Programming Assignment 8 (100 Points)

Due: 11:59pm Thursday, November 19th
START EARLY!

Overview:

This programming assignment is a simplified version of the classic game Snake. For those that are not familiar with the game Snake, the goal is to make the snake cover the entire grid. When you start the game, the snake will be continuously moving in the direction you tell it to. The snake will grow if it eats an apple, which will be randomly placed on the canvas. If the head of the snake crashes into a part of its body (it tries to eat itself), then the game ends. In this specific version of the game, let's say that if the snake hits a border, it also crashes and the game should end.

For a more general idea of how Snake works, follow this Wikipedia Link. Remember, the Wikipedia link **is not** a substitute for the write-up; it should only be used to understand how the game in general should work. Implementation of this game must follow the exact specifications found in this write up.

README (10 points)

You are required to provide a text file named **README,** NOT Readme.txt, README.pdf, or README.docx, etc. with your assignment in your pa8 directory. There should be no file extension after the file name "**README**". Your README should include the following sections:

Program Description (3 points) :
Explain how the user can run and interact with each program. What sort of inputs or events does it take and what are the expected outputs or results? How did you test your program? How well do you think your program was tested?
Write your README as if it was intended for a 5 year old or your grandmother. Do not assume your reader is a computer science major. **The more detailed the explanation, the more points you will receive.**

Short Response (7 points) : Answer the following questions:

Academic Integrity Related Questions:

1. How do you maintain your integrity even when you are stressed, pressured, or tired?

Java Questions:

2. A high school student is trying to write a Java program that will draw different shapes and in different possible colors. To do this, she has written just one Java class called ShapeDrawer, which contains all the necessary methods like drawRedCircle(), drawBlueCircle(), drawYellowSquare(), drawGreenSquare(), and so on. Using object-oriented terminology, describe how you can help the student improve her design.
3. List at least two ways in which Java Interfaces and Java abstract classes are **different**.
4. List at least two ways in which Java Interfaces and Java abstract classes are **alike**.

Vim Questions:

5. How can you run gvim through the command line to open all Java source code files in the current directory, each file in its own tab?

Unix Questions:

6. Suppose you are currently inside a directory and in there you want to make a new directory called fooDir. And inside fooDir, you want another directory called barDir. Using only a **single** mkdir command, how can you create a directory called fooDir with a directory called barDir inside it?

STYLE (20 points)

Please see previous programming assignments for details on Coding Style.

CORRECTNESS (70 points)

All of your code/files for this assignment need to be in a directory named **pa8** in your cs11f home directory. Please see previous programming assignments to setup your **pa8** directory, compile, and run your programs.

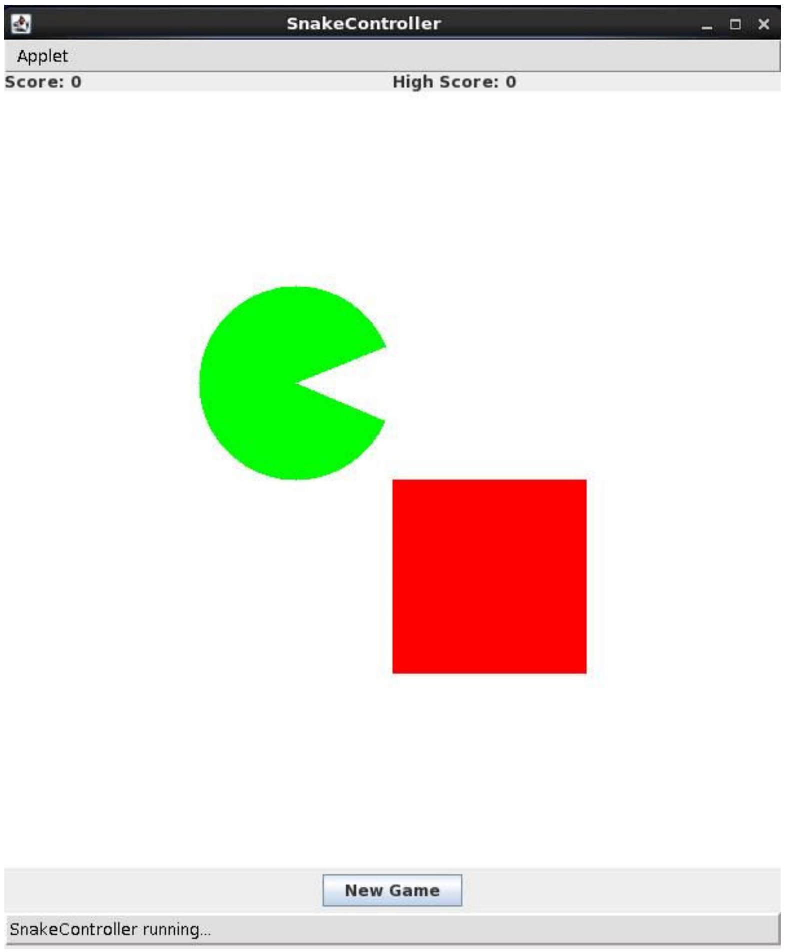
Part 1 – Prepare

Copy the files from ~/../public/PA8/ into your ~/pa8/ directory.

START EARLY!

This might be a long assignment. This is the first time we are doing this programming assignment, so we have no idea how easy or difficult this assignment might be. So, just in case, start super early!

Part 2 – GUI:



How the application runs with command line args 600, 150, 200.

SnakeController.java – This class controls all the UI of the game. It should have a score and a high score on the upper panel, and a new game button on the lower panel. The score should increase by 100 points every time the snake eats an apple. The high score will only update when you press new game after the game ends (whether it be on game over or on a win). If the current score is greater than the high score at the end of the game, it should be updated. The rest of the main frame will be the space where the snake is moving around, and where apples can be placed.

The controller should also have Text objects that will be displayed once certain events occur. When the snake crashes, the gameOverText should be displayed to the screen (same with winText when a win occurs and pauseText for a pause). The font size of these text objects should be 40. These text objects should be centered vertically and horizontally. There is an easier way of doing this as opposed to calculating the starting location manually. HINT: check out Text's getWidth(), getHeight(), and moveTo() methods.

This application does not support canvas resizing. The canvas must be initialized as a valid size (for more info, see the command line arguments section in Part 3 Implementation). Canvas resizing will not be tested (WNBT).

After initializing all the labels and buttons, call validate to reset the canvas size. This canvas size should be equal to what was passed in as a command line argument (more details under implementation).

Finally, you will need to create a new snake and apple. The snake should be placed one square down and one square right from the upper-left square location (picture it as a 2D grid, and you are accessing element grid[1][1] and placing the head there) at the start of each game. The snake will be green (Color.GREEN). The apple should be placed randomly on the board. The apple will be a red FilledRect (Color.RED). This will be covered in detail in the Implementation part of the write-up.

The snake will move in response to the four arrow keys. Snake and SnakeController need to implement KeyListener, and the keyPressed method (keyReleased and keyTyped methods should have an empty implementation). Depending on what arrow key is pressed, the associated direction will be set as the Snake's current direction, and if the spacebar is pressed, the pause text is shown/hidden in the controller, and the snake should stop running. This will be covered in detail in the Implementation part of the write-up.

At the end of begin and actionPerformed, add this line of code:

```
canvas.requestFocusInWindow();
```

Without this line, you will need to click the canvas every time you want the KeyListener to work. This line of code makes sure that the focus is always set to the canvas. Normally If you were to click the New Game button, the focus would shift to the button (that's why when a button is normally selected / highlighted, you can press space to click the button). However, in this assignment, we <u>don't</u> want this to happen, so adding this line of code avoids that issue.

<u>Part 3 – Implementation:</u>

Command Line Arguments:

For this assignment, instead of hard-coding the dimensions of the canvas and such into the Controller class, let's pass them in as command line arguments. There shall be 3 arguments: the dimensions of the canvas, the diameter of each SnakeSegment / length of each apple, and the delay.

