronic turnin) 200 points

This is a programming assignment and will be turned in electronically. You may develop your code on your own machines, however, the code *must* run on lab machines and *must* be turned in from a lab machine.

The Program – The Snake Game

This is a double programming assignment. It is intended to NOT be doable in a single night. It's long with a non-optimized solution taking about 700-800 lines of code and white space (not including comments). It's much more complex than any of your other programs but at the time of the assignment we have already covered the critical material. This assignment synthesizes nearly everything covered so far in CSE11.

The goal is to create a "snake" program that you play on your computer. To achieve this goal you will be using Arrays (or ArrayLists), rectangular arrays, Threads, graphical user interfaces with java Swing objects, for loops, while loops, boolean expressions, and good program design.

This is a much more substantial program than your previous assignments and you should expect it take you more time. Begin early. Ask questions. Be Patient and develop in stages. You have two weeks. Use them!

What the Final Game looks like when initialized and after being played for awhile with default width, height, segmentsize of snake

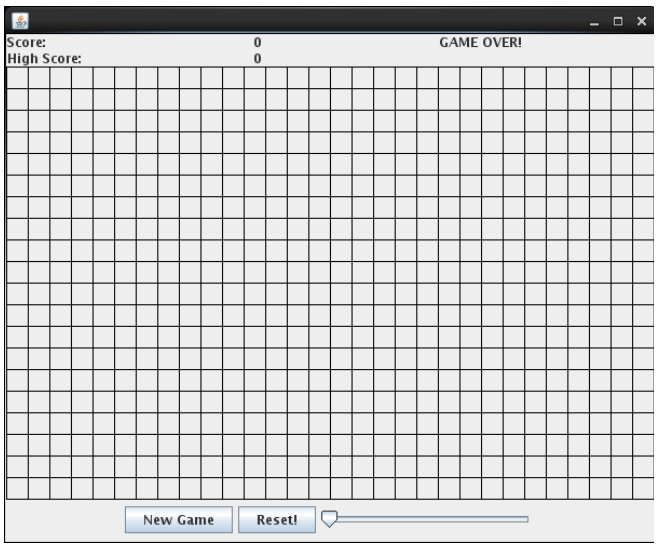

Illustration 1: When Game First Begins

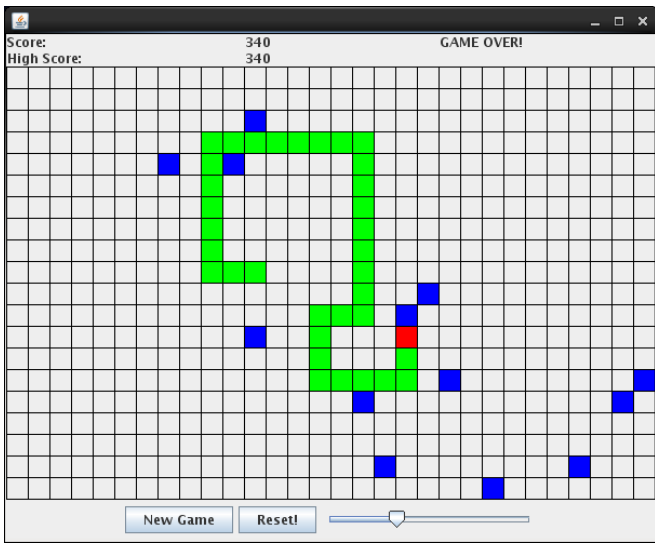

Illustration 2: After playing for awhile

Basic rules/operation of the game

When the game begins, a snake that is one segment in length (the head) starts in the top center and moves downwards automatically through the field. A user has two controls: turn right, turn left. The snake moves forward under program control. The logic of the Snake moving is programmed in a thread.

Every time the snake is turned (by the player), 1 segment is added to the snake. Every so often, a new obstacle is placed randomly in the field. Every so often, the snake speeds up. The goal is to get as many points as possible before running into an obstacle, the edge of the field, or the snake itself.

