Programming Assignment 8 (PA8) - SnakeGame

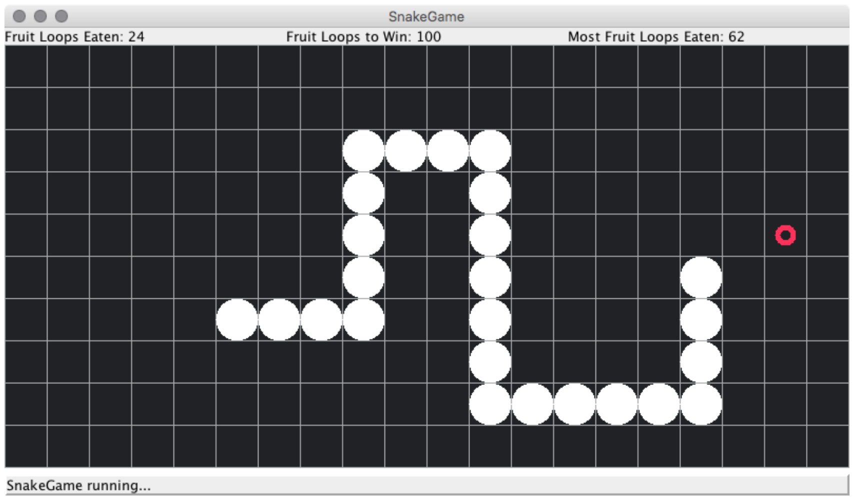
Due Date: **Wednesday, November 21 @ 11:59 pm**

Assignment Overview	Grading	Gathering Starter Files	Command Line Arguments	SnakeGame
Sample Output	Sample Screenshots	README File	Extra Credit	Turnin Summary

Assignment Overview

This programming assignment is a simplified version of the classic game Snake. For those that are not familiar with the game Snake, it is a game where you are a snake and your goal is to move around and eat pieces of food (in our case, Fruit Loops). Once the game starts, the snake moves continuously in the direction you tell it to. Each time the snake eats a piece of food (a Fruit Loop), it grows. The end goal is to eat enough food such that the snake covers the entire game area. You lose the game if the snake runs into itself or the borders of the game area (as you can imagine, this becomes more difficult as the snake fills up more of the game area).

For a more general idea of how Snake works, you can look at this Wikipedia link. Remember though, that the wikipedia link **is not** a substitute for the writeup; it should only be used to understand how the game works in general if you are unfamiliar with it. Implementation of this game must follow the exact specifications detailed in this writeup.



Grading

- **README: 10 points** - See README Requirements here and questions below
  - http://cseweb.ucsd.edu/~ricko/CSE11READMEGuidelines.pdf
- **Style: 20 points** - See Style Requirements here
  - http://cseweb.ucsd.edu/~ricko/CSE11StyleGuidelines.pdf
- **Correctness: 70 points**
- **Extra Credit: 5 points** - View Extra Credit section for more information.

NOTE: If what you turn in does not compile, you will receive 0 points for this assignment.

Gathering Starter Files

You will need to create a new directory named pa8 and go into that directory. The `$` represents your command prompt. What you type in is in **bold.**

```
$ mkdir ~/pa8
$ cd ~/pa8
```

Copy the starter files from the public directory:

```
$ cp ~/../public/objectdraw.jar .
$ cp ~/../public/Acme.jar .
$ cp ~/../public/PA8StarterCode/* .
```

Starter files provided:

objectdraw.jar	Direction.java	PopupMessage.java
Acme.jar	PA8Constants.java	

Command Line Arguments

Overview:
There are 4 main components to the implementation of this version of Snake: the grid, the snake, the fruit loops, and the controller. The snake will move around on the grid, with the goal to eat fruit loops placed randomly on the grid. In this version of the game, the game is won once the snake eats a specified number of fruit loops. The controller will be responsible for taking in user input through the keyboard (where arrow keys will be used to move the snake) and directing the snake on what to do based on the input. The controller is also responsible for handling all the game logic (pausing, winning, losing, restarting, etc).

Command Line Arguments:
There are 3 optional command line arguments for this program. All arguments are optional, except if you provide an argument, all the previous arguments need to be provided as well (otherwise, how would we know which argument we are supposed to interpret the arg string as?).

Here is the usage statement for the program:

```
Usage: SnakeGame [rowsXcolumns [loopsToWin [delay]]]
  rowsXcolumns: size of the grid; 'X' is not case sensitive
    -- rows must be valid integer in the range [2, 25]
    -- columns must be valid integer in the range [2, 45]
    -- defaults to 20x18
  loopsToWin: the number of fruit loops that must be eaten to win
    -- must be valid integer in the range [1, (rows * columns) - 1]
    -- defaults to (rows * columns) - 1; (the full grid)
  delay: length of the animation pause measured in milliseconds
    -- must be valid integer in the range [50, 1000]
    -- defaults to 120 milliseconds
```

Arg 1: rowsXcolumns
As mentioned before, the snake travels on a grid, so naturally one of the command line arguments is to specify the grid size, giving the number of rows and columns the grid will have. This should be a string formatted as `rowsXcolumns`, where `rows` is replaced with the number of rows, `columns` is replaced with the number of columns, and `X` (the delimiter between the `rows` and `columns`) can be uppercase or lowercase. For example, the default value for this argument is `20x18`, which translates to a grid with 20 rows and 18 columns.

Arg 2: loopsToWin
The next command line argument, `loopsToWin`, specifies the number of fruit loops the snake has to eat in order to win the game. In most versions of Snake, you win the game once the snake fills the entire game area (which you achieve by eating the pieces of food). In our version of snake, you can specify the number of fruit loops (pieces of food) the snake needs to eat in order to win. If this argument isn't specified, it defaults to the number of cells in the grid - 1. The minus 1 is because the head of the snake automatically takes up 1 cell in the grid without needing to eat a fruit loop to make it appear.

Arg 3: delay
The next command line argument, `delay`, represents the animation delay for the snake measured in milliseconds. This is the length of time the snake will pause in each iteration of the run loop. This correlates to how fast the snake moves, and therefore how difficult the game is. The smaller the delay value, the faster the snake will move; the larger the delay value, the slower the snake will move. The game is easier when the snake moves slower, so you'll want to use a larger delay value when testing your program.

Now that you have a basic overall understanding of the program and the command line arguments, now would be a good time to go look at the demo video (link posted on piazza) to help solidify everything we just went over.

Parsing the Command Line Arguments:
Start off by creating a new file called ***SnakeGame.java***. This will be the main controller for the game that contains all the game logic; as such, it needs to extend WindowController. The first thing we want to do in this class is write a main method where we will parse the command line arguments. Many of the arguments will be parsed in a similar manner, so it is in your best interest to create a few helper methods to help with argument parsing (more tips on this later). Now would also be a good time to look in PA8Constants.java--there is an entire section dedicated to parsing the command line arguments, which will help give you some context before reading through the rest of Stage 1.

Check Number of Arguments:
First, we need to make sure the user didn't enter too many arguments. If they did, print the appropriate error message, print the usage statement, and exit with code 1. You'll notice here that the usage statement has a lot of format specifiers to fill in. Do yourself a favor and make a static helper method (yes, you read that right: *static* helper method) to print the usage statement. Doing so means you'll only have to fill in the format specifiers *once*, so there's only one place in your code where you might make a mistake (so double triple check you don't make any mistakes here!).

Special Case: rowsXcolumns
For the first argument, `rowsXcolumns`, we need to first extract the `rows` and `columns` into two separate strings, with the `X` delimiter being case insensitive (meaning it can be either uppercase or lowercase). There are two methods in the String class you'll want to use for this (one to ensure the `X` is case insensitive, and one to extract the `rows` and `columns` into two separate strings). If you aren't able to extract the two separate strings from the argument, this means the user didn't format the string properly, so print the appropriate error message, print the usage, and exit with code 1.

Parse All the Things:
Now that we have all the arguments separated into their own strings, we can run through the following checks in the order they are listed below. If an argument isn't provided, it should be given the appropriate default

value. **For each argument, as soon as an error is detected, print the appropriate error message, print the usage, then exit with code 1.** Only 1 error should ever be printed before the program exits.

	Checks to Perform	Default Value
`rows`	• Must be a valid integer • Must be within the range [2, 25] inclusive	`20`
`columns`	• Must be a valid integer • Must be within the range [2, 45] inclusive	`18`
`loopsToWin`	• Must be a valid integer • Must be within the range [1, (`rows` * `columns`) - 1] inclusive	`rows * columns - 1`
`delay`	• Must be a valid integer • Must be within the range [50, 1000] inclusive	`120`

Handling Invalid Integers:
You'll notice that all the arguments need to be converted from a string to an int. We can save ourselves a lot of headache by making another static helper method here to perform the conversion and print an error message (and usage and exit) in the case of an invalid integer. Note that this is the ONLY place in your code where you should have a try-catch block.

In addition to passing the string (to be converted to an int) as a parameter, you should also pass in the string containing the name of the value you are trying to convert. This way you'll be able to fill in the %s format specifier in the error message that says the name of the value that caused the error (the other %s should contain the erroneous string that couldn't be converted to an int). For example, if `loopsToWin` is the string `"4hundred2wenty"`, the error message should say:

```
Error: loopsToWin (4hundred2wenty) must be a valid integer
```

Handling Integers Out of Range:
You'll also notice that once each argument is (successfully) converted to an int, it needs to be within a specific range. Here we can also save ourselves some headache by creating another static helper method to check if a value is within its appropriate range, and print an error message (and usage and exit) otherwise. In order to make this method modular, you should pass in the minimum and maximum accepted values (in addition to other parameters) so it can be used to check each argument, despite them having different ranges.

Note that `loopsToWin` presents a special case when checking its range because it has a different error message for when the value is out of range than the other arguments do. Simply check this argument separately without using the helper method.

If All Arguments are Successfully Parsed:
If no errors are encountered while parsing the arguments, you should **print out the usage message** and then later we will be creating the grid and Acme MainFrame at the end of this main method (more info later).

Sample Output

A sample executable for command line parsing is provided in the public directory for you to try and compare your output against. Note that you cannot copy the executable to your own directory; you can only run it using the following command (where you will also pass in the command line arguments):

```
$ ~/../public/pa8test
```

The following are some examples of input to show how to run the public executable. You are responsible for running the public executable with these and many of your own examples to fully understand what the output should look like. Note that the output of your program must match the output of the public executable *exactly*, character-for-character, newline-for-newline. What you type in is shown in bold.

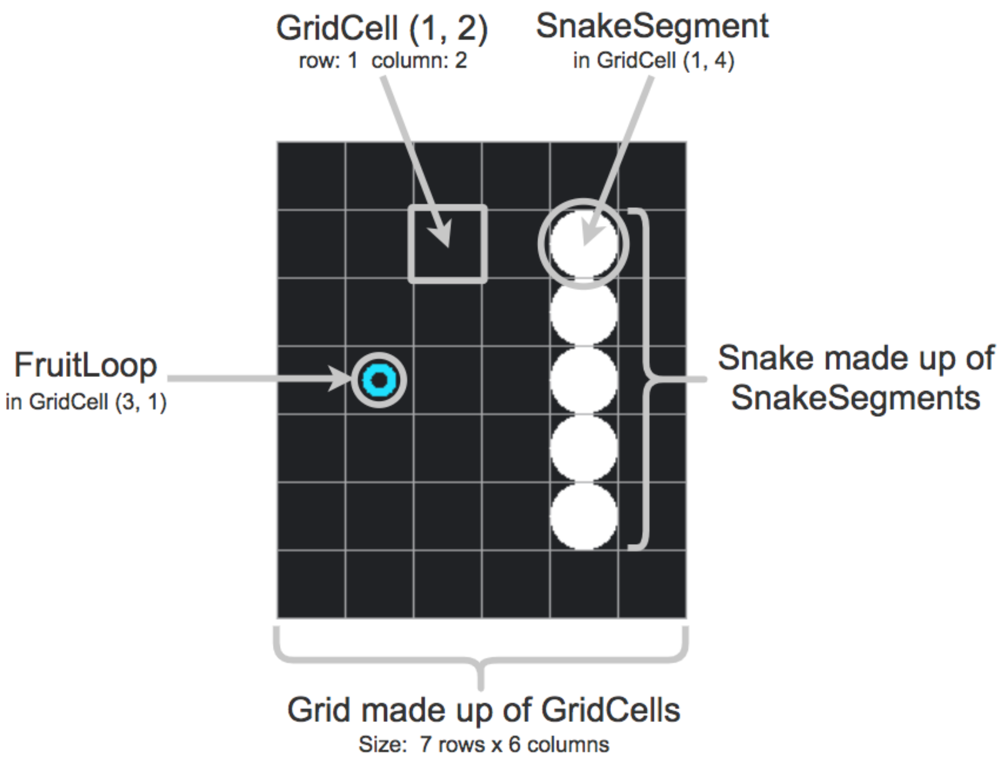
This executable does not implement the Snake game.
It will only show you the terminal output for both error cases and non-error cases.

No arguments, all values set to default:	`$ ~/../public/pa8test`
Too many arguments:	`$ ~/../public/pa8test arg1 arg2 arg3 arg4`
Grid size argument is not formatted correctly:	`$ ~/../public/pa8test 15,16 12 500`
Grid rows is not a valid integer:	`$ ~/../public/pa8test 10ax15`
loopsToWin is not a valid integer:	`$ ~/../public/pa8test 20X20 notAnInt`
delay out of range:	`$ ~/../public/pa8test 9x12 107 1420`

This list is NOT exhaustive--that's your job.

SnakeGame

You will be writing the following classes (and therefore .java files) to implement this program:
SnakeGame, ***Grid***, ***GridCell***, ***Snake***, ***SnakeSegment***, ***FruitLoop***



As we can see in the diagram above, a Grid is made up of GridCells, and a Snake is made up of SnakeSegments. We can also see that a GridCell can contain a FruitLoop or a SnakeSegment.