Usage:

```
java -cp ... SnakeController [dimensions] [segment_size] [delay]
```

Example using valid arguments:

```
java -cp ./Acme.jar:./objectdraw.jar:. SnakeController 600 200 300
```

You will be parsing the command line arguments in main() in SnakeController.java. The arguments will be stored in the String [ ] args array passed in as a parameter to main() in the order that they are given to the program. This means the first argument is in args[0], the second is in args[1], and so on.

Before we can start up the program, we need to check for errors in the command line arguments. If any of the arguments fail a requirement, print the appropriate error message (if there is one), print the usage, then exit the program using:

```
System.exit(1);
```

All the error messages are given to you in PA8Strings.java. You should, however, always print the usage string when there is an error.

Error Checking:

Number of arguments: Before we can check the command line arguments, we need to determine how many arguments were passed in. If we don't have the correct number of arguments, print the corresponding error message, usage string, and exit with system status 1 (as shown above).

Dimensions: The canvas (the area that the snake can traverse) will be a square, so we will only need one value to represent the dimensions of the canvas. Here's how you get the first argument as an int:

```
dimensions = Integer.parseInt(args[0]);
```

Validate that the `dimensions` variable is between 500 and 800 inclusive (think checkRange from pa3). If the variable is not in that range, print the OUT_OF_RANGE String from the PA8Strings.java file provided. In order to format this string, you can use the following code:

```
System.out.printf(PA8Strings.OUT_OF_RANGE, value, low, high);
```

The out of range String has three placeholders in the string (represented as %d for ints). When you pass it additional arguments in the printf call, it replaces these placeholders with the value of the passed in variable. Then exit the program using system status 1.

Segment Size: The second argument represents the diameter of the snake segments and the length of each apple. We need to convert this argument to an int like we did for dimensions.

Validate that `segmentSize` is between 20 and 400 inclusive. We also need to make sure that the dimensions are evenly divisible by the size. In other words, when we place the segment, each segment should fit perfectly on the canvas, no matter where it is (it should not hang off past the edges of the canvas). If dimensions is not evenly divisible by segment size, print the appropriate error message. Lastly, we need to make sure that the segment size is less than or equal to half the size of the canvas. If the size is greater than half the size of the canvas, we should print the appropriate error message. If there were any errors with segment size, we also need to print the usage, and exit the program.

Delay: Finally, the third argument, delay, represents the amount of time to pause in each iteration of the while loop in the Snake's run() method. The delay must be between 100 and 1000 inclusive. Remember that 100 will correspond to a faster snake speed (because the time delay is shorter) and

1000 will correspond to a slower snake speed (because the time delay is longer). If the delay is not within 100-1000 inclusive, print the usage, and exit the program.

Note for Testing:
To simplify things, we **WILL NOT** be testing your command line argument processing with anything besides integers. You can assume that all input given will be an integer. However, you will also need to check to make sure that the parameter list is the correct length.

Since we will be providing the strings for you, the print statements must look EXACTLY as those in the examples. No leniency will be given to those that do not match the examples, and you will lose full points per test regarding input validation.

Starting up the game:
Once you have determined that the command line arguments are all valid, then you can create a new Acme.MainFrame. Because we need access to all the other command line arguments as non-static variables, we are going to change the SnakeController constructor to take in a String [] args, instead of just using the default constructor.

```
new Acme.MainFrame(new SnakeController(args), args,
                   dimensions, dimensions+Y_PADDING);
```

The dimensions value will not be the same number you pass into the Acme.MainFrame initialization call**.** The reason for this is because we added labels and buttons to our application. The parameter passed into the Acme.MainFrame call sets the size of the entire frame, and placing other components such as labels and buttons on this frame will reduce the size of the canvas. To avoid this, we'll pad the vertical dimension argument with a 50 pixel padding. There's been a weird error when I tested this on ieng6. Sometimes, the width of the canvas is 6 more pixels that what was specified in the argument list. Print the canvas width after your call to validate, and check to see it equals what was passed in. If this error is consistent, try and edit the dimensions argument so the size of the canvas is correct after validate.

EXAMPLE OUTPUT:

```
[cs11fxx@ieng6-201]:~:$ java -cp ./objectdraw.jar:./Acme.jar:. SnakeController
Usage: java -cp ./Acme.jar:./objectdraw.jar:. SnakeController DIMENSIONS SEGMENT_SIZE DELAY

[cs11fxx@ieng6-201]:~:$ java -cp ./objectdraw.jar:./Acme.jar:. SnakeController 600 1 1
Error: value 1 is out of range. It should be between 20 and 400
Usage: java -cp ./Acme.jar:./objectdraw.jar:. SnakeController DIMENSIONS SEGMENT_SIZE DELAY

[cs11fxx@ieng6-201]:~:$ java -cp ./objectdraw.jar:./Acme.jar:. SnakeController 600 400 1
Snake segment size of 400 is too large for the given dimension 600 x 600
Usage: java -cp ./Acme.jar:./objectdraw.jar:. SnakeController DIMENSIONS SEGMENT_SIZE DELAY

[cs11fxx@ieng6-201]:~:$ java -cp ./objectdraw.jar:./Acme.jar:. SnakeController 600 37 1
Snake segment size of 37 does not evenly fit inside the given dimension 600 x 600
Usage: java -cp ./Acme.jar:./objectdraw.jar:. SnakeController DIMENSIONS SEGMENT_SIZE DELAY

[cs11fxx@ieng6-201]:~:$ echo “Continue to test your code”
```

You need to include the newline between the usage statements and the prompt. If this is not matched EXACTLY, no partial credit will be given for these test cases.

SnakeController.java – Aside from initializing UI elements, the controller is also responsible for placing apples on the canvas, and updating the text objects that show the game's state (paused, game over, win). This first ask becomes complicated, because we must place an apple in an open space. In other words, we cannot place an apple on top of the snake.

public void keyPressed(KeyEvent e) – This method should only really be listening for the VK_SPACE key (which corresponds to the space key). If the space key is pressed, switch the state of the paused boolean variable, and either display or hide the pause text on the screen, depending on the value of the boolean.

public Coordinate placeApple(DrawingCanvas canvas) – An apple can only be placed on an open space. That means we should have one ArrayList to keep track of all possible open spaces (represented as Coordinates). These Coordinates will be the upper left corner of the segment / apple. If the Coordinate is 0 <= xValue < dimensions && 0 <= yValue < dimensions, then it is an open spot. We are going to be modifying this ArrayList, so we need to make sure that every time this method is called, this ArrayList contains all possible open spaces, and not some subset of them.

After generating all the possible open spaces, we need to disclude all the Coordinates that have a SnakeSegment on top of them. To do this, create a method in Snake that returns the Coordinate ArrayList of all its snake segments. Once you do that, iterate over the snake Coordinate ArrayList and for each SnakeSegment Coordinate, call remove on the open spaces ArrayList. You can call the remove method as follows:

```
openSpaces.remove(snakeSegmentCoordinate);
```

We can use this method because Coordinate has an equals method. So this remove will check to see if there are any Coordinates in our openSpaces ArrayList that are equal to our Coordinate, which is the exact behavior we are looking for.

Finally, we just need to randomly place the apple in one of the open locations left, after removing all the ones that overlapped SnakeSegments.

Snake.java - This class defines what a Snake is (an ArrayList of SnakeSegments) and what a Snake can do (it is an Active object that moves in response to the arrow keys). The snake's constructor must have this signature:

public Snake(Coordinate coord, int size, int delay, DrawingCanvas canvas, int dimensions, SnakeController controller) –

This time, we need to pass in the controller into the Snake constructor in order to inform the controller if the snake has crashed during its run. If it has, the controller should display the Game Over text and stop the game. Clicking the New Game button shall reset the canvas.

private boolean move() – Takes care of moving the snake. First we need to check to see what the snake's current direction is. Move the head one location forward in the direction that the snake is moving, and keep track of where the old location was. We're going to do this because we need to know where to place the next SnakeSegment in the ArrayList; it should be where the segment before it was before the move. After moving everything over, your Coordinate reference should be pointing to a spot that no longer has a SnakeSegment. If the snake has eaten an apple, place a new SnakeSegment there.