A slider controls the speed of the snake, game score and high score are kept track. A new game (without resetting the high score) can be asked for at anytime.

The Major Classes

When creating a larger program, one has to figure out how to break up the problem into smaller components. Since, this is likely the first big program you've ever created, this assignment guides you through some of this process. Let's begin by looking at the most likely classes, and then expanding

- the Snake itself
- the grid (or field) over which the snake is slithering and obstacles are placed
- The graphical interface

There are two less clear classes at this stage.

- a mover, which causes the snake to move under program control
- a coordinate class (basically row,column coordinates)

The Snake Class (`Snake.java`)

There can be several approaches to designing a Snake. The following functions are clearly essential

- Create the Snake at a particular location
- Move the snake in a particular coordinate direction (e.g. north-south, east-west)
- Turn the snake left or right
- Grow the snake (add a segment to it).
- Determine if the Snake has run into itself.

What about displaying the Snake on the canvas? That's a good design question. This assignment recommends that you do NOT make the Snake class a graphics object, but instead represent the snake as a growing Array of coordinates (See the Coord class below). Does the snake need to know the boundaries of the field? Perhaps, but is a simpler design if the Snake does not understand either boundaries or obstacles. Easier in the sense that a Snake could be used under a variety of game rules

The Snake as a list of (row,column) coordinates.

For simplicity, this assignment assumes that any time a snake is *n* segments long, each segment has the same (unit) length. One way to think of this is that each segment of the snake takes up a single 1x1 block.

A good approach to the snake is to ignore graphics and think of the snake as occupying a list of squares each at logical coordinate point (row, column). It’s easiest to think of the coordinate system as integers only. When a snake moves, it moves 1 unit only. It is probably best think of the snake as being on a coordinate plane that is infinite in all directions.

The Snake Head and Valid Movement
Suppose the Snake head is located at coordinate location (`n,m`) (`n,m` can be any integers). When the snake head moves it cannot move along a diagonal. What does that mean? The next coordinate the snake head can move to is limited to four coordinate locations:

- `(n+1,m)`
- `(n-1,m)`
- `(n,m+1), or`
- `(n,m-1).`

The is an additional caveat to this: The snake head cannot move in the direction of the body segment right behind it.

The Snake Body
The body of the snake is similarly a set of coordinates. Like movement of the Snake head, each subsequent segment can only be in one of four coordinate locations relative to the segment in front (or behind it).

Before going on, let's talk about the Coord class (You could inherit from the Point Class of Java AWT)

The Coord Class (`Coord.java`)
It is highly recommended that you create a Coord class where an instance of the class is an integer pair `(l,m)`. You probably want to write an equals() method and a toString() method, but this is not essential. You can then represent the location of the Snake head and body segments is a list (array) of Coords.

You probably also find it easier if you define two constructors for Coord, namely

```
Coord (int r, int c)
Coord (Coord initial)
```

The first constructs a coordinate pair from `r,c,` the second one constructs a copy from an existing Coordinate instance.

Back to the Snake.

Now that you have a Coord class, the snake itself is just a list of occupied coordinate locations.

Moving the Snake
Think about what it means to move the Snake. This is probably the first "hard" thing to get right and warrants significant testing to make sure you have your logic correct. To get this right, think about how the coordinate of each subsequent segment relates to the one before it. e.g. If you have snake of length two and you move the head, what coordinate does the tail (next segment) become? If you do this correctly, you can check that an asked-for move is valid and move the Snake (logically) in less than 15 lines of code.

Growing the Snake
Growing the Snake is just like moving the snake, except that the tail "stays put". A hint to make this efficient is 1) move the snake, then add a tail segment if the snake hasn't reached its maximum length. The professor's solution for growing the snake is less than 10 lines of code.

Other methods?
You may find some private methods useful. That's a design decision up to you. I would highly recommend that your "move" and "grow" methods return booleans to indicate that what was asked for was a valid move (or grow) request.