Back to Top

Important Implementation Notes:

No static variables. No static methods. The only exception is that you should have static helper methods to help with parsing the command line arguments (however, these methods should *not* contain any static variables). From the wise words of Steve McConnell in his book *Code Complete*, "the road to programming hell is paved with global variables."
Use helper methods! The key to completing this assignment will be to create many helper methods. If you do not create helper methods, this assignment will become very difficult very quickly. Some more wise words from McConnell: "Classes and routines are first and foremost intellectual tools for reducing complexity. If they're not making your job simpler, they're not doing their jobs."

Stage 1: Create the Grid

To create the grid, we'll be working in two different files: ***Grid.java*** and ***GridCell.java***.

GridCell:

This class represents a single cell in a grid. A GridCell needs to know its row and column index in the grid (which should be passed in as parameters to the constructor), and it should also calculate the position of its upper left corner. You can calculate the upper left corner of the GridCell based on its row and column index, and the size of a GridCell (`GRID_CELL_SIZE`).

A GridCell should also be able to hold a reference to a FruitLoop or a SnakeSegment. A GridCell should be empty upon creation (meaning its FruitLoop reference and its SnakeSegment reference are both null). Note that a GridCell should never contain both a FruitLoop and a SnakeSegment at the same time (this is something that will be handled outside of this class, but is worth being aware of).

A GridCell should have the ability to:

- return data it has
- indicate whether or not it is empty
- insert a FruitLoop or a SnakeSegment
- indicate whether or not it has a FruitLoop; indicate whether or not it has a SnakeSegment
- remove the FruitLoop or SnakeSegment

You should write a bunch of methods to implement these abilities (most of the bullet points will correspond to more than one method).

Grid:

Now that we have well-defined GridCells, we can define the Grid. A Grid has a specified number of rows and columns (passed into the constructor) and should contain a 2D array of GridCells. A Grid should also have a random number generator to help with selecting random cells in the grid (in this assignment you can create a Random object without passing in a seed for it--see the Javadocs for Random for how this works). As in previous assignments, you should only ever create 1 (**one**) Random object in the entire program.

A Grid should have the ability to:

- reset itself so that all the GridCells are empty
- return its height and width in pixels (you know the number of rows and columns, and the size of a GridCell...where are my Math-CS majors at?)
- return its centerpoint

Back to Top

- return the GridCell at the specified row and column number (if the GridCell is not within the Grid, return null to indicate that)

[The following abilities/methods will require some further explanation below]

- getCellNeighbor: given a GridCell and a Direction, return the neighboring GridCell in that Direction
- getRandomEmptyCell: return a random empty cell in the grid
- draw: draw the Grid on the canvas

`public GridCell getCellNeighbor(GridCell cell, Direction dir)`

In order to implement this method, you'll need to understand what Direction is. Take a moment to read through Direction.java, one of the starter files provided to you. This file defines a Direction enum. Click the following link for an explanation of Java enums.

In this method, our goal is to return the cell's neighbor in the direction passed in. For instance, if the cell passed in is at row 3, column 4, and the direction passed in is UP, we want to return the GridCell at row 2, column 4. Likewise, if the direction passed in was RIGHT, we would want to return the GridCell at row 2, column 5. In this method, we will want to make use of our other method we wrote to get a GridCell from the Grid, given its row and column (which already handles invalid GridCells properly).

Look at the x and y values for each of the Direction enum constants defined in Direction.java and see how you can use those values to calculate the desired neighbor's row and column indices (you will want to use the getX() and getY() methods in Direction). Note that x corresponds to columns, and y corresponds to rows. In other words, picture placing x, y axes on top of a grid and pointing at one of the GridCells: adjusting the x value would change which column you are pointing at, and adjusting the y value would change which row you are pointing at.

`public GridCell getRandomEmptyCell()`

In this method, simply generate a random row index, and a random column index, and then retrieve the GridCell at that location. However, it is not guaranteed that the GridCell will be empty. If the GridCell is not empty, keep getting a new random GridCell until we find an empty one.

In this case where there are no more empty GridCells, we should return null, but how do we know if all the GridCells are full? Since GridCells will be changing between empty and non-empty quite often (as the snake moves around), it would be very inefficient to check if every GridCell is full everytime we want a random empty GridCell. One way to solve this problem is to add an instance variable to Grid that will keep track of the number of empty cells. Since only GridCell knows when a cell becomes empty or full, we should modify the GridCell constructor to also take in a reference to the Grid. Then we can add a method in Grid that will adjust the count of empty cells (either incrementing or decrementing it). We can then call this Grid method from within GridCell whenever a GridCell is filled or emptied.

After making those adjustments to Grid and GridCell, figuring out when there are no more empty cells becomes trivial, because we can simply check our counter.

`public void draw(DrawingCanvas canvas)`

In this method, the Grid should draw itself on the canvas passed in. This will consist of 3 parts: drawing the background, drawing the horizontal lines between the rows, and drawing the vertical lines between the columns.

To draw the background, we simply need to draw a FilledRect that is the size of the grid. The color should be set to `BACKGROUND_COLOR`. Note that you don't need to store the FilledRect as an instance variable.

To draw the horizontal lines between the rows, we can create two Location objects for the beginning and the end of a line, starting at top of the grid. Then for every row (plus one extra, since if there are 3 rows, we want to draw 4 lines) we should draw a line with the current endpoints, and then move the endpoints down by the size of a GridCell. Make sure to set the color of the lines to `GRID_LINE_COLOR`. Note that you don't need to store the lines as instance variables. The vertical lines can be drawn in a similar fashion.

Back in SnakeGame.java:

Define a SnakeGame constructor to save the Grid, loopsToWin, and animation delay for the snake as instance variables. In other words, save all the information we've gathered from the command line arguments, or lack thereof, in SnakeGame.

Let's go back to where we left off in the main method in SnakeGame.java. Now that we've defined the Grid class, create a Grid as a local variable. Now we can finally finish off this main method by creating the Acme MainFrame. For now, let's set the window size to the width and height of the grid, and add 1 to both the width and the height. We are adding 1 here for the very last horizontal and vertical grid lines that will later be drawn on the canvas. The lines are 1 pixel wide, and the Grid width and height does not account for the extra 1 pixel.

Let's now add a definition for the begin() method in SnakeGame and in here draw the grid on the canvas. Create two new files for FruitLoop and SnakeSegment with empty class declarations so that your code can compile (for example: `public class FruitLoop {}`). Now compile and run your code, and you should see the grid drawn on the canvas. Try running your program with different grid sizes to make sure it works correctly.

Stage 2: Create the FruitLoops

To create the FruitLoops, we'll be working in ***FruitLoop.java***.

FruitLoop:

This class defines a single FruitLoop that can be placed in a GridCell. A FruitLoop is made up of 2 FilledOvals and contains a reference to the GridCell it is located in. To create a FruitLoop, we will need to pass in the GridCell to create it in, the Color we should make the FruitLoop, and the canvas to draw it on. In the constructor, make sure you save the reference to the GridCell, and also insert the FruitLoop into the GridCell.

Now let's draw the FruitLoop on the canvas. The outer FilledOval should be the color of the FruitLoop and should be the size of a GridCell divided by `OUTER_SIZE_DIVISOR`. The inner FilledOval should be the same color as the Grid's background, and should be the size of a GridCell divided by `INNER_SIZE_DIVISOR`. Center the FilledOvals in the GridCell.

A FruitLoop should have the ability to:

- be eaten, because what else do you do with fruit loops? (when a FruitLoop is eaten, it needs to be removed from the canvas and removed from the GridCell it was in)

Stage 3: Create the Snake

To create the snake, we'll be working in two different files: ***Snake.java*** and ***SnakeSegment.java***.

SnakeSegment:

This class represents a single segment of a snake. A SnakeSegment is made up of a FilledOval and contains a reference to the GridCell it is currently located in. To create a SnakeSegment, we will need to pass in the GridCell to create it in and the canvas to draw it on. In the constructor, make sure you save the reference to the GridCell, and also insert the SnakeSegment into the GridCell. To draw the SnakeSegment, its FilledOval should be the size of a GridCell, and should be centered in the GridCell it is in. Make the FilledOval white.

A SnakeSegment should have the ability to:

- return the GridCell it is in
- move to a new GridCell (you can assume that the new GridCell will be empty prior to moving)

<u>*Snake:*</u>
Now that we've defined SnakeSegments, we can define the Snake that is made up of SnakeSegments. Since the Snake will need to move around the Grid on its own, it needs to be an ActiveObject. The Snake will need to be passed a reference to the SnakeGame object, the grid, and the canvas. It will also need to be passed the delay that was determined in SnakeGame's main method so it knows how long to pause in each iteration of the run method. These should all be saved as instance variables in the constructor, as they'll be needed later.

Now let's set up the actual pieces of the snake. We first need to create a SnakeSegment for the head of the snake, placing it in a random empty GridCell. The head of the snake is what we will use to perform all collision checking and FruitLoop-eating. Next we will want to initialize an ArrayList of SnakeSegments for the body of the snake. This ArrayList should initially be empty, because we haven't eaten any FruitLoops yet. We recommend storing the head separately outside the ArrayList since it is treated differently than the rest of the snake when moving around the grid.

Lastly, we need to keep track of the Direction the head of the snake is moving. The direction should initially be `Direction.NONE`, because we want the snake to start out stationary.

A Snake should have the ability to:

- toggle between paused and unpaused

[The following abilities/methods will require some further explanation below]

- setDirection: update the snake's current direction to a new Direction
- move: move every SnakeSegment in the snake by one GridCell, checking for collisions and FruitLoops
- die: cross the rainbow bridge; cease to exist; end the animation loop of life
- run: while the snake is still alive, animate the snake by moving it around the grid (unless it is paused)

`public void setDirection(Direction dir)`

In this method, you should only update the snake's direction if it is <u>not</u> paused (the only thing the snake should respond to when paused is unpausing). If the snake doesn't have a body yet (hasn't eaten any FruitLoops), there are no restrictions on how its direction can change. If the snake does have a body, only update its direction as long as the new direction isn't the opposite of the current direction.

`private boolean move()`

This method should move each SnakeSegment in the snake by 1 GridCell and should return whether or not the move was successful. Let's start with just moving the head first, since the rest of the body will follow the head. When we move the head, we want to move it in the direction that the snake is moving, and we need to check for wall collisions, self collisions, and FruitLoop-eating. We only need to check for these things with the head, since the head will essentially clear the path for the body (the body can't possibly run into something that the head didn't run into). If the snake's movement direction is `Direction.NONE`, we can simply return right away, counting this as a successful move (if we didn't move anywhere, we know we didn't collide with anything, so we can consider that a success).

Step 1: Collision Detection
First get the current GridCell that the snake's head is in, and the next GridCell that we want to move the head to (hint: the current and next GridCells are neighbors). If the next GridCell is outside of the bounds of the Grid, or contains part of the snake, we can't move there because that would result in a collision (so return the appropriate value to indicate that the snake couldn't move). Next, check if there is a FruitLoop in the next GridCell. If there is, eat it, and take note that we will need to grow a new SnakeSegment at the end of this method.

Step 2: Move the SnakeSegments
Now we can begin the actual movement. Since we've guaranteed that the next GridCell is empty (because we've checked for walls, SnakeSegments, and FruitLoops), we can now move the head into the next GridCell without concern. Now loop through all the SnakeSegments in the body of the snake, moving them one at a time into the previous SnakeSegment's GridCell that it just moved out of. The end result should be that each SnakeSegment moved by 1 GridCell.

Step 3: Grow a New SnakeSegment
If we ate a FruitLoop earlier, grow a new SnakeSegment, placing it at the end of the snake. We also need to tell the SnakeGame controller that we ate a fruit loop, so it can keep track of scoring and spawn another fruit loop (write a method in SnakeGame for this).

`private void die()`

In this method, take note that the snake died, such that it will cause the run() loop to end. We also need to tell the SnakeGame controller that the snake died (write a method in SnakeGame for this; this method should lose the game).

`public void run()`

While the snake is still alive, the entire snake should move by 1 GridCell each iteration of the run loop. Don't move the snake if it is paused. If the snake couldn't move, it dies. Make sure you remember to pause in the run loop so we can actually see the animation.

Back in SnakeGame.java:
Now would be a good time to write some ***temporary*** code in your begin() method to make sure that you can display a Snake and a FruitLoop on the grid. If things are working properly, the Snake should show up as a single SnakeSegment and should not be moving. The FruitLoop should look like a fruit loop.

Stage 4: Implement the Gameplay

Now that we've created all the pieces we need for this program, we can finally tie it all together to create an actual game. All of the GUI components and game logic should be in ***SnakeGame.java***.

SnakeGame should have the ability to:

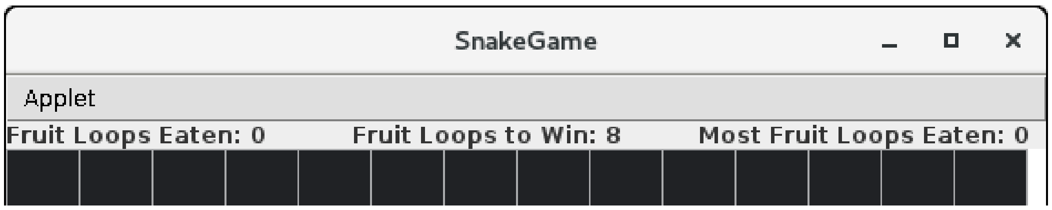
- begin: perform setup that should only happen once at the very beginning
- initGame: perform setup that should happen every time a new game is started
- place a FruitLoop randomly in the grid
- respond to keyboard input
- change the game state (pause the game, lose the game, win the game, restart the game)
- react to the snake eating a fruit loop

Back to Top

`public void begin()`

In this method, we need to perform all the initial setup that should only happen once. Make sure to remove any temporary code you had in here for testing previous stages.