This method returns true if it was a valid move. It shall return false if the snake crashes (whether it be into itself, or the borders of the grid). The run method should check the return type of the move method, and call the appropriate SnakeController methods depending on the result.

If the snake is moving one direction, and the user pressed the key that represents the opposite direction, the snake would eat itself with our implementation. Given a snake of size two, however, it would not do so, because of how our move is implemented. This behavior will not be tested (WNBT).

public void keyPressed(KeyEvent e) – This method will be fired every time the user hits a key on the keyboard. The snake must respond to multiple buttons: if any of the four directional keys (up, down, left, and right) are pressed, the snake should change direction to face that new direction when it moves next. These buttons correspond to the constants VK_UP, VK_DOWN, VK_LEFT, and VK_RIGHT, respectively. These constants are associated with the KeyEvent class, and must be used in your code. You can compare the keyCode of the event by using this code:

```
int keyCode = e.getKeyCode()
if(keyCode == KeyEvent.VK_UP) {
      // Handle up direction.
} // Etc.
```

When the game opens, pressing any of the four movement keys should start the game, with the snake moving in that direction. When the game ends, whether it be from a win or a loss, the movement keys will not do anything. On a win or loss, the **only** way to reset the game is to press the New Game button on the UI. You should be able to click restart at **any point** of the game (including when the game is paused). Only when the game is properly reset (snake head is placed in the correct location and an apple is randomly placed) can the game start by pressing one of the directional keys. This will cause the snake to start moving in the direction of the arrow key that was pressed.

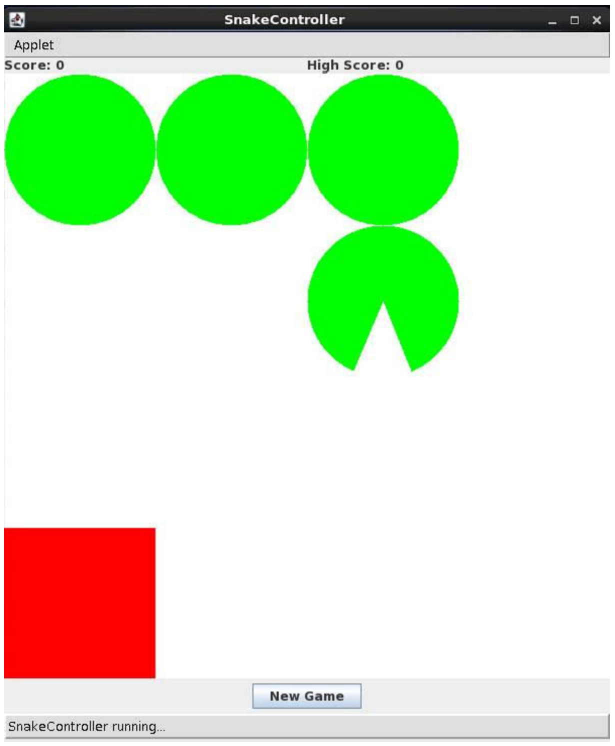
In addition to responding to the movement keys, the snake must also be able to pause its movement. The pause action will occur when the user presses the spacebar. If the user presses pause, the snake needs to ignore all incoming input from the keyboard, except for another spacebar press. For example, if the snake is moving down, and the user presses space, followed by up, and then space again, the snake should still be moving down. Pausing should also make the pauseText from the controller appear on the canvas.

SnakeSegment – This class is defined within Snake, and defines what a segment of a snake should be. A SnakeSegment is going to need a FilledArc to represent its visual appearance on the canvas. If a SnakeSegment is the head, then the arc angle of the FilledArc is HEAD_ARC_ANGLE (provided in the starter code below). If a SnakeSegment is part of the body, then the arc angle of the FilledArc is BODY_ARC_ANGLE (also provided in the starter code below). Using these arc angles, the head segment should look like a circular pizza with one slice missing, and the body segments should be complete circles. Each SnakeSegment should also have a Coordinate of where it is on the grid. This class should be able to return its coordinate through a getter, and should be able to move itself (hint, implement a moveTo method for SnakeSegment, as opposed to a move).

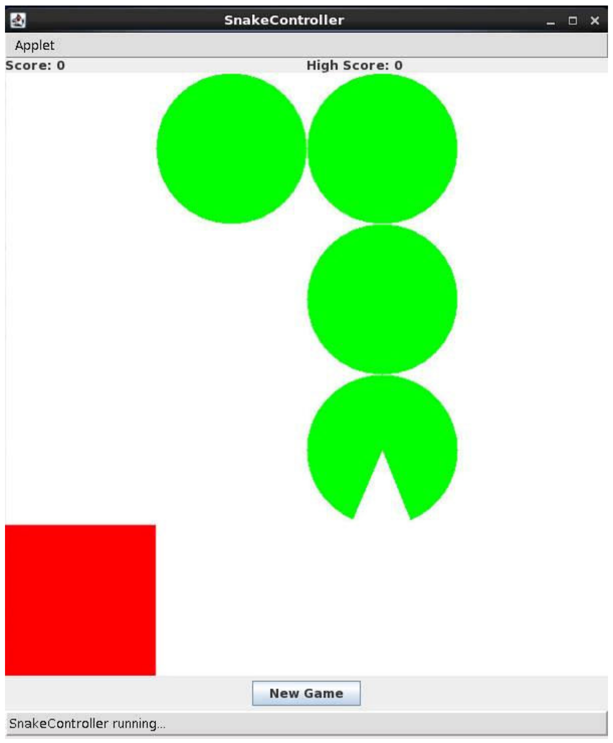
Coordinate.java - a utility class that encapsulates the location (an x and a y) of a SnakeSegment or apple, and the locations that SnakeSegments/apples can be. Whenever you are dealing with placing things on the canvas, you should be using this utility class. <u>You DO NOT have to comment this class, because you did not write it.</u>

Direction.java - another utility class that encapsulates the possible directions a snake could be traveling. For the purposes of this assignment, you will only need to use the enum to select one of the four directions, and use it to compare directions. More specifically, you can access these directions in your code like this: Direction.UP, Direction.DOWN, etc. If you wanted to compare these directions, use the == operator. You DO NOT have to comment this class, because you did not write it.

GENERAL MOVEMENT:

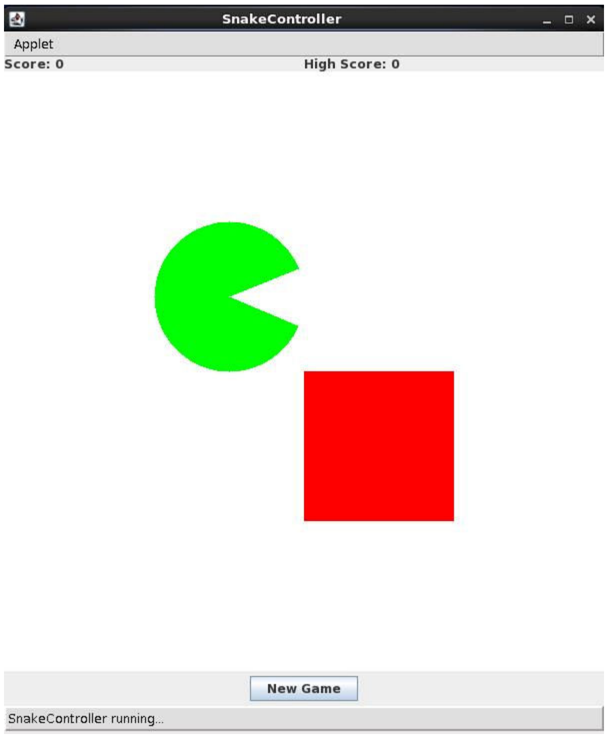


Before moving

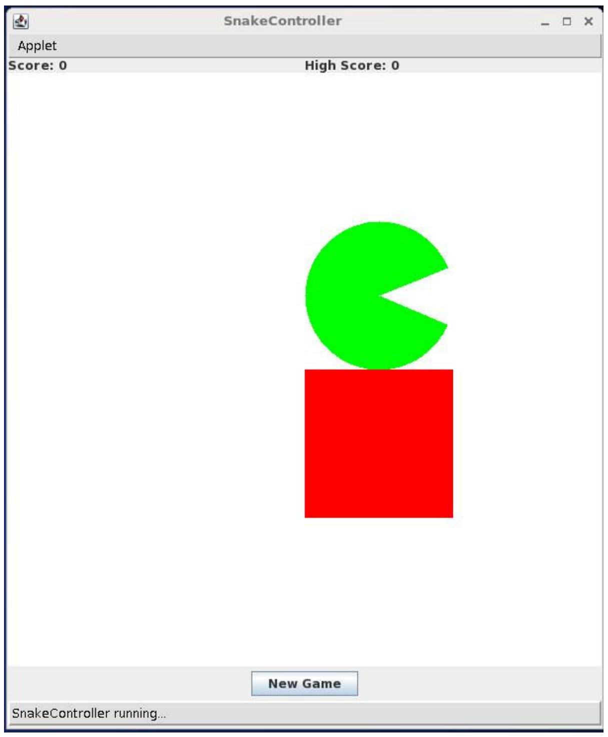


After moving

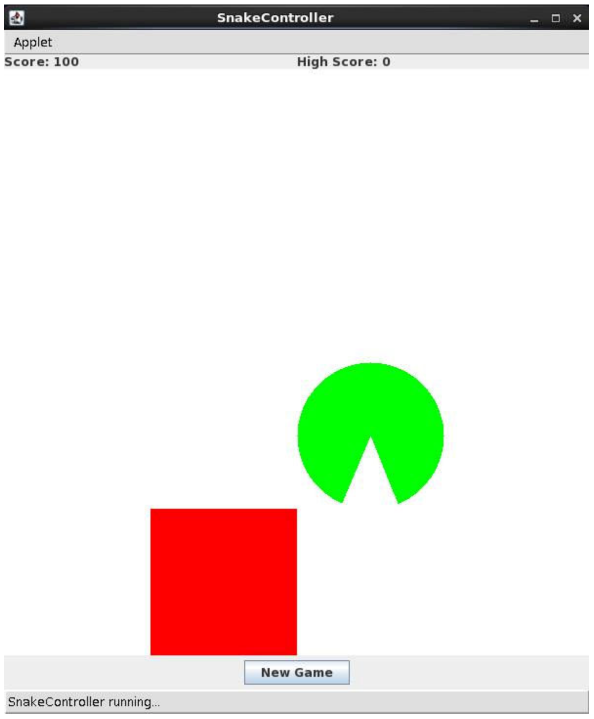
GROWING:



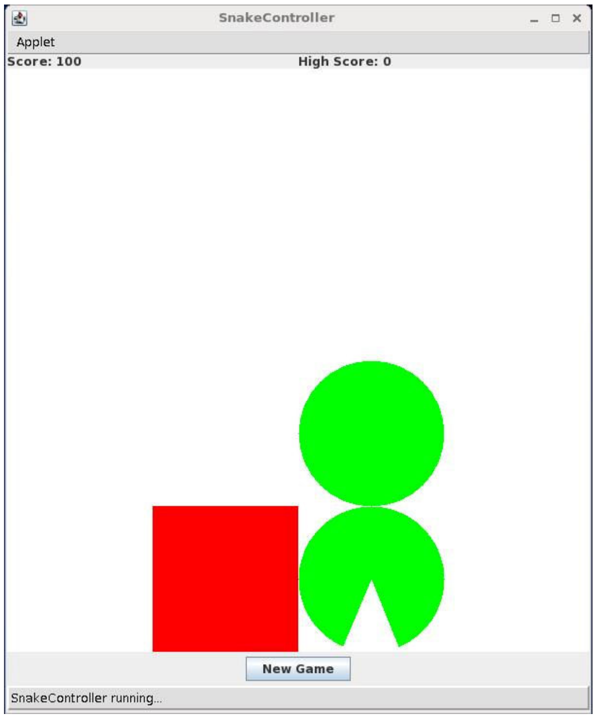
How the game looks on start up



After pressing the right arrow key.



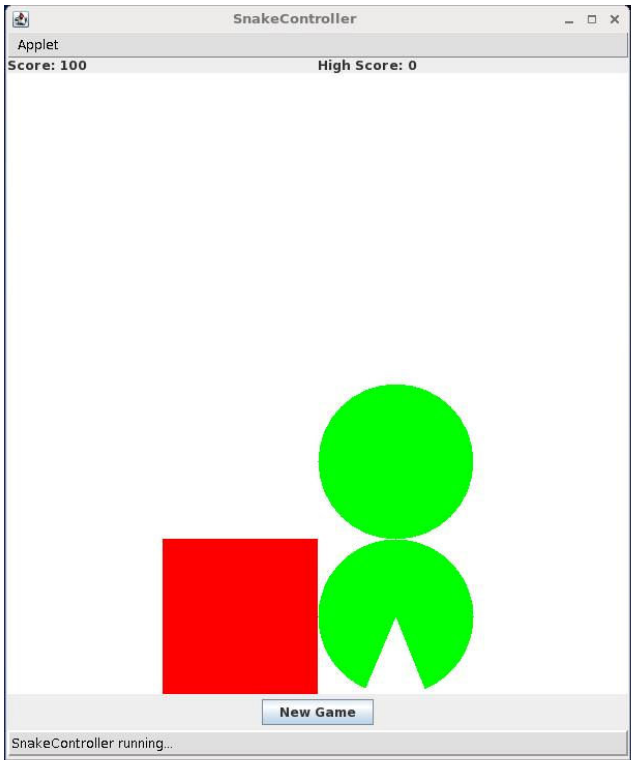
The snake ate the apple, so another apple needs to be placed in an open space.



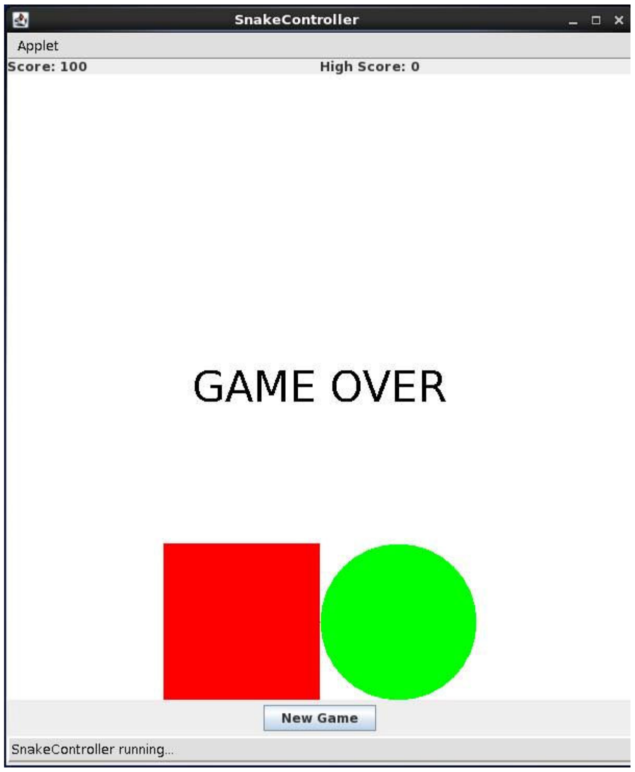
The snake needs to grow, because it just ate. It is done growing.