The GameGrid Class (`GameGrid.java`)

The GameGrid is the bounded X,Y coordinate plane on which the Snake and obstacles are placed. Think about it as "slots" that can be (a) empty, (b) have snake segments (or snake head), (c) have an obstacle. The body and head of the snake occupy coordinate points on the Game grid. The GameGrid is logical.

Clearly the Snake moves along the GameGrid, However the snake doesn't know about the GameGrid (or game rules, it only knows how to grow and move on an infinite grid). For example, if you invoke a move method on a Snake instance, the result might be that head would be outside the grid, or it would have run into an obstacle, or the Snake ran into itself. Other than running into itself, the Snake as class shouldn't need to know the specific rules of the game (or that there is grid boundary or obstacles)

Since the GameGrid class should enforce the rules of the game, it is logical that anything in the game that wants to move the snake or grow the Snake would ask the GameGrid class to move the Snake (and therefore also test all of the game rules in the process).

1. Copy the the current set of segments of the snake (record its current position)
2. move (or grow the Snake), then check if the move or grow violates any of the game rules
3. If move (or grow) doesn't violate the any of the game rules, erase the snake and re-draw it in its new position. If does violate the game rules, then don't move

The GraphicsGrid (`GraphicsGrid.java`)

The GameGrid must ask "something" to draw the snake and obstacles on the screen. You should be able to reuse a good fraction of the code you developed for PR6. You will need to modify to support different-colored squares.

The GameGrid will need to eventually tell the GUI when a move has occurred or when a rule of the game has been violated. Hint: you want to invoke methods (sometimes termed callbacks) on main GUI so that it can update game scores, game over and high score.

The SnakeGame (`SnakeGame.java`)
This is the GUI interface for the game. It needs to display scores (high score) present buttons and a speed slider, and indicate when a game is over.

Hint: Each time a New Game is requested, create a new GameGrid (which should in turn create a new Snake instance and new GraphicsGrid)

The SnakeMover (`SnakeMover.java`)

The SnakeMover must move the Snake forward on a timed interval. It must also respond to "left" and

"right" keystroke commands from the user. In other words, it needs to respond to keyboard events and run on a clock (think Thread). Detailed requirements of the game are given below. The SnakeMover will make requests of the GameGrid to move or grow the Snake

Summary of Class Interactions

1. A Snake is list of logical coordinates. It knows nothing about graphics. It can be told to move, grow, and report information about itself.
2. The GameGrid is the bounded X-Y playing field. It is the only object that directly moves/grows the Snake. This is so it can enforce the rules of the game. It will also convert the logical X-Y coordinates of the snake (and obstacles) to a graphics representation.
3. The SnakeMover is the automated mechanism of forcing the snake to move forward. The SnakeMover will request the GameGrid to move the snake, or grow the snake. If the GameGrid reports a successful move or grow, the SnakeMover reports this to the SnakeGame GUI
4. The SnakeGame is the Graphical interface that sets up the GameGrid, the mover, and responds to user input.
5. Coord is a "helper" class.

Rules/Requirements of the Game

Command-line invocation

```
$ java SnakeGame [width height segmentsize]
      width – Integer width of the playing grid in pixels
      height – Integer height of the playing grid in pixels
      segmentsize – Integer size of each snake segement in pixels

      defaults:  width = 400, height = 400, segmentsize = 10
```

This is a long section, because there a many details to get right.

1. Game Setup
    1. The Grid is initialized to Width x Height. Each cell in the grid is K x K pixels (K == segmentsize in the commandline). The command-line parameters. Width, Height, K are all integers.
    2. The Grid itself is N rows X M columns. N = height/K, M = width/K. Although you can specify width and height when you first run the program, the width and height when New Game is pressed, should be the actual width and height of the GraphicsGrid
    3. Print out a usage statement if the parameters are non-sensible (e.g. negative width/height, K that is larger than width or height, non-integer). (Reuse your code from PR6!) It is NOT an error for K to unevenly divide width or height. Fit as many KxK blocks in the width x height game grid as possible without any partial sized blocks: 200 300 23 ➔ 8 blocks in the width direction, 13 blocks in the height direction.
    4. Empty parts of the field are colored White/background color of your choice
    5. The playing field has each grid square outlined by a black line (See Pictures above). If you choose your own background color, the lines should be apparent.
    6. The head is colored differently from the snake body (Red for the Head, Green for the body in the above, you may choose your own colors)
    7. Obstacles are colored differently from the head,body, and empty parts of the field. Obstacles are colored Blue in the above, you may choose your own color)
    8. The Grid should centered both vertically and horizontally.