Setup GUI Components:
Create a JPanel with a GridLayout and add the following labels to it. Make sure to call `this.validate();` after adding the panel to the window. When we originally set the window size back at the end of main, we only accounted for the size of the grid. Since we are now adding a panel to the window, we need to increase the height of the window by `SCORE_PANEL_HEIGHT`.



Setup the Game:
In order to setup the first game, we will want to delegate to our other method that performs setup every time a new game is started.

Setup Keyboard Input:
Since the SnakeGame will be controlled through keyboard input, we'll need SnakeGame to implement the appropriate listener interface. Note that any unused methods still need method headers (in the description explain that they intentionally don't do anything, and why they need to be there). On the very last line of the begin method, add the following line:

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

Without this line, we would need to first click on the canvas before SnakeGame would be able to receive any keyboard input. This line of code makes sure that the focus is set to the canvas, so we don't need to manually "set" the focus ourselves by clicking.

`private void initGame()`

In this method, initialize all the game logic (current state of the game, scoring) and game components (grid, snake, first fruit loop).

We'll also need to setup popup messages that will appear during the game. You have been provided with a file called PopupMessage.java which you should go look at now. This is a helper class to help display messages on top of the grid. In order to use it, you'll need to create a PopupMessage object (as an instance variable in SnakeGame) centered over the grid. Then create Text objects for each of the messages that will be displayed during the game (don't worry about the positions of these Text objects since PopupMessage will handle that for you). PA8Constants.java has constants that will help with this. The pause, win, and lose messages should have the `BIG_FONT_SIZE` and be bold. The restart message should have the `SMALL_FONT_SIZE` and not be bold. We recommend creating a helper method to create a Text object given a String, the font size, and whether or not it should be bold. Make sure the Text objects are initially hidden.

Overview of Gameplay:
Now that we've helped you with all the setup, it is up to you to tie everything together into a game using the following guidelines. We recommend guiding your logic in SnakeGame based on the current state of the game (there are constants in PA8Constants.java to help track the game state).

Back to Top

Before the games starts, the usage message mentioned should be printed to the terminal.

When the game starts, the Snake and the first fruit loop should be randomly placed on the grid. The fruit loop should be the first color in the `FRUIT_LOOP_COLORS` array. Cycle through the colors in this array for each successive fruit loop that is placed (see demo video and screenshots). The snake should initially be stationary.

While the game is running, you should be able to control the direction that the snake is moving using the arrow keys (see KeyEvent documentation for key code constants; use the non-numpad constants for arrow keys). Whenever the snake eats a fruit loop you should update the "Fruit Loops Eaten" label to reflect that. You should also place a new fruit loop in the grid to replace the one that was eaten unless the snake has eaten enough fruit loops to win.

Pressing the spacebar should toggle between paused and unpaused. While the game is paused, display the "Paused" popup message. Once the game is over (either because of a win or a loss), the snake should stop moving. The appropriate popup message should be displayed (either "You Win!" or "Game Over"), along with the instructions on how to restart the game. Every time the game ends, update the "Most Fruit Loops Eaten" label. You should only be able to restart the game (by pressing the 'r' key) when the game is over. Pressing 'r' while the game is running should have no effect. Pressing the spacebar once the game is over should have no effect.

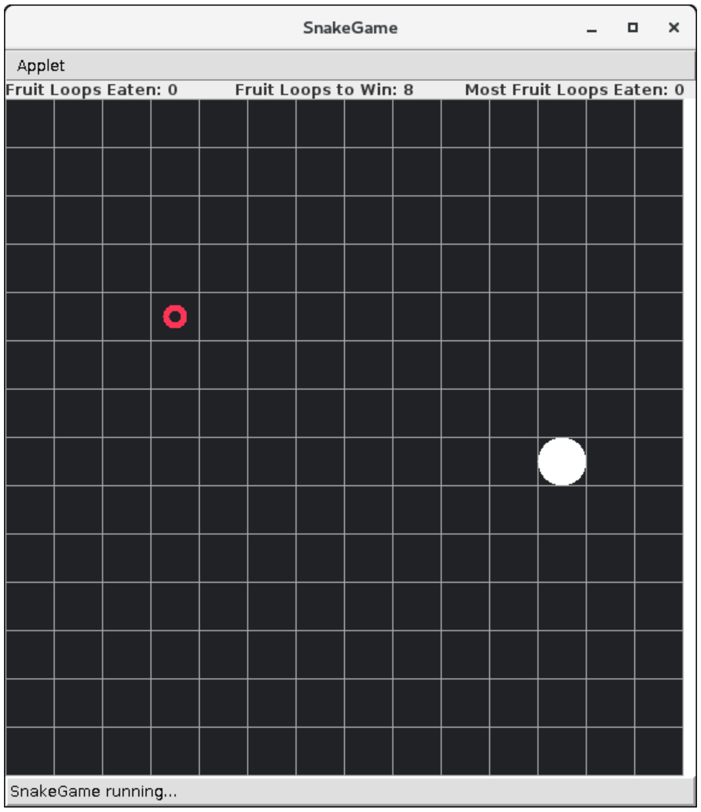
When displaying the "You Win!" or "Game Over" text (in a popup message), offset the text up by `TEXT_Y_OFFSET`. When displaying the restart instructions (in a popup message), offset the text down by `TEXT_Y_OFFSET`.

Sample Screenshots

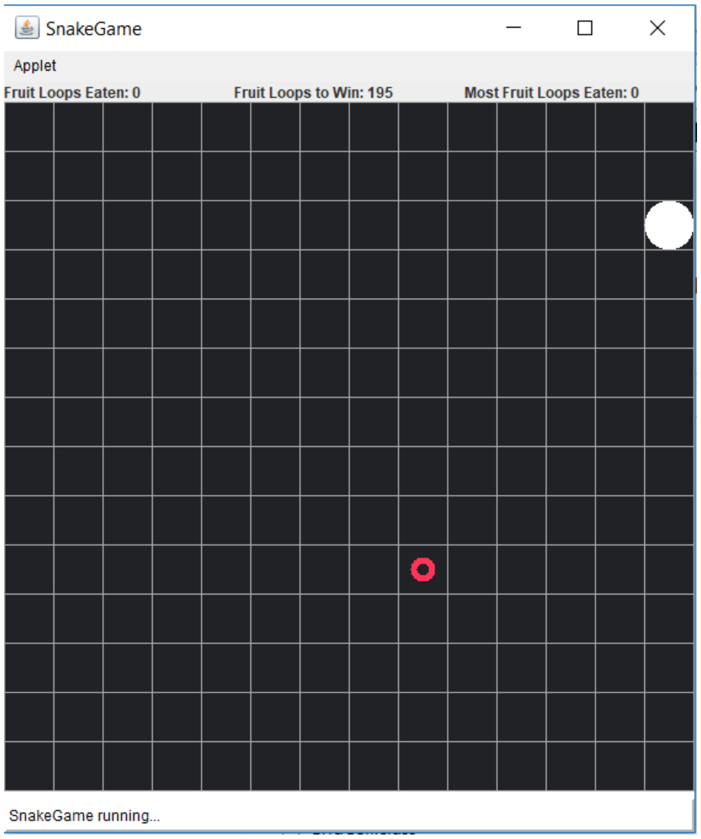
Note that some of the grid lines in the screenshots below may appear to be slightly transparent and/or have different widths. This is purely an effect from taking screenshots and then shrinking them down so they don't take up a million more pages than they already do. When you run your program, all of your grid lines should appear uniform.

You may also notice that when you run the program on different operating systems, the window size differs slightly. This is normal and you do not need to worry about this as long as your program matches what the screenshots below show:

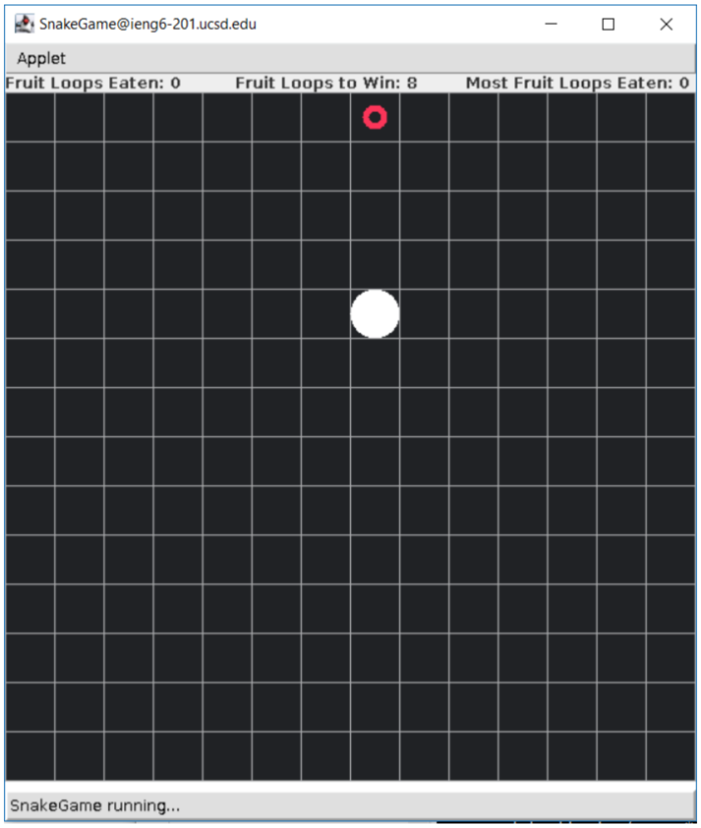
Back to Top



On a **lab computer**. The grid fits perfectly in the window vertically, but horizontally there is about 5px of white space to the right of the grid.

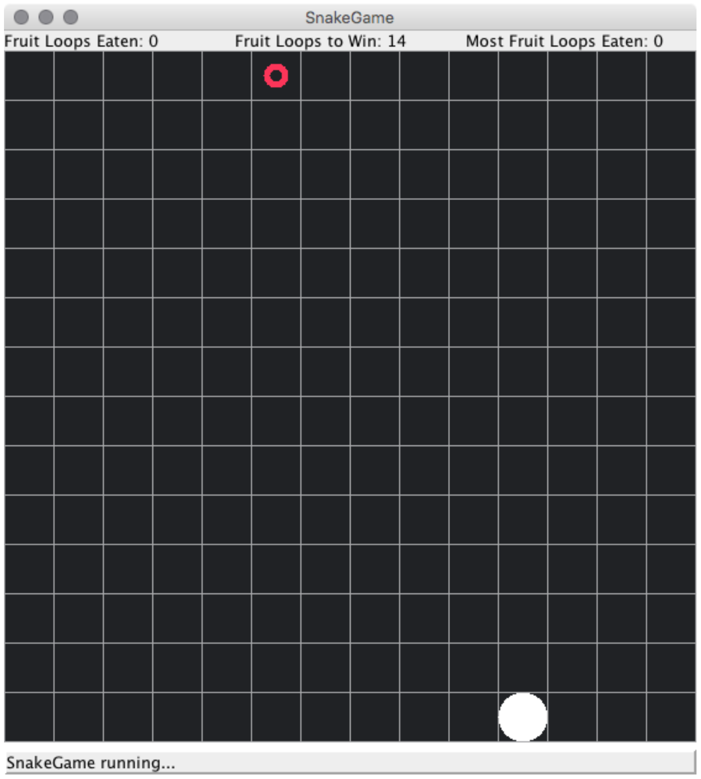


On a **Windows laptop, running locally**. Notice there is about 5px of white space below the grid, but there is no white space to the right of the grid.

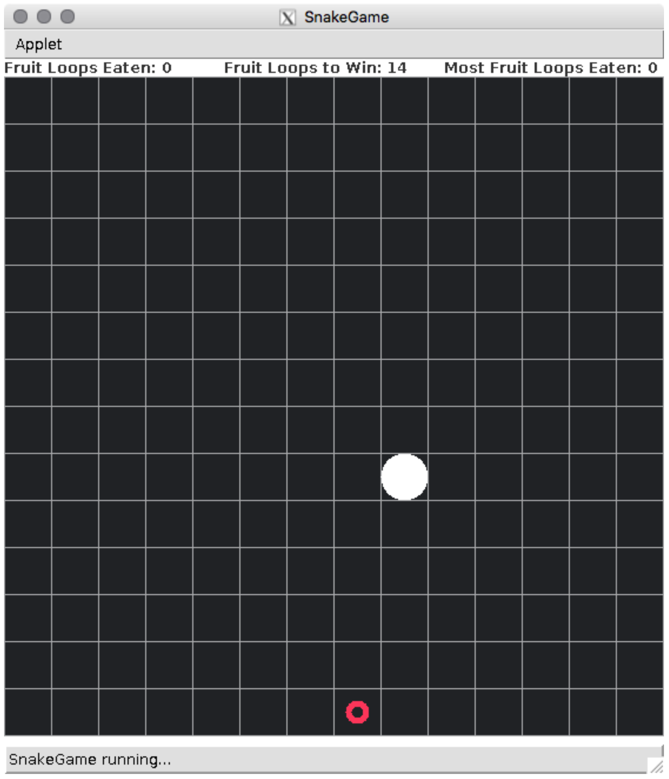


On a **Windows laptop, running through SSH** into ieng6. Notice how just like running locally, there is white space below the grid but no white space to the right of the grid.