LOSING:

HITTING A WALL:

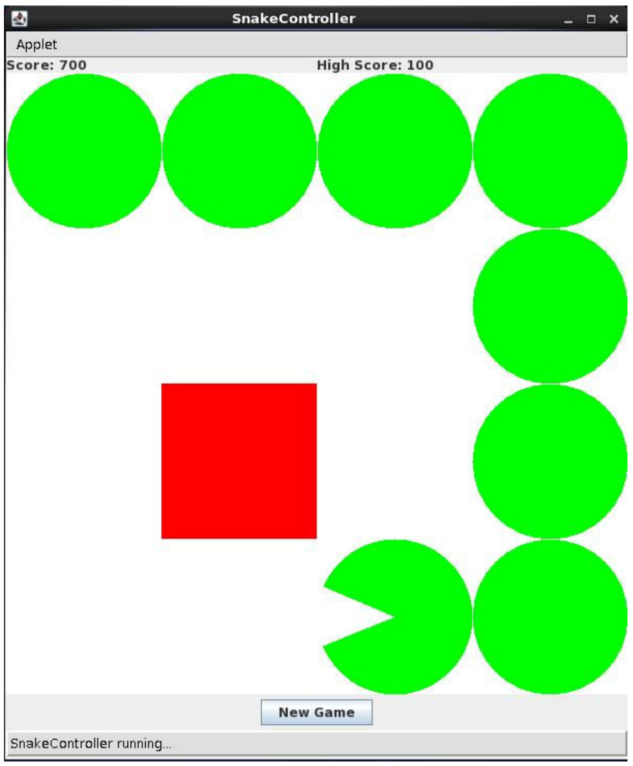


The snake is about to crash into the wall!

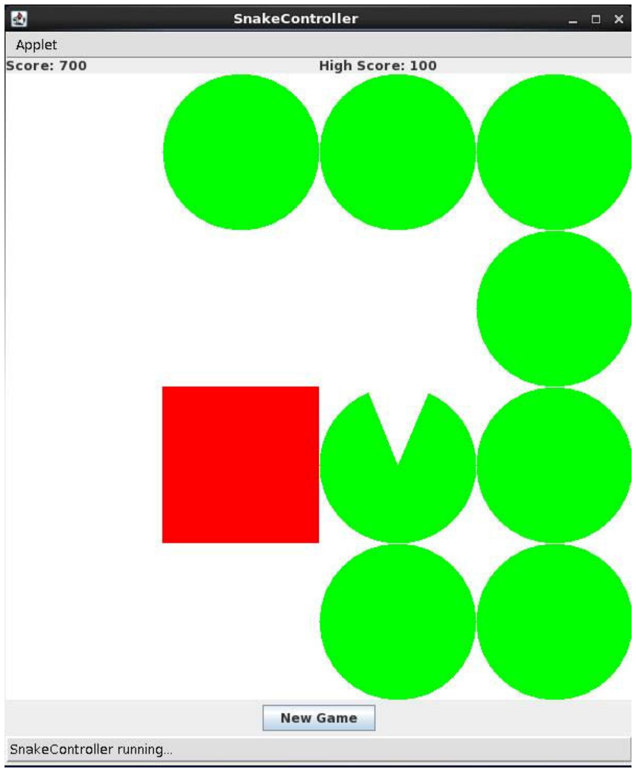


The snake crashed into the wall, so the game is over.

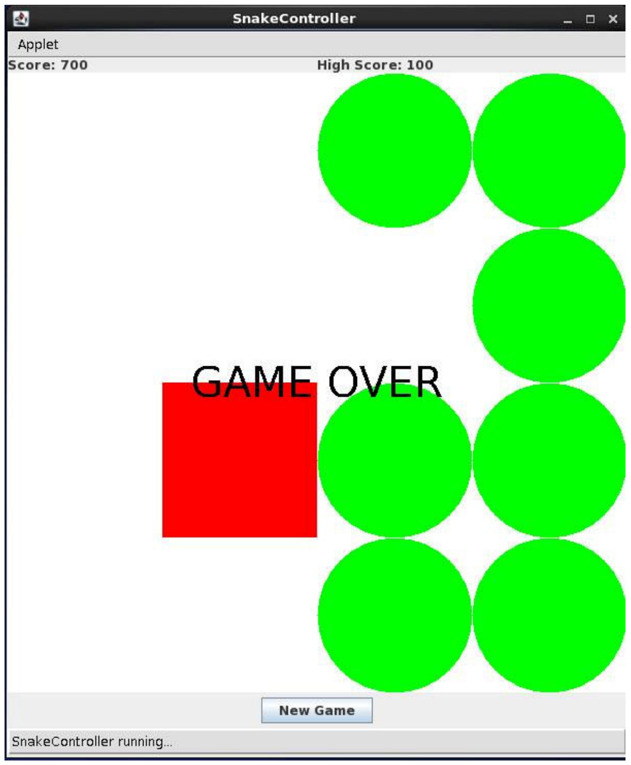
EATING ITSELF:



Playing the game normally, user hits the up key.

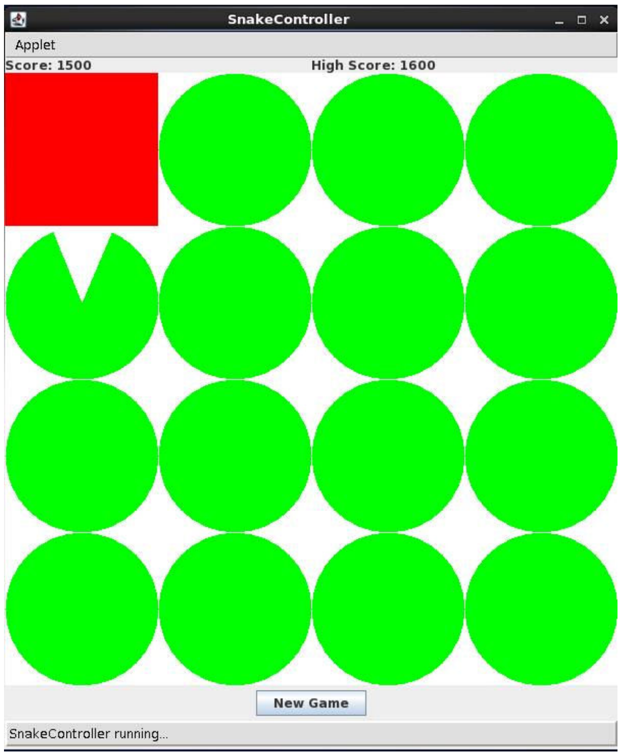


If the user presses right, the snake will crash into itself!

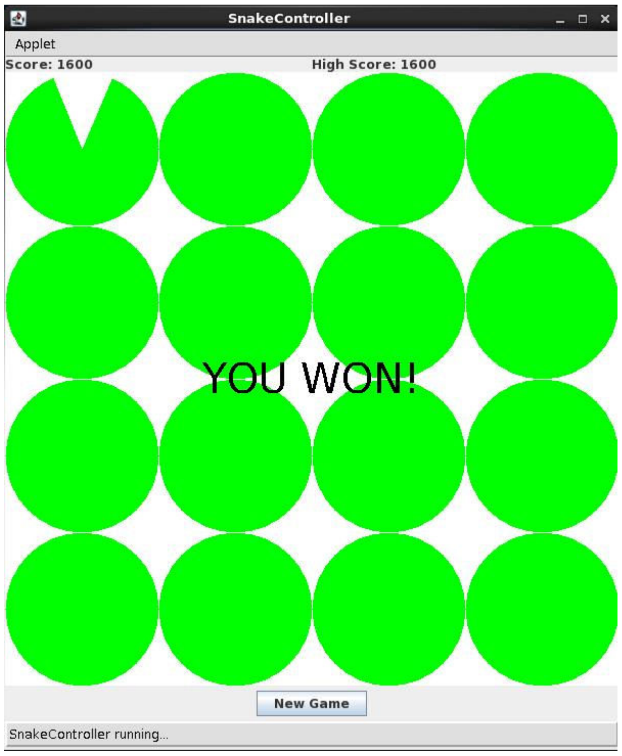


This is what it looks like when the snake crashes into itself!