9. You must provide a button for “New Game” and “Reset”
10. You must provide a speed slider to change the speed of the snake
11. Above the play field are labels for Score, High Score, and an indicator for when a game is over. See Layout for the pictures above, your game should have the same layout

2. Game Initialization
    1. A snake of length 1 (just a head) is created at the top of the field and in the middle.
    2. The initial direction of movement is downwards
    3. If the speed slider is more than the minimum, that should be the speed of snake.
    4. No obstacles should be in the field
    5. When the New Game button is pressed, you should get the current size of the GraphicsGrid and use that as the width and height. A new Grid that fits as many squares as possible should be created. See figures below.
    6. If the window is resized by the user while a game is active (snake is moving), the Grid should NOT be resized. Only resize the grid when New Game is pressed.

3. Snake Movement
    1. The “j” key should turn the snake left.
    2. The “l” (ell) should turn the snake right.
    3. Left and right are relative to the current forward direction of the Snake. For example if the snake is currently moving downwards then a right turn would make the snake move towards the left side of the screen. If the snake were moving upwards, a right turn would cause the snake to move towards the right side of the screen
    4. When the snake is turned by the user, it grows in the new direction. Only left or right turns can make the snake grow longer
    5. If the user does nothing, the snake moves forward on its own.
4. Scoring and obstacles
    1. Each time a snake grows a segment, 10 points should be added to the current score
    2. The high score should keep track of scores between new games, if the current score would be higher than the high score, the high score should update immediately.
    3. Every 10^{th} time the snake either moves or grows, a new obstacle should be randomly placed in the field
    4. The game is over if the snake hits the edge of the field, hits an obstacle or hits itself.
5. Timed movement
    1. The slow speed moves the snake ~once/second
    2. The fast speed moves the snake ~20 times per second
    3. The speed of the snake should increase every 100 points until the maximum is reached.
    4. The SnakeMover should not ever sleep for longer than 50ms. This is to keep the interactivity of the game. It’s easiest to think of moving once/second as counting down twenty, 50ms sleeps
    5. As soon as the user “turns” the snake, the automated timer used by the snake mover should reset.
6. Reset Button
    1. The speed should be reset to the slow speed
    2. The high-score should be reset
    3. The current game should stop immediately
7. Game Over
    1. When the snake violates a rule of the game, all movement should stop and the GAME OVER! Indicator should be shown.
8. These requirements may be amended/clarified during the assignment.

Example of initial screen and then after New Game has been pressed for

```
$ java SnakeGame 200 300 23
```