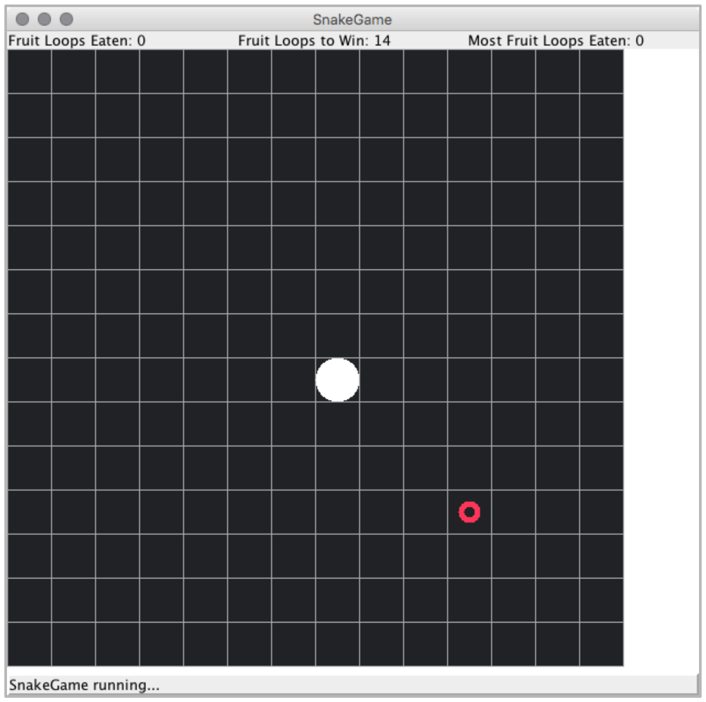
Back to Top



On a **Mac laptop, running locally**. There is about 5px of white space below the grid, but there is no white space to the right of the grid.



On a **Mac Laptop, running through SSH** into ieng6. There is about 5px of white space below the grid, but there is no white space to the right of the grid.

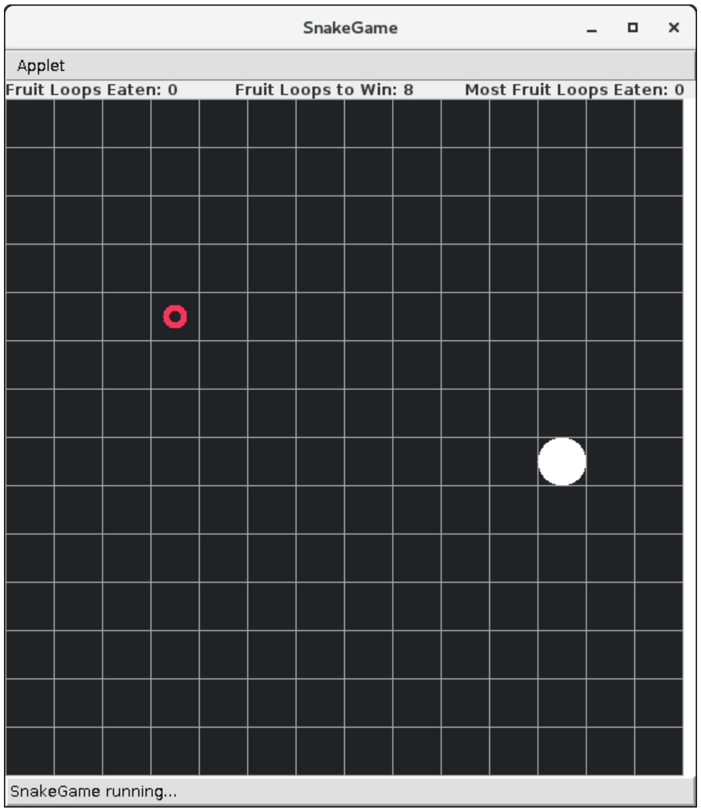


Note that sometimes when you run the program, the window size will not be the size that you specified (the window is much wider than it should be). You do not need to worry about this case. If this happens, just re-run your program until it has the expected window size. This tends to happen most often when running locally on a Mac (which is where this screenshot was taken).

See post @866 on piazza for an explanation of why this happens if you're curious.

Back to Top

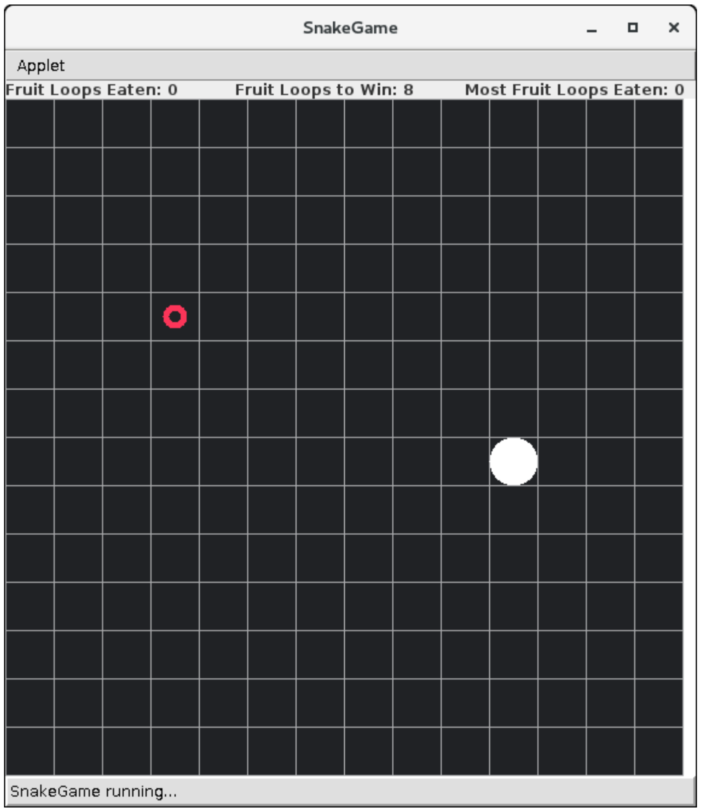
The following screenshots are taken on a **lab computer** to show you what the final game should look like:



Fruit Loops Eaten: 0   Fruit Loops to Win: 8   Most Fruit Loops Eaten: 0

SnakeGame running...

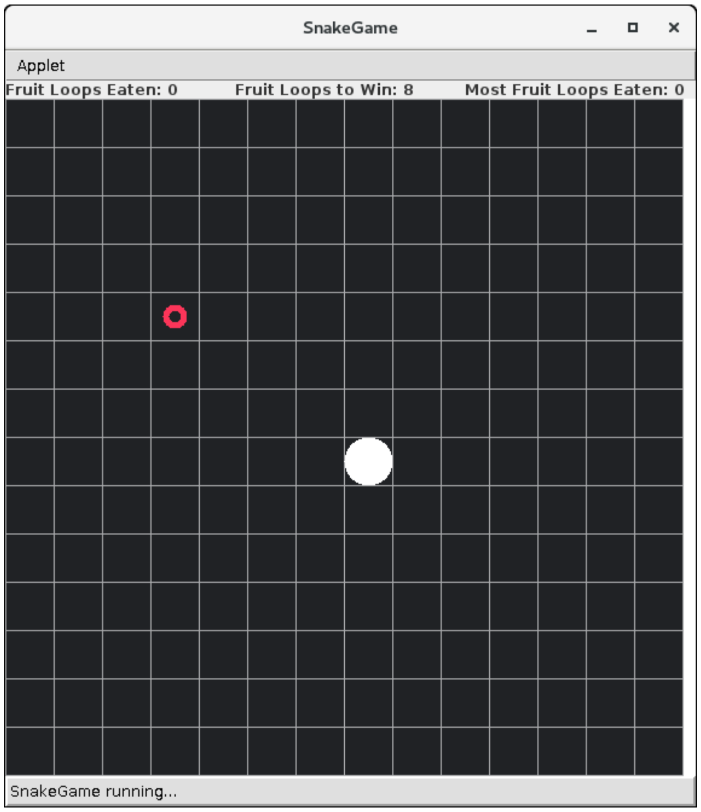
On startup. Snake and fruit loop are placed in random GridCells.



Fruit Loops Eaten: 0   Fruit Loops to Win: 8   Most Fruit Loops Eaten: 0

SnakeGame running...

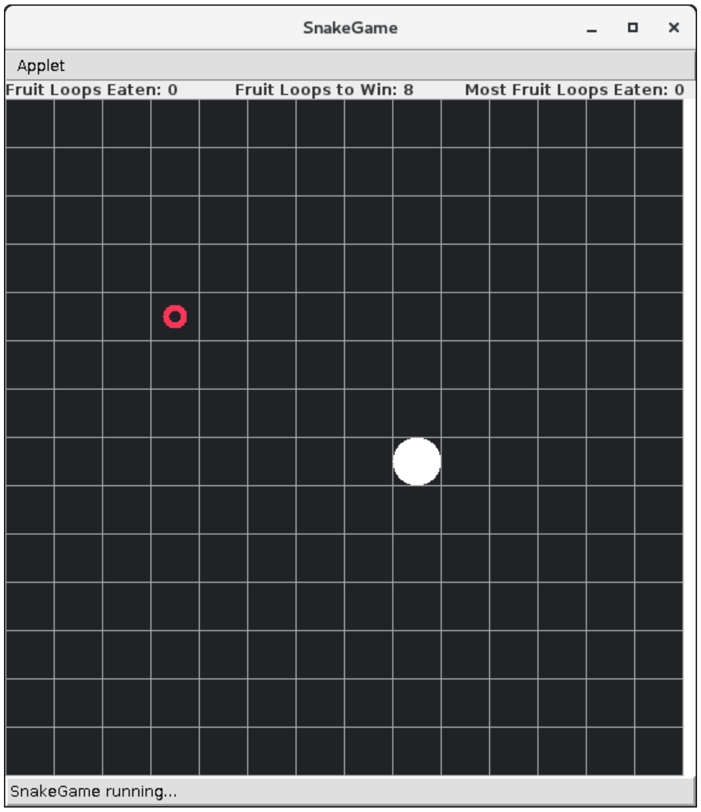
After pressing left arrow key, snake has moved one GridCell left.



Fruit Loops Eaten: 0   Fruit Loops to Win: 8   Most Fruit Loops Eaten: 0

SnakeGame running...

Snake continues to move left.

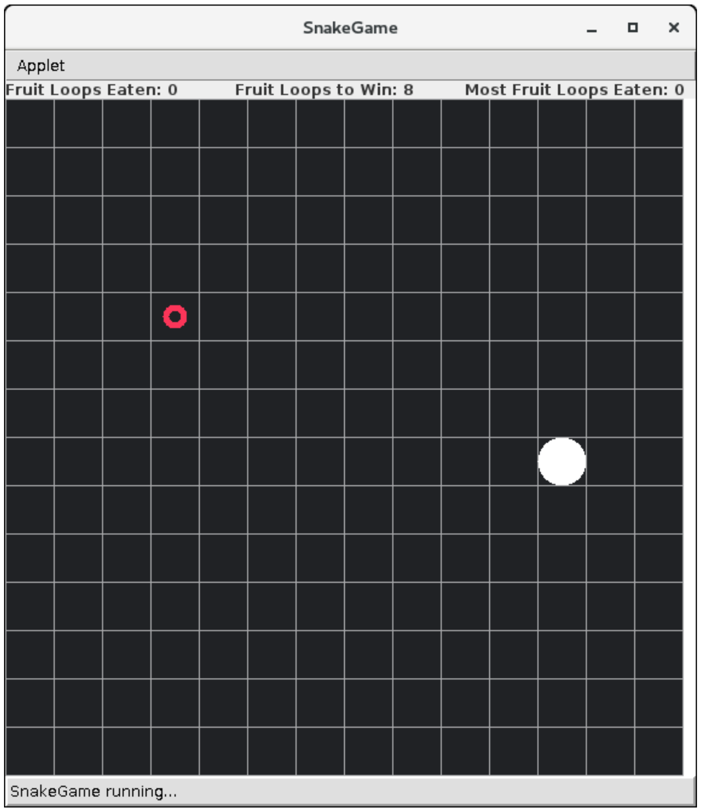


Fruit Loops Eaten: 0   Fruit Loops to Win: 8   Most Fruit Loops Eaten: 0

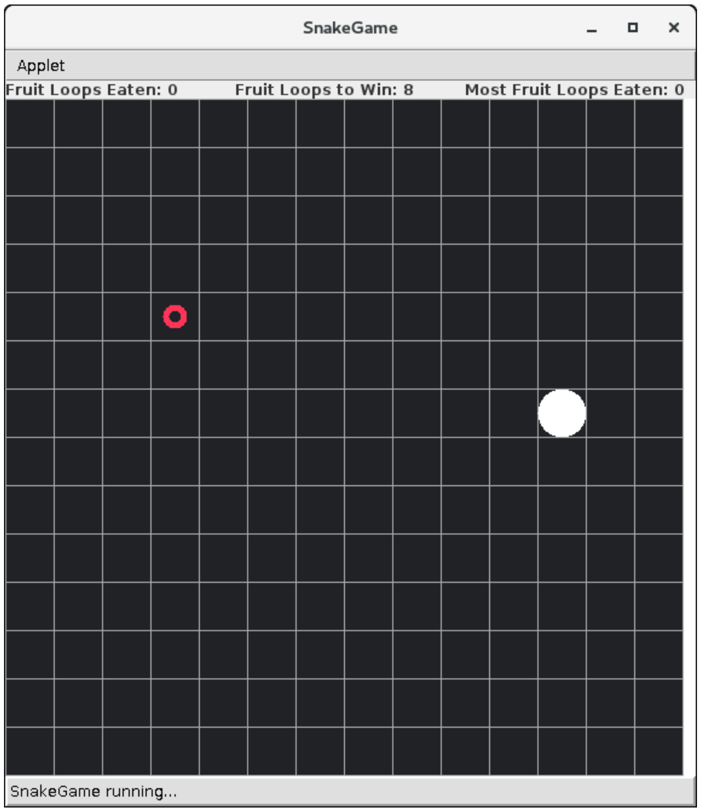
SnakeGame running...

After pressing right arrow key, the snake is now travelling in the opposite direction (it is able to do this since it doesn't have a body yet).