WINNING:

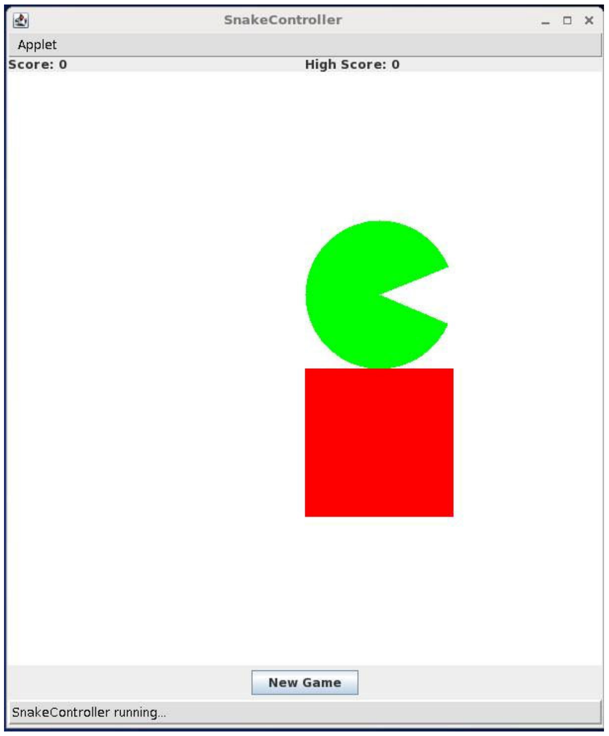


The snake is about to cover the entire canvas!

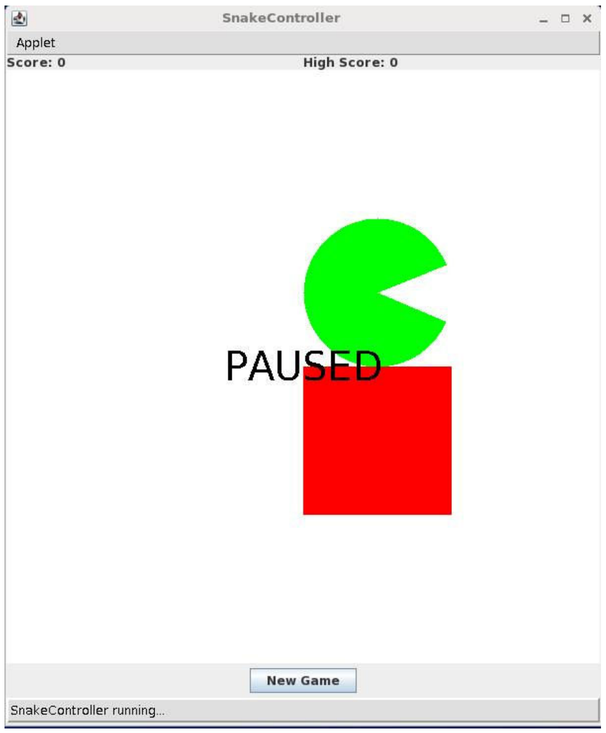


An apple cannot be placed on the canvas, so the user won the game!

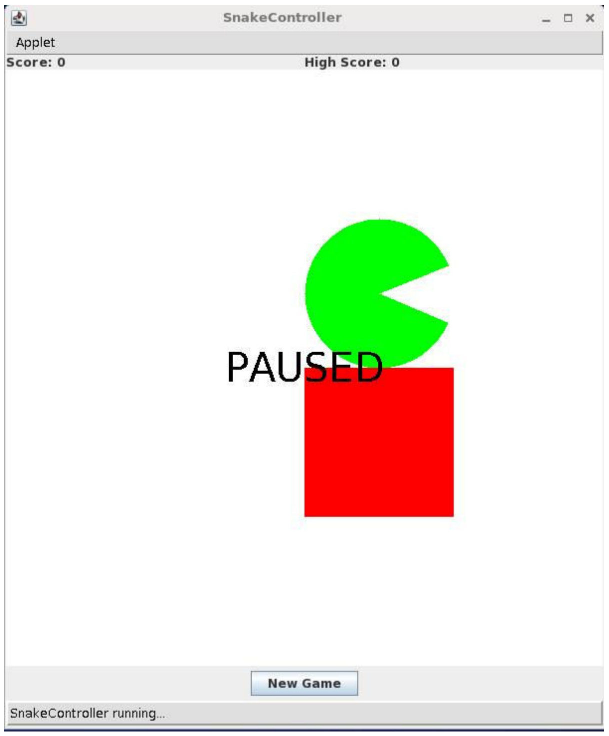
PAUSING:



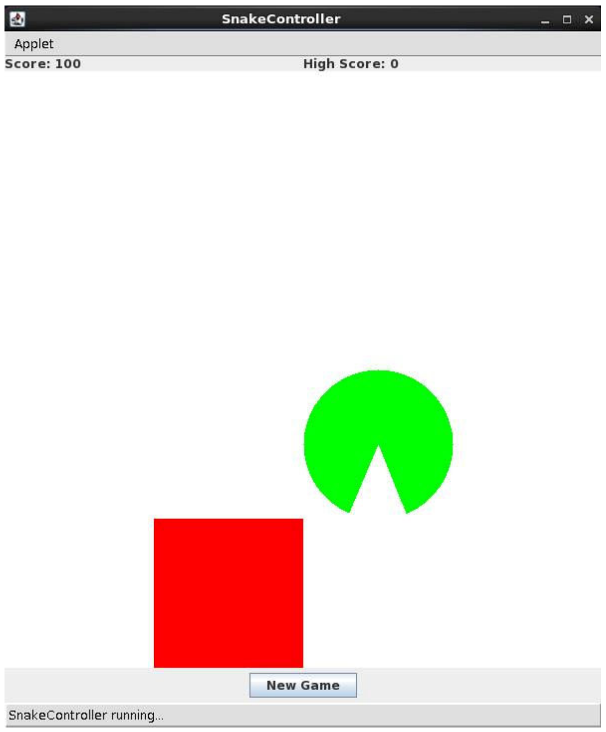
The user is about to press pause!



The user pressed pause!



The user presses down. Nothing happens!



The user presses space again, un-pausing the game (and immediately presses down)!

Starter Code:

```
public class Snake extends ActiveObject implements KeyListener {

    // How much the Snake grows by when it eats an apple.
    private static final int GROW_BY = 1;

    // The number of cells the snake has left to grow.
    private int leftToGrow;

    // The diameter of each SnakeSegment.
    private int size;
    // The delay between each pause in run.
    private int delay;
    private DrawingCanvas canvas;

    // Which way the snake is going.
    private Direction currentDir;

    // Whether the game is activated or not.
    private boolean isRunning = false,
            paused = false;

    // The coordinate the snake needs to go to in order to grow
    private Coordinate nextApple;

    // The snake is a collection of segments.
    ArrayList<SnakeSegment> snake;

    // We need to know where the head is for apple eating and crashing
    SnakeSegment head;

    SnakeController controller;

    public Snake(Coordinate coord, int size, int delay, DrawingCanvas canvas,
                 SnakeController controller) {
        ...
    }
    …
    private class SnakeSegment {
        // visible appearance of the snake.
        private final Color SNAKE_COLOR = Color.GREEN;
        private FilledArc segment;
        // the location of each snake segment.
        private Coordinate coord;
        // head constants
        private static final double UP_ANGLE = 90 + 22.5;
        private static final double LEFT_ANGLE = 90 + UP_ANGLE;
        private static final double DOWN_ANGLE = 90 + LEFT_ANGLE;
        private static final double RIGHT_ANGLE = 90 + DOWN_ANGLE;
        private static final double HEAD_ARC_ANGLE = 360 - 45;
        private static final double BODY_ARC_ANGLE = 360;
        /**
         * Creates a snakeSegment, by putting its parts on the canvas.
         *
         */
        public SnakeSegment(Coordinate coord, int size, boolean isHead,
                            DrawingCanvas canvas) {
            …
        }
}

public class SnakeController extends WindowController
                             implements ActionListener, KeyListener {
    private static final int Y_PADDING = 50;
    private static final int LINUX_MIGHT_HAVE_THIS_EXTRA_WEIRD_PADDING = 6;
    private static final int MIN_DIM = 500;
    private static final int MAX_DIM = 800;
    private static final int MAX_SPEED = 100;
    private static final int MIN_SPEED = 1000;
    private static final int MIN_SIZE = 20;
    private static final int MAX_SIZE = 400;
    private static final int NUM_TOP_COLUMNS = 2;
    private static final int NUM_TOP_ROWS = 1;

    private static int dimensions;

    private int size;
    private int delay;
    private int score;
    private int highScore;

    private JLabel scoreLabel;
    private JLabel highScoreLabel;

    private JButton newgame;

    private Text gameOverText;
    private Text winText;
    private Text pauseText;

    private boolean gameOver;
    private boolean won;
    private boolean paused;

    private Snake snake;

    private FilledRect apple;

    private Random randomIndexGenerator = new Random();

    ...
    public SnakeController(String [] args) {
        ...
    }
}
```

WARNING:MUST READ

1) DO NOT use any data structures that have not been covered in class (HashMap, HashTable, HashAnything). You must only use ArrayList.
2) DO NOT use any static variables to communicate between classes.
3) DO NOT start late. This will be a long, difficult assignment. There will be no mercy for those that start late.