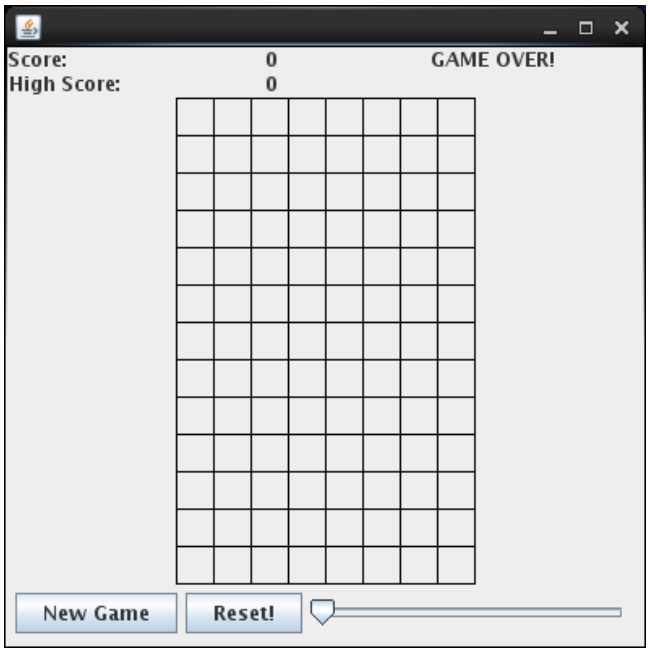

Illustration 3: When Game First Begins. Notice that the Grid is centered and uses creates 13 row x 8 column Grid.

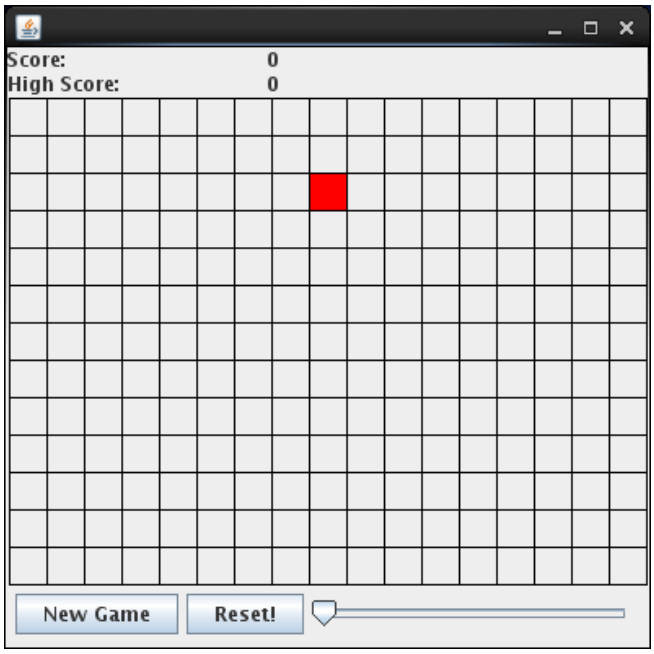

Illustration 4: a few seconds after New Game Button was pressed. Note how the Grid fills out as many squares as possible

Development Strategy

This is a big program and you must develop in stages, the final program is roughly 700 -800 lines of code (some of the code you have already written, so it's less). Take a deep breath. And plan to work on this assignment in small chunks. Celebrate the progress as you make it. What follows is a suggested plan of attack You should be looking to do 2-4 steps at a time. Estimate of the whole project is somewhere between 10 and 20 hours. (You have TWO weeks and potentially a partner). If you find yourself getting stuck, ask for help. **We really want you to succeed at this program**. The strategy is pretty detailed so that you can make good, measured progress. This is a very challenging assignment.

<u>When you get things working at various steps, make copies of your working code and store in a safe place.</u>

1. Forget about graphics as you start out. You need to get the Snake and Coord Classes coded correctly. Together they total 125 – 150 lines of code.
2. Write the Coord class and test it. This shouldn't take too long, but make sure you can create arrays of Random coordinates, print them out, compare them, and get the individual elements of a Coordinate reliably. Feel free to inherit from the Point class. It's handy for your Coord class to think about (rows,columns) instead of (x,y). We specify graphics width x height, but arrays are indexed (row, column) (The transpose of graphics).
3. Write the first edition of the Snake class that creates a snake, can retrieve the array of

coordinates that make up the snake, and print the coordinates.