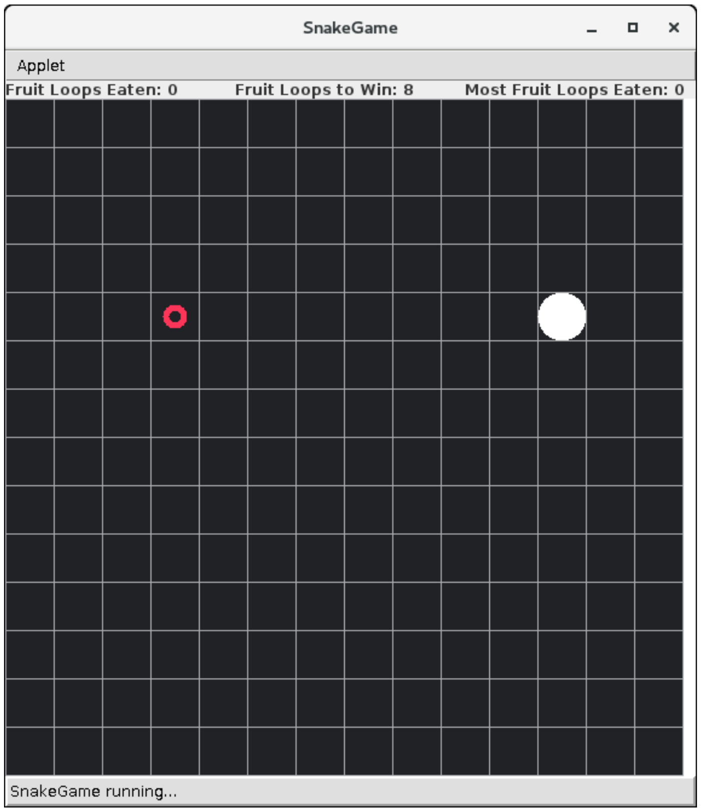
Back to Top



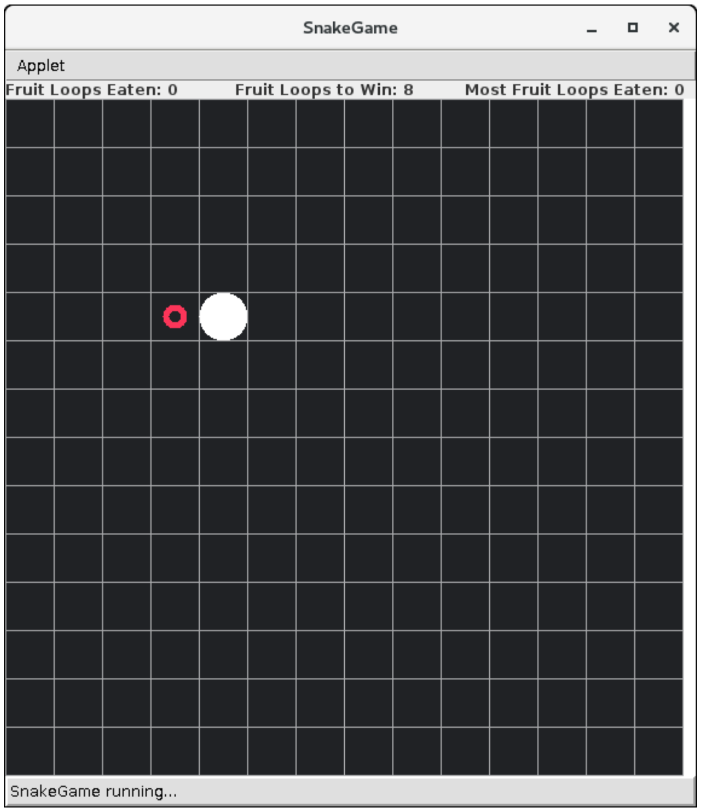
Snake continues to move right.



After pressing the up arrow key.

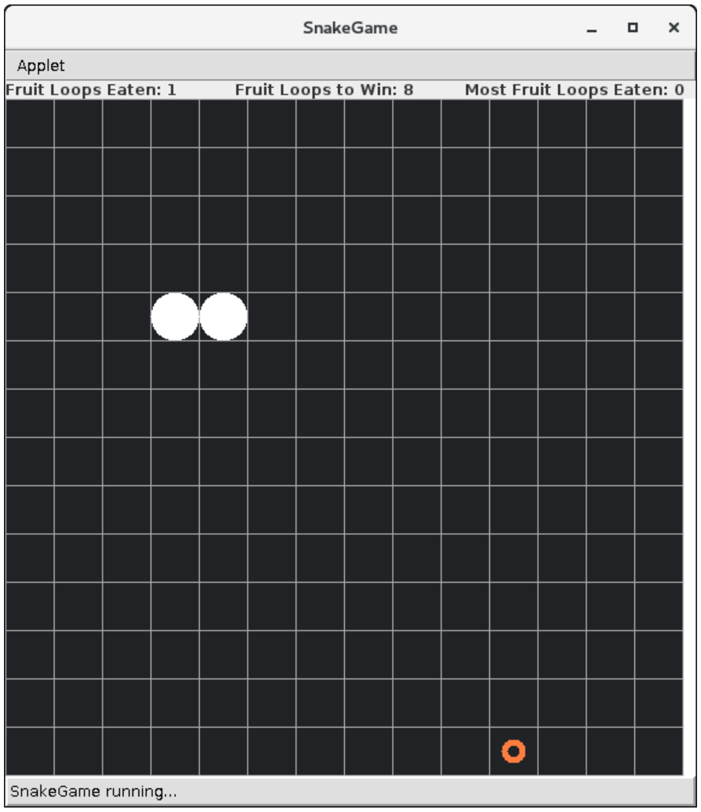


Snake continues to move up.



After pressing the left arrow key, snake is traveling left and about to eat the first fruit loop.

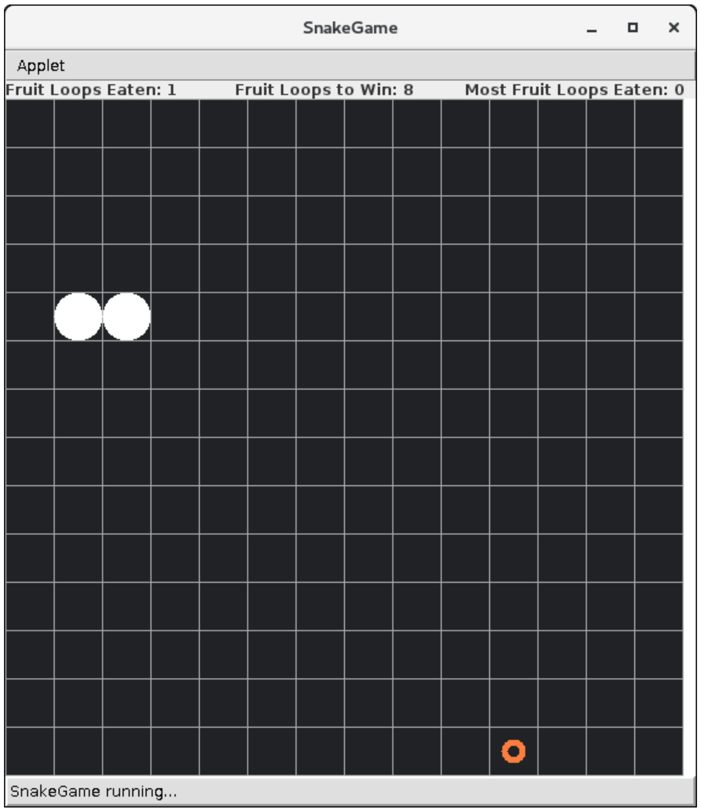
Back to Top



Fruit Loops Eaten: 1 Fruit Loops to Win: 8 Most Fruit Loops Eaten: 0

SnakeGame running...

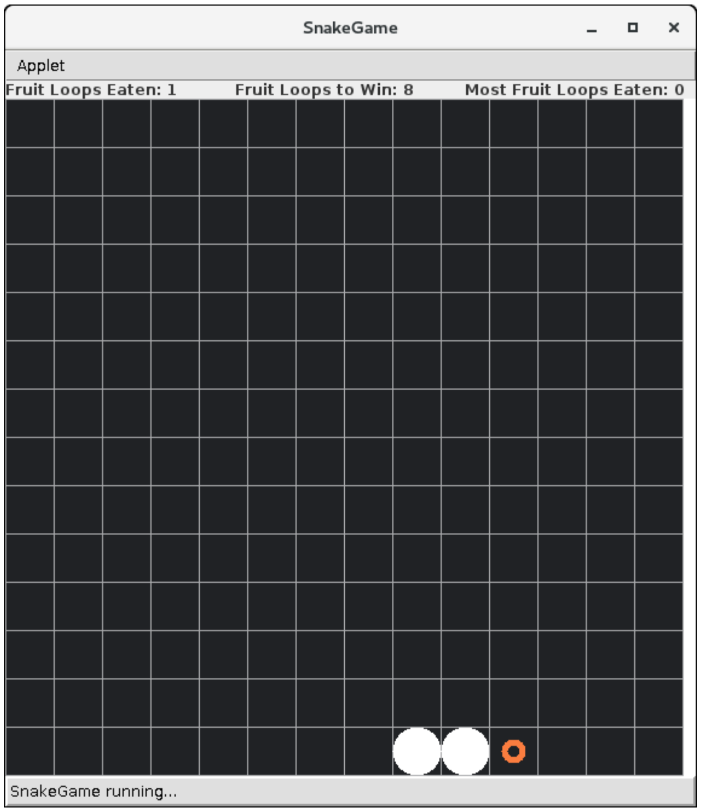
The snake eats the fruit loop and has grown a new segment. The “Fruit Loops Eaten” label is updated. A new fruit loop is placed.



Fruit Loops Eaten: 1 Fruit Loops to Win: 8 Most Fruit Loops Eaten: 0

SnakeGame running...

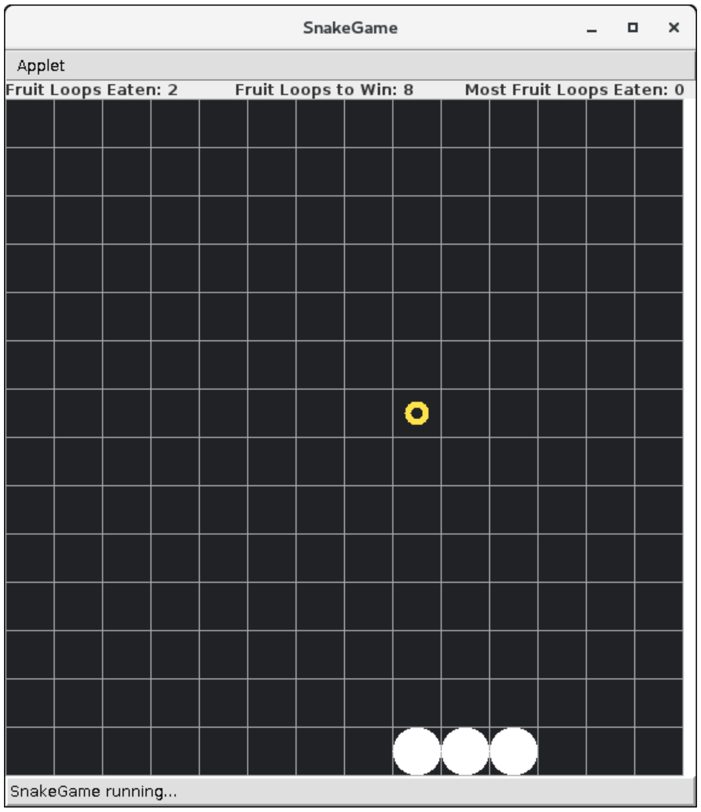
Snake keeps moving left.



Fruit Loops Eaten: 1 Fruit Loops to Win: 8 Most Fruit Loops Eaten: 0

SnakeGame running...

Snake is about to eat another fruit loop.

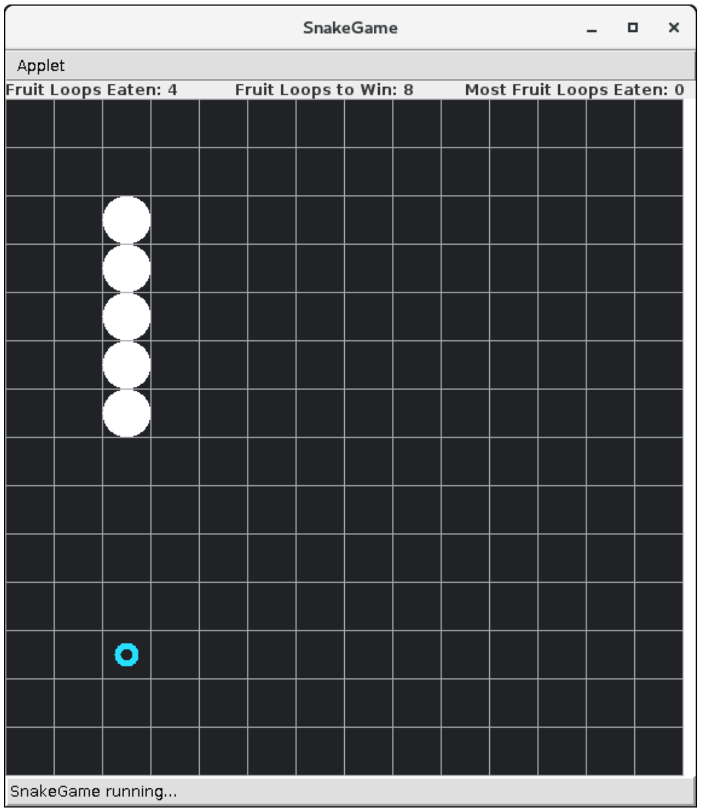


Fruit Loops Eaten: 2 Fruit Loops to Win: 8 Most Fruit Loops Eaten: 0

SnakeGame running...

The snake eats the fruit loop and has grown another new segment. The “Fruit Loops Eaten” label is updated. A new fruit loop is placed.