EXTRA CREDIT (5 points)

For this extra credit, create two new files: SnakeControllerEC.java and SnakeEC.java. If you do not make these extra credit files, you 1) will not get extra credit and 2) your file will be graded as if that is your regular file. TL;DR: Make separate EC files or your grade will suffer!

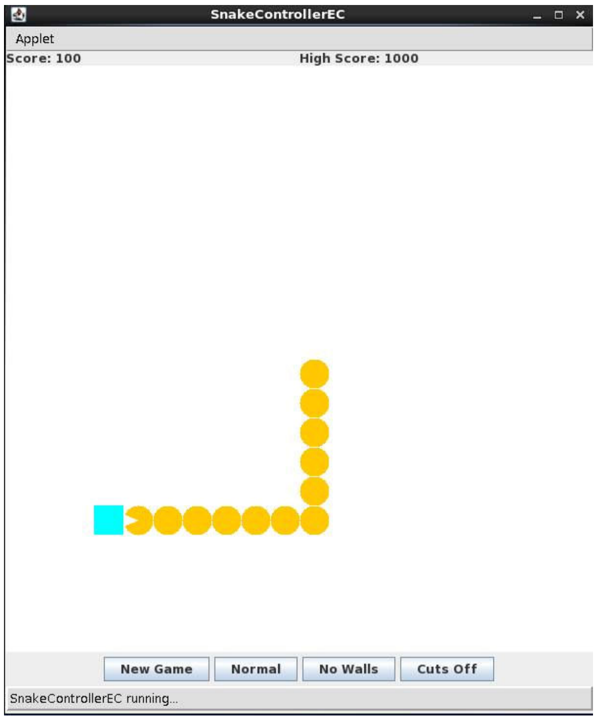
```
> cd ~/pa8
> cp SnakeController.java SnakeControllerEC.java
> cp Snake.java SnakeEC.java
```

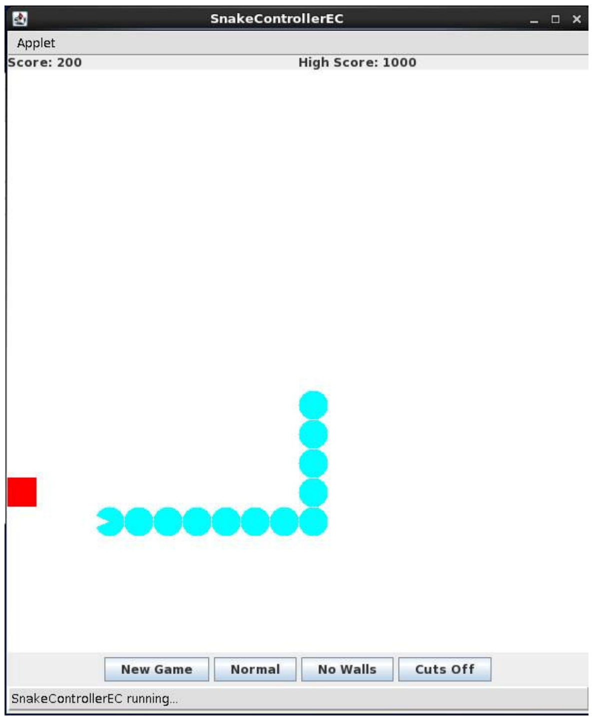
1) **Changing color to match fruit [1 point]**

Have the snake start off in its traditional green color. The fruit's color, however, should be randomly selected from this array:

```
Color [] colors = {Color.RED, Color.ORANGE, Color.GREEN, Color.CYAN, Color.MAGENTA};
```

The color of the fruit should never match the color of the snake. When the snake eats the fruit, the snake should change color to match the fruit, and a fruit should be placed that's a different color than the snake's new color. You might need to modify the constructors' parameter lists and add methods to accomplish this task.





Right before it changes color.

Right after it changes color and before it grows by one SnakeSegment

2) Wall Hack (No Walls Mode) [2 points]

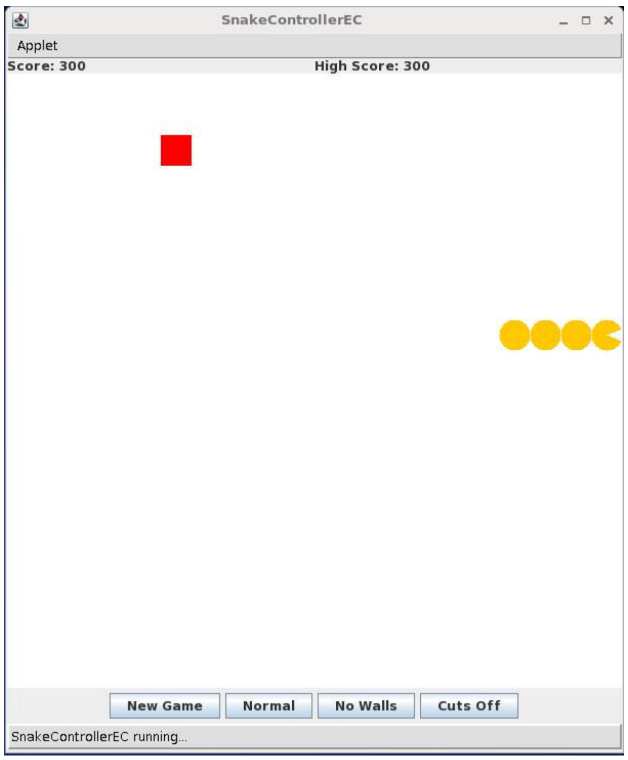
In this game mode your snake will have the ability to pass through the walls, and reemerge from the other side. We will begin as usual, with the snake a single SnakeSegment that moves only when an arrow key is pressed. The only difference in gameplay will be when the snake comes in contact with a wall. Instead of the game ending, the snake should move such that it emerges from the opposite wall and continues moving along the same horizontal or vertical line (until an arrow key is pressed).
For example, if the snake hits the right side of the frame, then the head of the snake should appear on the left side of the frame, at the same horizontal level on the next move.

<u>GUI changes:</u>

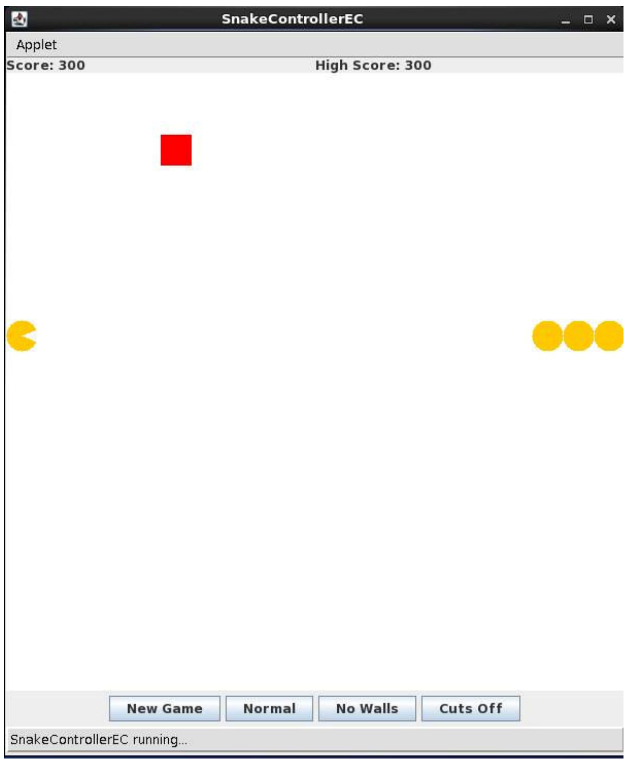
To start the game mode you will need to have a JButton called “No Walls”. If you decide to do this extra credit, you will also need to create a JButton called “Normal”, so you can switch between the different modes.