4. Write a small testing program that creates a Snake and performs a sequence of 20 – 50 random moves. Print out the coordinates after each move/grow. Verify that the coordinates are being updated the way you think they should. Try giving your Snake invalid moves/grows and make sure that it behaves properly.
5. You may want to implement a method that determines if the Coord intersects any part of the Snake. That's just a friendly hint.
6. If you have the above working well, you ready to start on the game grid.
7. Create a GameGrid that is NxM where each element of the grid is a character. Use a '.' for an empty space, a "#" for a snake body part, an 'H' for the snake head. You don't have to implement all aspects of the GameGrid at once
    1. You need to initialize an empty grid, create a new Snake and place the snake at the top center of the grid.
    2. you need to implement move and grow methods in GameGrid that will move/grow a snake AND check that the move/grow was valid (Stayed within the GameGrid)
8. Test this first edition of the GameGrid by reading input from the user. If the user hits "k" the snake should move forward, hits "j" it should turn (and grow) left, hit "l" it should turn and grow right. Use print statements liberally to see how things are going, but after each keystroke draw out the game grid. This is primitive ascii graphics, but you will know right away if things are going the way you expect them to go. Please note, there is no easy way in a java console program to react to just a keypress, you have to hit the key and then return.
9. At this stage you should be able to move your snake under keyboard control and have it turn/grow. The "graphics" should all be ascii (plain text).
10. Now it's probably time to make the first graphical version. Copy your current GameGrid into a safe place. Now instead of ascii graphics have your GameGrid support graphics and work to get the graphics right This where GraphicsGrid comes in. I would create test driver (e.g. SnakeTest) that is a JFrame wth a GraphicsGrid in Panel. It should create a GraphicsGrid in the Panel. You can (and probably should) reuse much of PR6. . Still use your keyboard (+ return) to move the snake. You are simply moving from ascii graphics to Swing graphics. Getting the graphics right means the each logical square on the GameGrid is KxK pixels.
11. When you have that working, It's time to build a SnakeMover class. Do this in two steps. First, have the SnakeMover implement the KeyListener interface. If you have things working correctly, j will turn the snake left, l will turn it right and k will move it forward. At this point you have basic movement in a graphics environment working (We haven't even started on the gui, yet). Note, you have to add the KeyListener to the JPanel (GraphicsGrid) and you have to click on the canvas to make the listener active.
12. The next step for SnakeMover is take away the manual "k" movement and make SnakeMover a Thread. Fix the timestep so that it automatically moves the snake forward every second. Have the run loop pause for 50ms, if the "j" or "l" (ell) keys are not hit for twenty consecutive pauses, then move the snake forward one. Reset that counter each time j or l is hit.
13. At this point, you have basic, automated snake movement with keyboard input working. Play with your program for while. Check that updates are correct, drawing is correct and no exceptions occur during testing.
14. Time to start on the GUI. If you have JFrame-based SnakeTest.java, copy it to SnakeGame.java and edit appropriately so that it still works. Then you can lay out the components so that the scores, buttons, and sliders show up where you want them. At this stage I wouldn't add any actionListeners. Just get the layout to look right.
15. Next, make sure that when you hit the "New Game" button, things work as you expect (GAME OVER! Goes away, score is zero, GameGrid is created.) You now need SnakeGame to be an

ActionListener.

16. If you've gotten to 15, you've have made excellent progress. The rest of the assignment makes the game more fun. Honestly, you are on the downhill slope now. If you only get this far and have no errors, your score for functionality is already in the 75 percent range.
17. Work on getting communication working between the SnakeMover and the GUI. The *mover needs to report to the GUI* every time a valid move/grow is made. And the SnakeMover needs to tell the GUI when to put the GAME OVER! On the display.
18. Next add obstacles. Every 10th move/grow a new obstacle needs to be randomly placed on the field. Create a method called addObstacle() in the GameGrid. *Invoke the addObstacle() method from GUI.* It is the GUI that deciding after how many moves are made, should an obstacle be added. You will have to also modify GameGrid to detect if the Snake moves into an obstacle.
19. Add speed adjustment. Every 100 points, the speed should increase. One way to think about this is there are 20 speed steps. Increase the speed by one speed step every 100 points. If you want, if the game starts at speed level 10, it doesn't need to increase the speed to 11 until the score is 110.
20. Play your game. Enjoy the fruits of a long programming assignment.

Grading (total of 200 points)

160 points – Does your program compile and run properly. If all aspects are met, then you will receive full credit. Various misfeatures or bugs will lose appropriate points. We will run your program. Since the assignment does not specify particular signatures of methods, we can only test your final product.