Back to Top

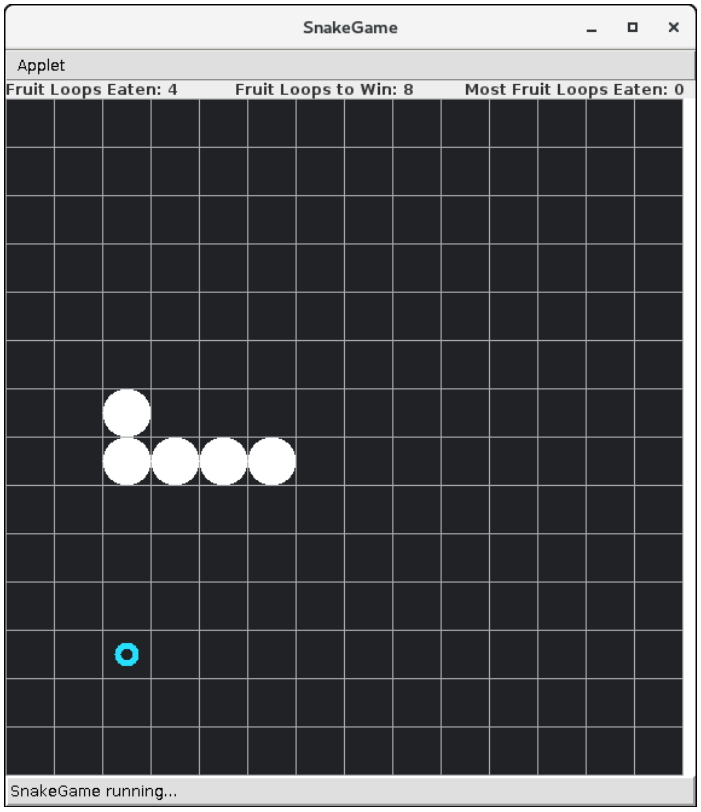

SnakeGame

Applet

Fruit Loops Eaten: 4 Fruit Loops to Win: 8 Most Fruit Loops Eaten: 0

SnakeGame running...

Snake after eating 2 more fruit loops.

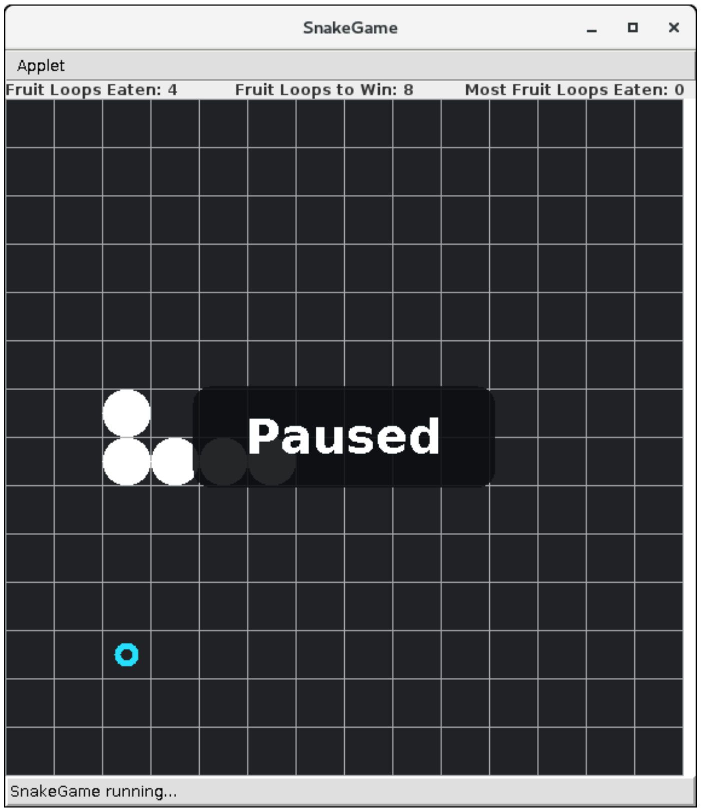

SnakeGame

Applet

Fruit Loops Eaten: 4 Fruit Loops to Win: 8 Most Fruit Loops Eaten: 0

SnakeGame running...

Note that when the snake turns, not all the SnakeSegments will be moving in the same direction; they are simply following each other.


SnakeGame

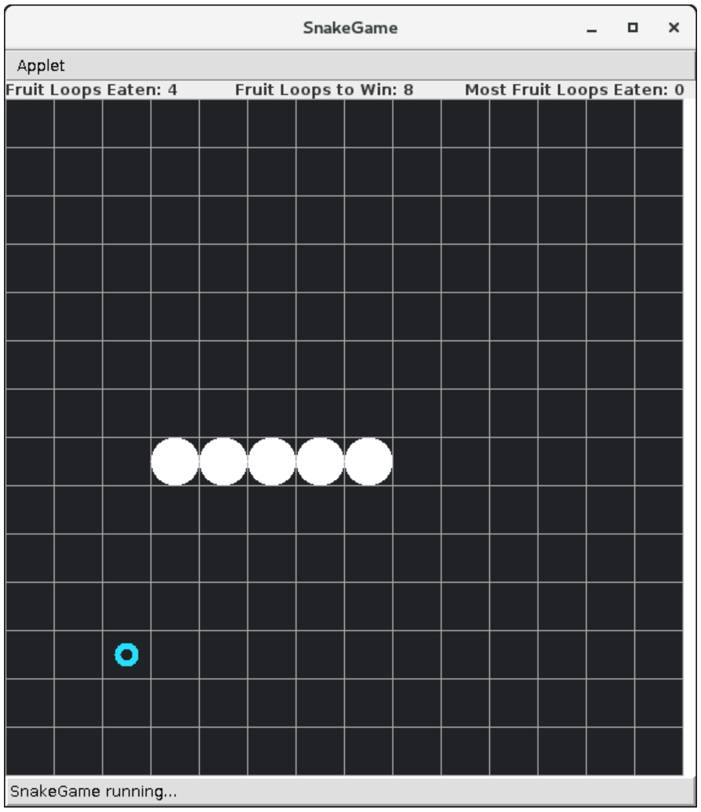
Applet

Fruit Loops Eaten: 4 Fruit Loops to Win: 8 Most Fruit Loops Eaten: 0

Paused

SnakeGame running...

After pressing spacebar to pause the snake.


SnakeGame

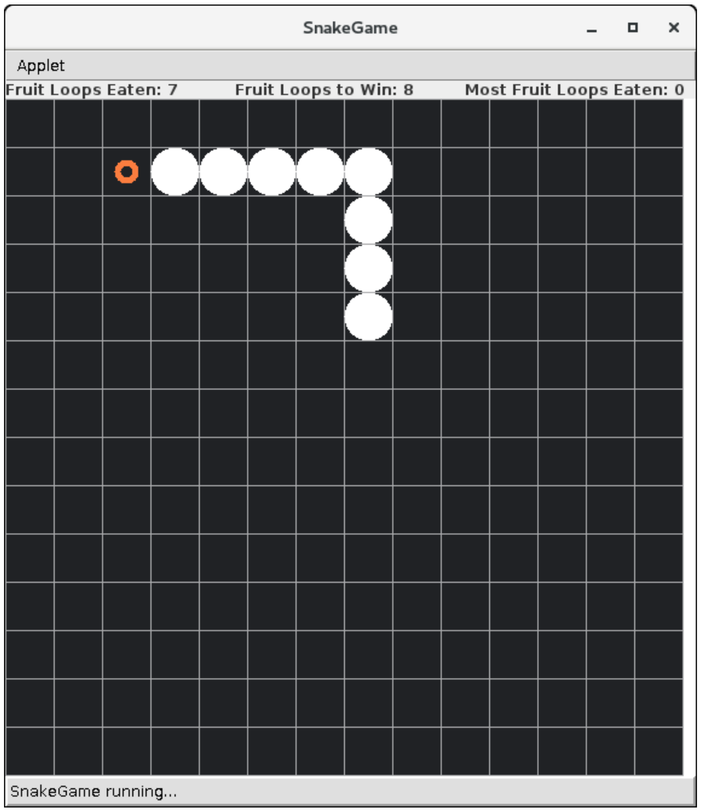
Applet

Fruit Loops Eaten: 4 Fruit Loops to Win: 8 Most Fruit Loops Eaten: 0

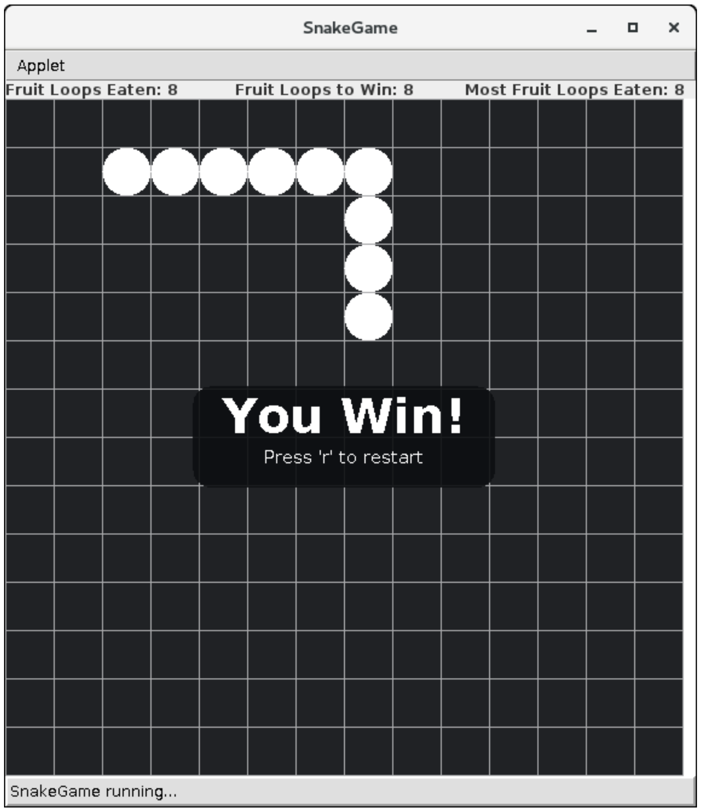
SnakeGame running...

Pressing arrow keys while paused does not affect the direction of the snake. After pressing spacebar again to unpause, the snake continues moving in the direction it was moving before pausing.

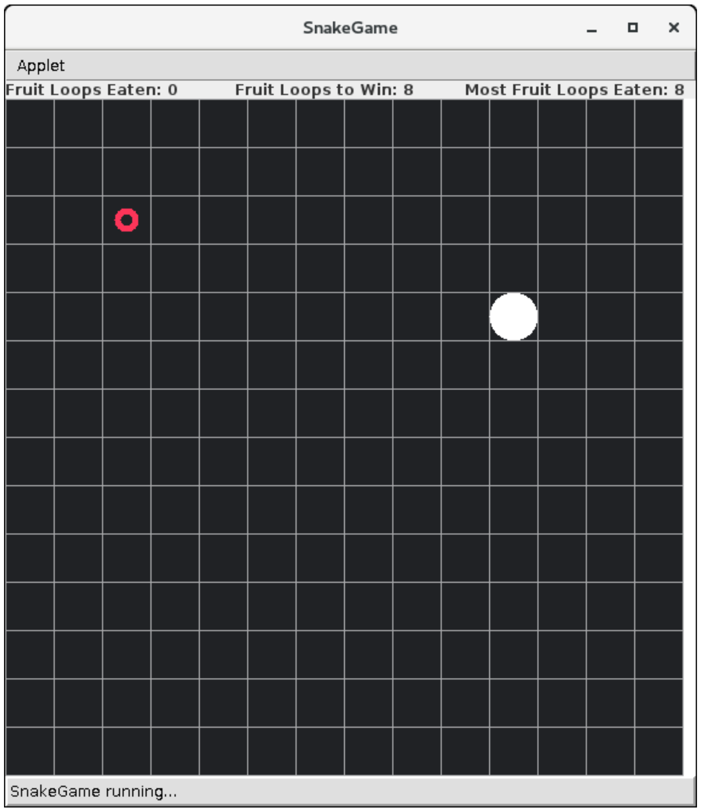
Back to Top



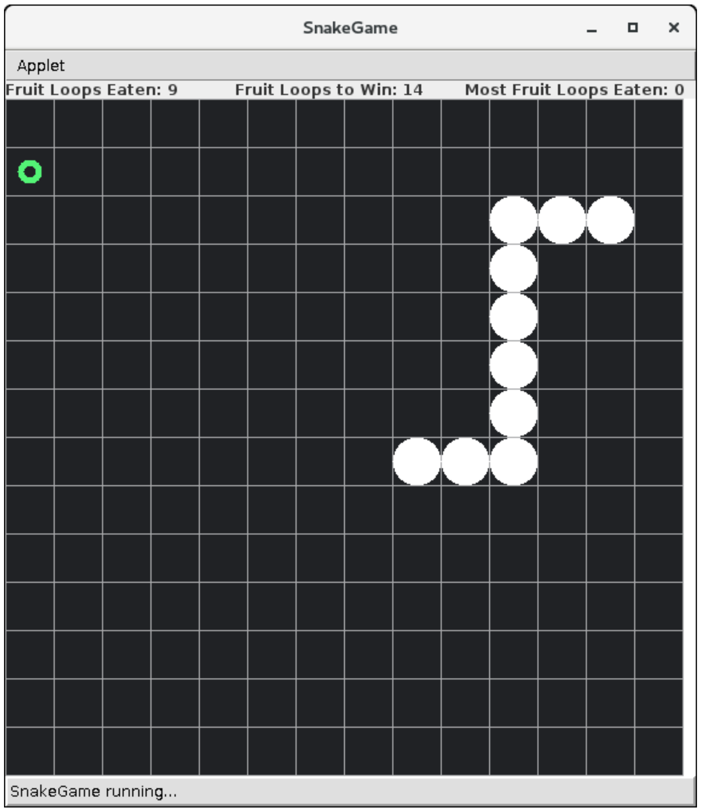
After eating 3 more fruit loops. Snake is about to eat the last fruit loop.



Snake eats the last fruit loop and grows a new segment. The win message shows up and the snake stops moving. The “Most Fruit Loops Eaten” label is updated.