The application should start in Normal mode, with the Normal button selected. Only when the No Walls button is selected should the snake be able to move through the walls. While you are playing the game, pressing the mode buttons will not change the game’s mode. It will behave as if it ignored the button press. It will also ignore the press if it is pressed when the game over text is displayed. It should only switch modes after pressing New Game, but before starting the game (by moving the snake). The New Game button, however, should still work mid-game. Pressing New Game will keep the game in whatever mode it was previously in.

The snake can still die if it eats itself, and if it does, game over is triggered as normal. Again, when the game is restarted, the game should remain in whatever mode it was last.



Before the head teleports to the other side.



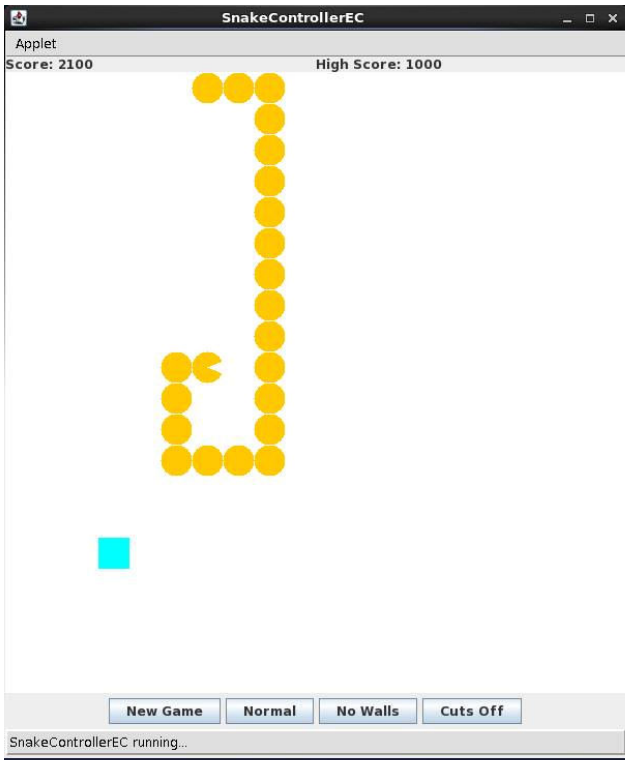
After the head teleports to the other side.

3) Snake eat Self (Cut Off Mode) [2 points]

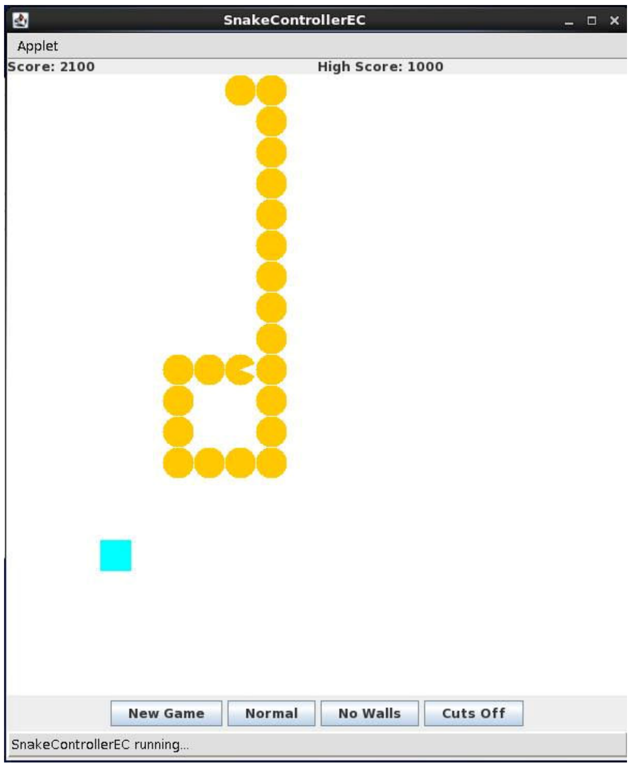
In this game mode, the snake learns a cool trick from its reptilian cousin, the lizard. The snake loses it's tail if it bites itself. Gameplay works as normal (as in, crashing into the walls results in game over)**.** The only change we will make for this mode is that you no longer die when the snake bites itself. To implement this game mode you will need to find out which segment the snake ate. Since we maintain an arraylist for the snake body, we can find the index at which the head comes in contact with the body. Once the correct index has been found, all elements from that index to the end of the snake should be removed from the arraylist and the canvas. Each segment lost should make a deduction from the total points. A player loses 200 points per segment lost. Because of this, the score can be negative. However, the high score will never be updated to be negative (remember, the high score is only updated at the end of the game). It will always be at least 0.

GUI changes:

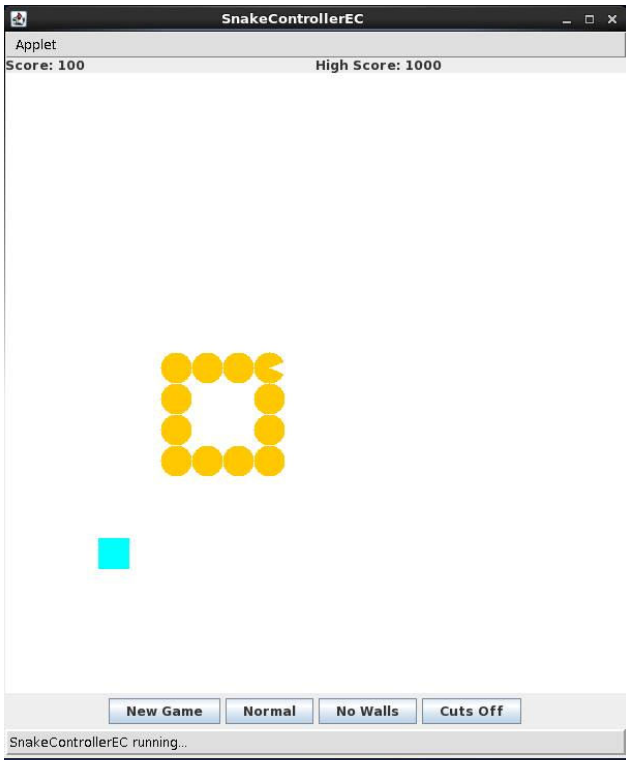
To start the game mode you will need to have a JButton called "Cuts Off". Every new game should begin in Normal Mode. Only when the Cuts Off button is pressed should the snake be able to eat itself/lose segments. The button behavior should be exactly the same as what is specified for part 2 of the extra credit. You can do this extra credit independently of part 2, but the buttons must have the same behavior. If you decide to do both extra credit modes, then you just need to make sure you include all three buttons (Normal, No Walls, and Cuts Off).



In the middle of playing Cuts Off mode



About to eat itself!!! Oh no!!!



Its tail just got cut off! Also, notice how the score changed!!!

Turnin

To turnin your code, navigate to your home directory and run the following command:

```
> cse11turnin pa8
```

You may turn in your programming assignment as many times as you like. The last submission you turn in before the deadline is the one that we will collect.

Verify

To verify a previously turned in assignment,

```
> cse11verify pa8
```

If you are unsure your program has been turned in, use the verify command. **We will not take any late files you forgot to turn in.** Verify will help you check which files you have successfully submitted. It is your responsibility to make sure you properly turned in your assignment.

Files to be collected:

Snake.java
SnakeController.java
Coordinate.java
Direction.java
Acme.jar
objectdraw.jar

Additional files to be collected for Extra Credit:

SnakeEC.java
SnakeControllerEC.java

The files that you turn in must be EXACTLY the same name as those above.

NO LATE ASSIGNMENTS ACCEPTED.

DO NOT EMAIL US YOUR ASSIGNMENT!

Start Early and Often!