40 points – Commenting/indentation/style. Please see the Style guide. You Must put your NAME:, LOGIN:, ID: in every file you turn in. Please use ALL CAPS for the NAME, LOGIN, ID labels. Use Javadoc-style comments on all public methods that you define.

Are you using private methods and variables?
Do you use accessor methods?
Are your methods overly long?
Are you repetitive in code?
Are the comments informative?
Are Variable names appropriate?
Are the classes defined used in a reasonable way?

Turning in your Program

YOU MUST BE ON THE LAB MACHINES FOR THIS TO WORK. PLEASE VERIFY WELL BEFORE THE DEADLINE THAT YOU CAN TURNIN FILES

You will be using the "bundlePR7" program that will turn in the following files

```
Coord.java
GameGrid.java
GraphicsGrid.java
Snake.java
SnakeGame.java
SnakeMover.java
```

No other files will be turned in and and they **must be named exactly as above.** BundlePR7 uses the department's standard turnin program underneath. If you want to define other classes, they should be helper or inner classes defined in one or more the above files.

To turn-in your program, you must be in the directory that has your source code and then you execute the following

```
$ /home/linux/ieng6/cs11w/public/bin/bundlePR7
```

Some of you have had trouble turning in programs and sometimes turn in the wrong version. To address this, a A HIGHLY Recommended sequence of commands is the following: (You are not ready to turn in files if you get any errors)

```
$ rm -i *.class
$ javac Coord.java GameGrid.java Snake.java SnakeMover.java
SnakeGame.java
$ java SnakeGame
$ /home/linux/ieng6/cs11w/public/bin/bundlePR7
```

The above will remove your compiled java classes (asking you first if you want to remove them, say yes to *.class files don't make a mistake and delete .java files!!!), then will compile the files you are about to turn in.

You can turn in your program multiple times. The turnin program will ask you if you want to overwrite a previously-turned in project. **ONLY THE LAST TURNIN IS USED!**

Suggestion: if you have classes that compile and do some or most of what is specified, turn them in early. When you complete all the other aspects of the assignment, you can turn in newer (better) versions.

Don't forget to turn in your best version of the assignment.

START EARLY! ASK QUESTIONS!
HAVE FUN PLAYING YOUR SNAKE GAME!

Some Frequently Asked Questions

Can I change the colors of the grid,head,snake,obstacle? Yes. Feel free to.

Can I add other elements to the game interface? Yes. As long as the required functionality is achieved, you may add extra information.

I see the word "callback" in the instructions, what is that? It's a well-defined method on an object. Suppose Object A has callback named callme(), If Object B has a reference to A (called a), then when something occurs that should be of interest to an A, B invokes a.callme(). In this code, the GameGrid is calling back to the GUI to indicate that the snake has moved. In turn, the GUI calls back to the GameGrid when a new obstacle should be placed on the Grid.

This program is enormous? Does anybody finish it? It is big. You can have a partner. Many people finish it completely. If you get through this program, you've learned quite a bit. You must develop in stages or you will not realistically complete the assignment.

Arrays or ArrayList, which should I use? I suggest both. The GameGrid itself is most easily implemented as two-dimensional array. Each cell in the GameGrid is one of four things: EMPTY, SNAKE HEAD, SNAKE BODY, or OBSTACLE. You could use anything to represent that state of a grid cell (integer constants, Colors, your own Object type). The coordinates of the Snake itself could be stored in an Array or an ArrayList.

I have KeyListener in my GraphicsGrid (JPanel), I don't see the key presses. **Why?** This may be a focus issue. Clicking on the grid, should set the keyboard focus correctly.

How come your rules of snake are different than my phone's? That's just the way it is.

Can I add extra classes and/or not use as many classes as specified? You may do both, but you cannot add/takeaway any of the defined files. If you want an "Empty class", create the file, and put in comments as to why you aren't using that class. If you want to add a class, make it a helper class or inner class defined in one of the files above.