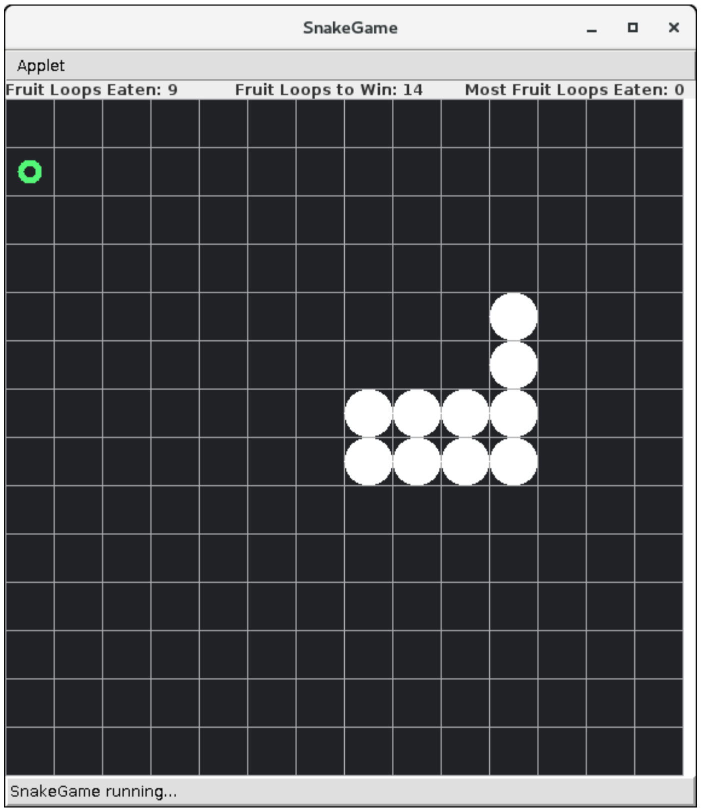


After pressing ‘r’ to restart the game. The snake is in a new random position and is stationary.



After restarting the program with a new number of loopsToWin. Snake after eating 9 fruit loops.

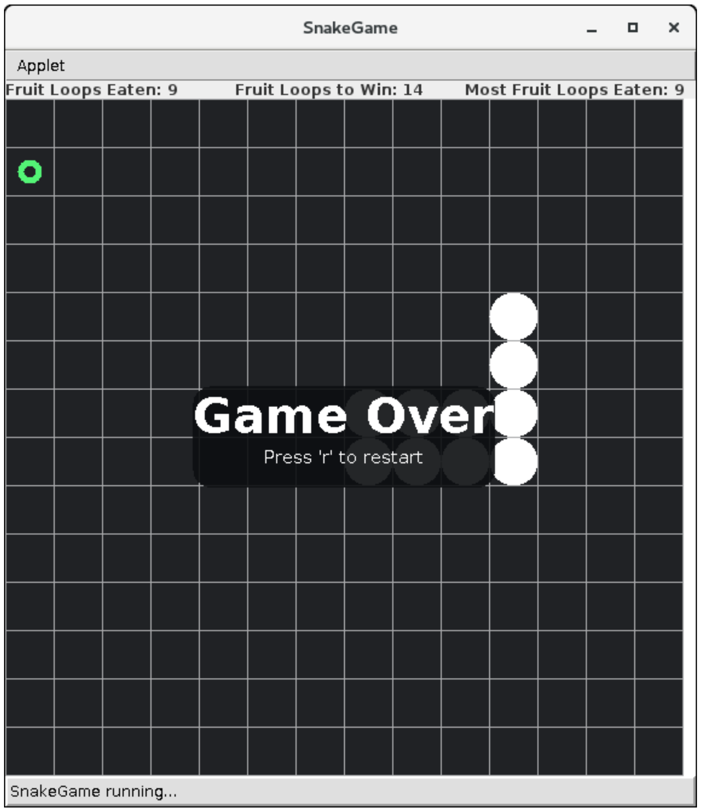
Back to Top



Fruit Loops Eaten: 9 Fruit Loops to Win: 14 Most Fruit Loops Eaten: 0

SnakeGame running...

Snake is moving right and is about to run into itself.



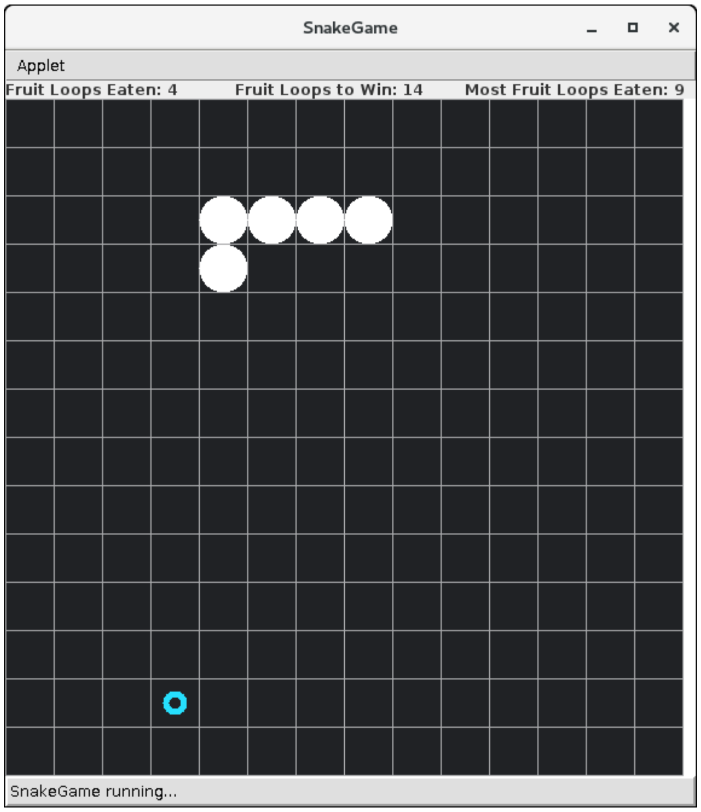
Fruit Loops Eaten: 9 Fruit Loops to Win: 14 Most Fruit Loops Eaten: 9

Game Over

Press 'r' to restart

SnakeGame running...

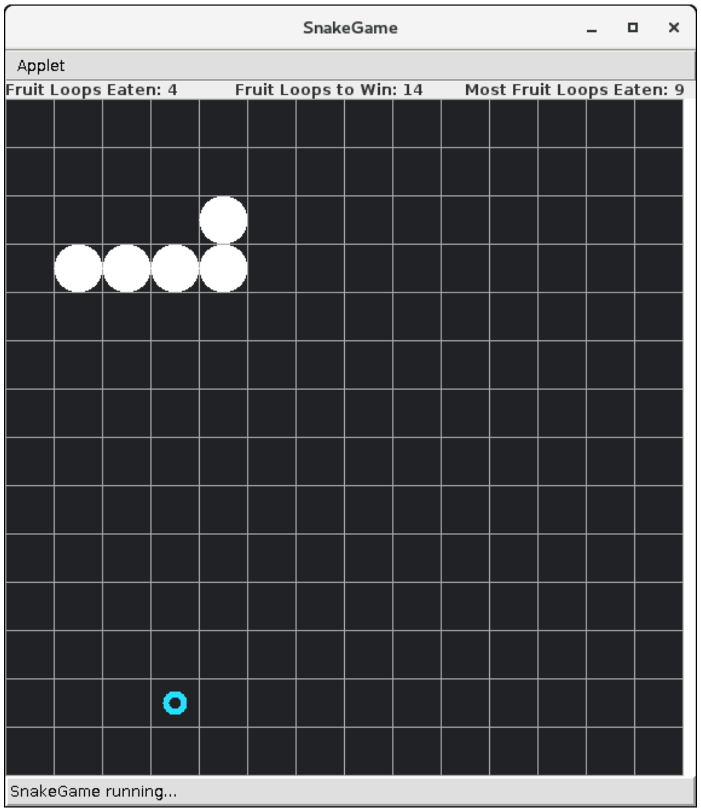
Snake ran into itself and the game is lost. The “Most Fruit Loops Eaten” label is updated.



Fruit Loops Eaten: 4 Fruit Loops to Win: 14 Most Fruit Loops Eaten: 9

SnakeGame running...

After pressing ‘r’ to restart and eating 4 fruit loops.

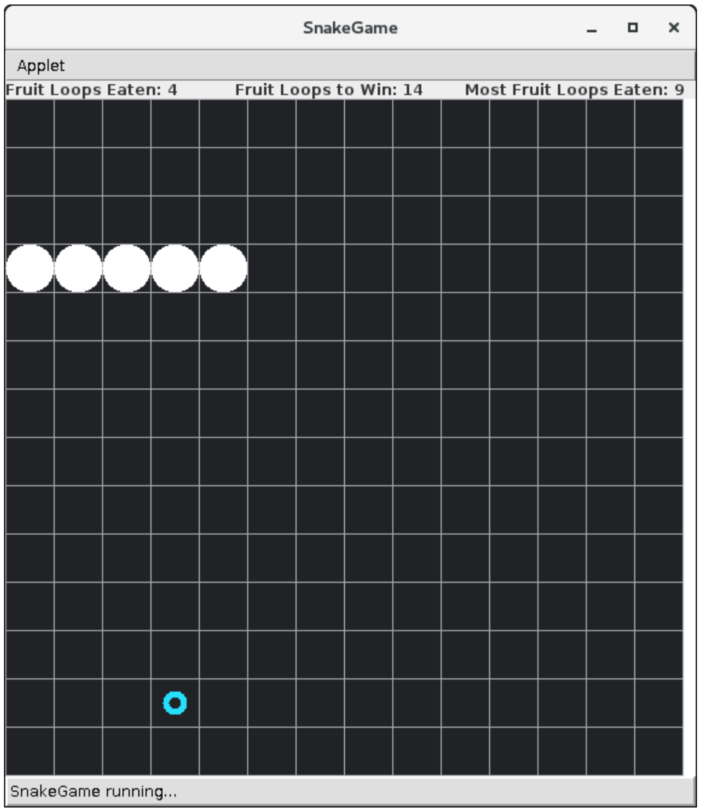


Fruit Loops Eaten: 4 Fruit Loops to Win: 14 Most Fruit Loops Eaten: 9

SnakeGame running...

Snake is heading left towards the edge of the grid.

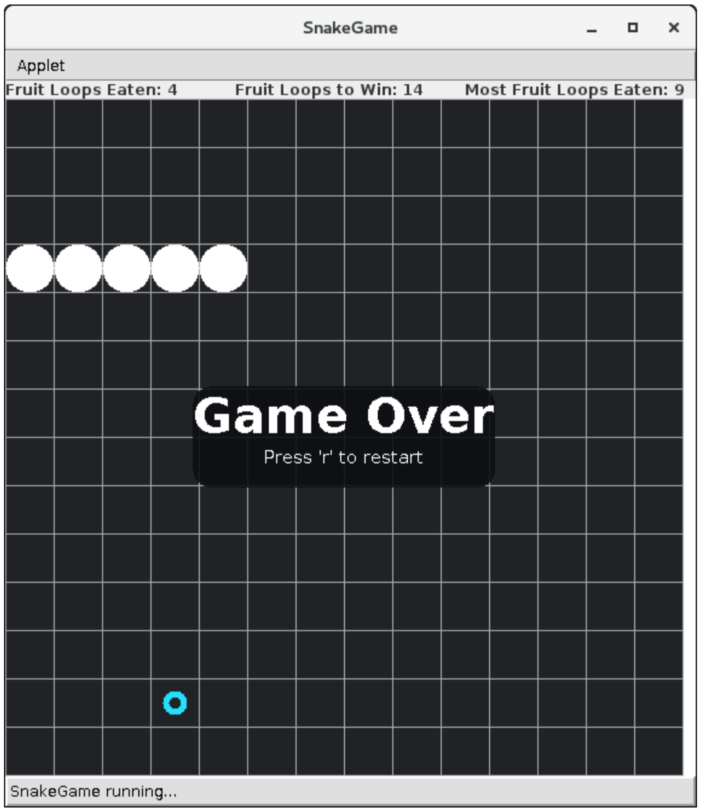
Back to Top



Fruit Loops Eaten: 4 Fruit Loops to Win: 14 Most Fruit Loops Eaten: 9

SnakeGame running...

Snake is about to run into the wall.



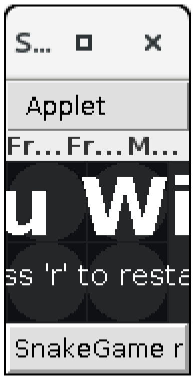
Fruit Loops Eaten: 4 Fruit Loops to Win: 14 Most Fruit Loops Eaten: 9

Game Over

Press 'r' to restart

SnakeGame running...

Snake ran into a wall and the game is lost.



Running the game with a very small grid can help with testing some edge cases. On a small grid, note that the popup messages will be larger than the window size, and the labels at the top will not be fully visible. This is fine and expected behavior.

README File

Remember to follow all of the guidelines outlined in the README Guidelines. If you did the extra credit, write a program description for it in the README file as well.

Questions to Answer in your README:

1. Why do the helper methods used for parsing command line arguments need to be static methods?
2. Describe the relationship between the GridCell, Grid, and FruitLoop classes.
3. How can you run vim through the command line to open all Java source code files in the current directory, each file in its own tab?
4. 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?
5. What are some of the consequences of not acting with academic integrity?

Back to Top

Extra Credit: Rainbow Bonus Mode

- **[5 Points]** After the game is won, allow the user to keep playing in rainbow bonus mode

Getting Started:

Make copies of the following files to do the extra credit in.

```
$ cd ~/pa8
$ cp FruitLoop.java EC_FruitLoop.java
$ cp Grid.java EC_Grid.java
$ cp GridCell.java EC_GridCell.java
$ cp Snake.java EC_Snake.java
$ cp SnakeGame.java EC_SnakeGame.java
$ cp SnakeSegment.java EC_SnakeSegment.java
```

Make sure you change all instances of FruitLoop, Grid, GridCell, Snake, SnakeGame, and SnakeSegment to their respective EC versions so your code can compile and run correctly.

Important: Your original files must remain unchanged. You need both the regular and the EC versions of these files for turnin.

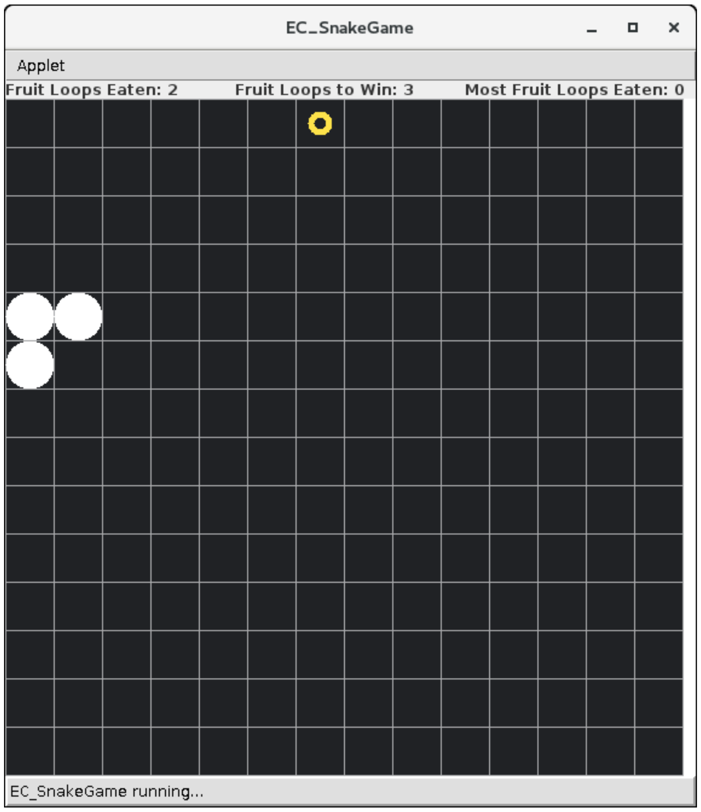
EC Requirements:

In the original version of SnakeGame, once the snake eats the number of loopsToWin, the game is won and the snake stops moving. In the EC version, rather than stopping the game and announcing that the game is won, let the user keep playing in rainbow bonus mode.

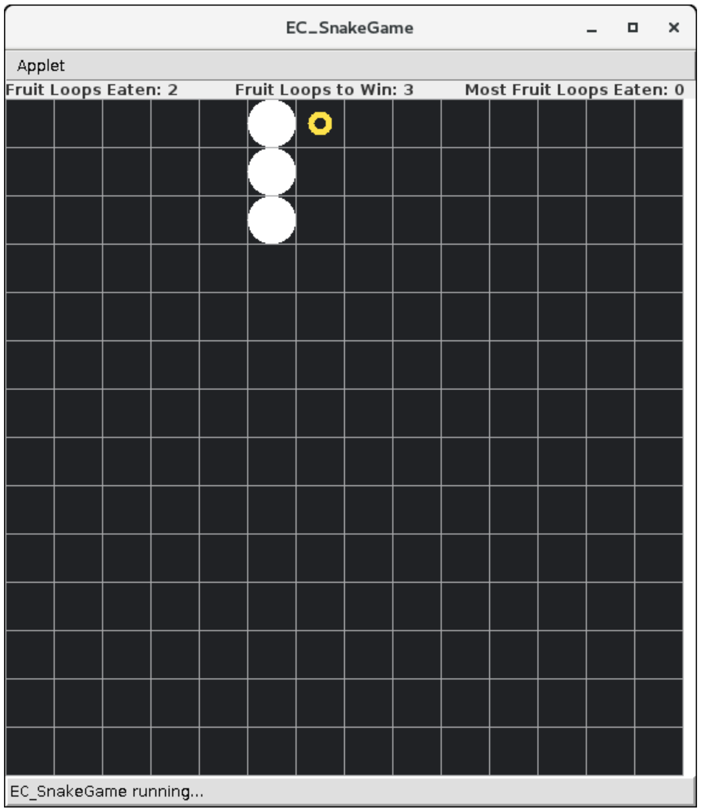
Rainbow Bonus Mode:

- Everytime the snake eats a fruit loop, set the color of the new snake segment to be the color of the fruit loop it just ate.
- After winning the game, let the snake keep eating fruit loops until the grid is full, the snake runs into itself, or the snake runs into a wall; then announce the win.

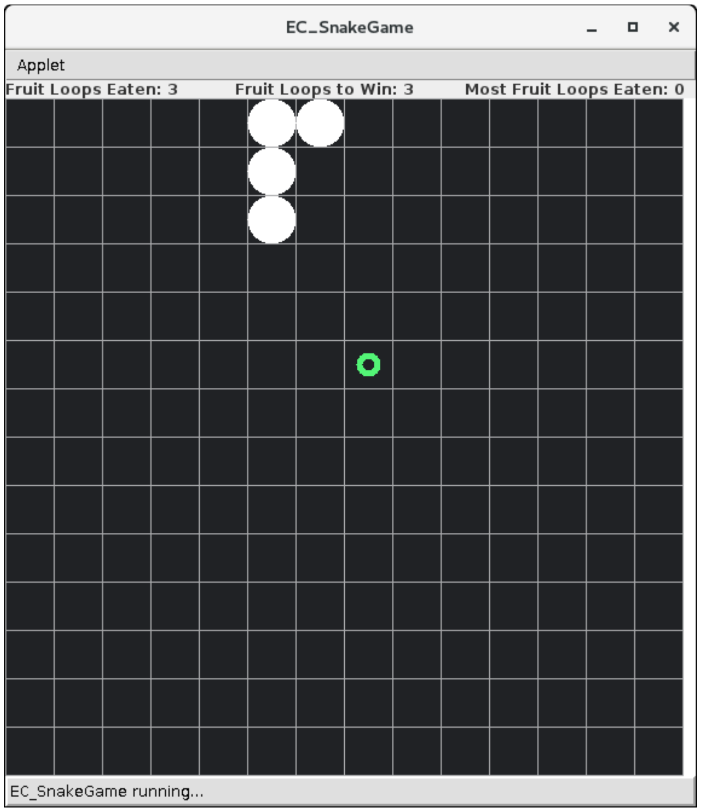
Sample Screenshots:



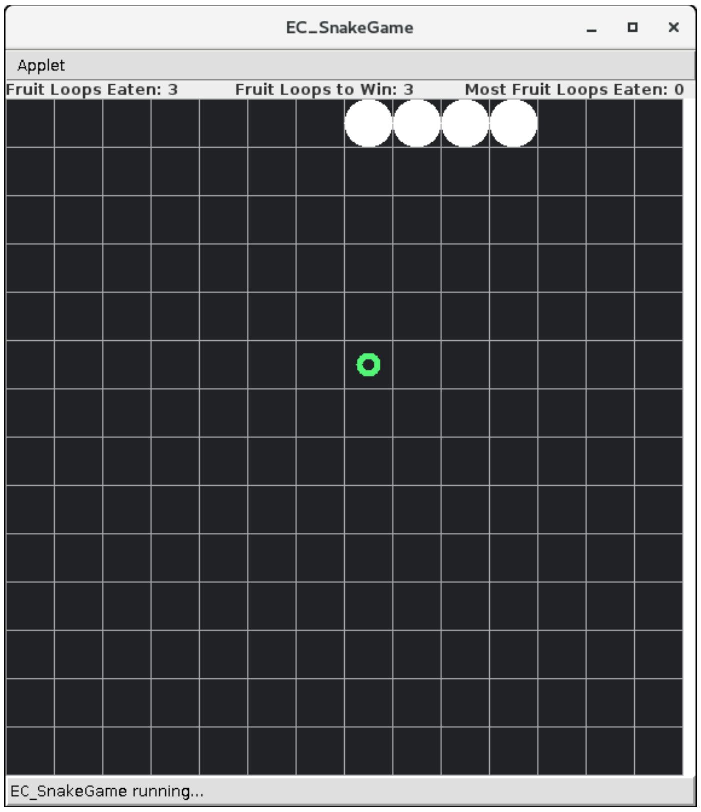
After eating 2 fruit loops. At this point, the behavior should be exactly the same as the regular version of the program.



Snake is about to eat the last fruit loop to win.

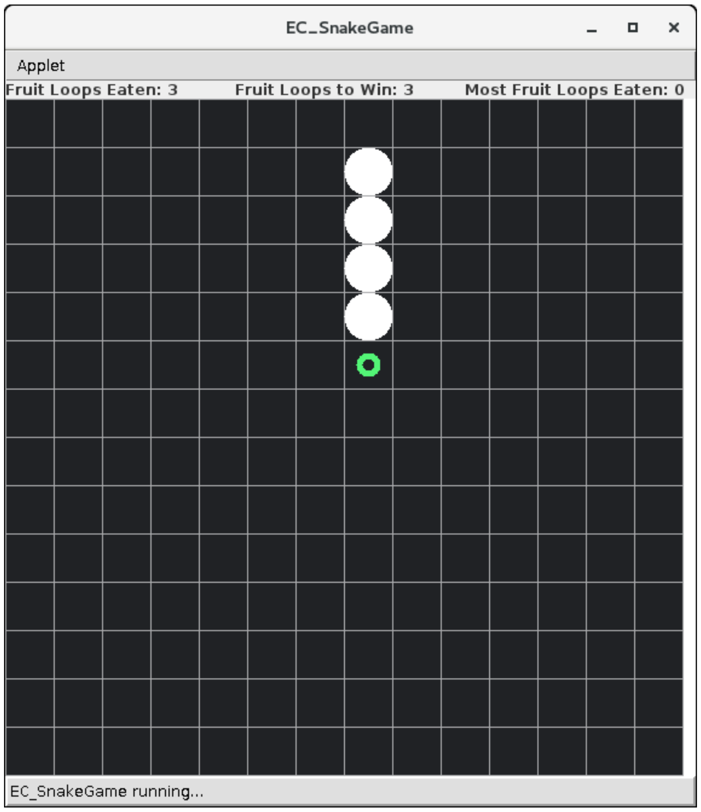


Snake eats the last fruit loop and grows a new segment as normal. The win message is not displayed. The “Most Fruit Loops Eaten” label is not updated. A new fruit loop appears.

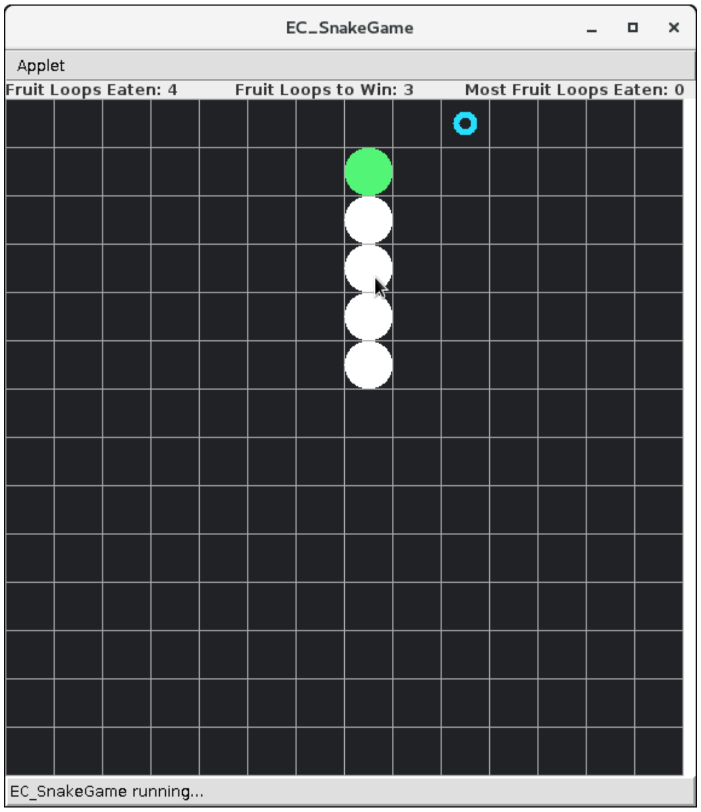


The snake keeps moving after eating the last fruit loop to win, and is now in rainbow bonus mode.

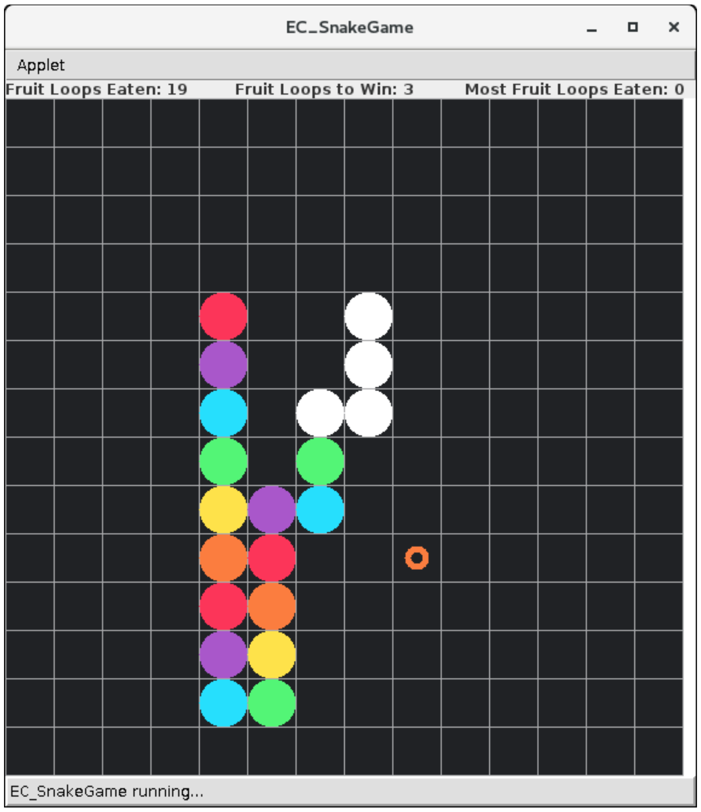
Back to Top



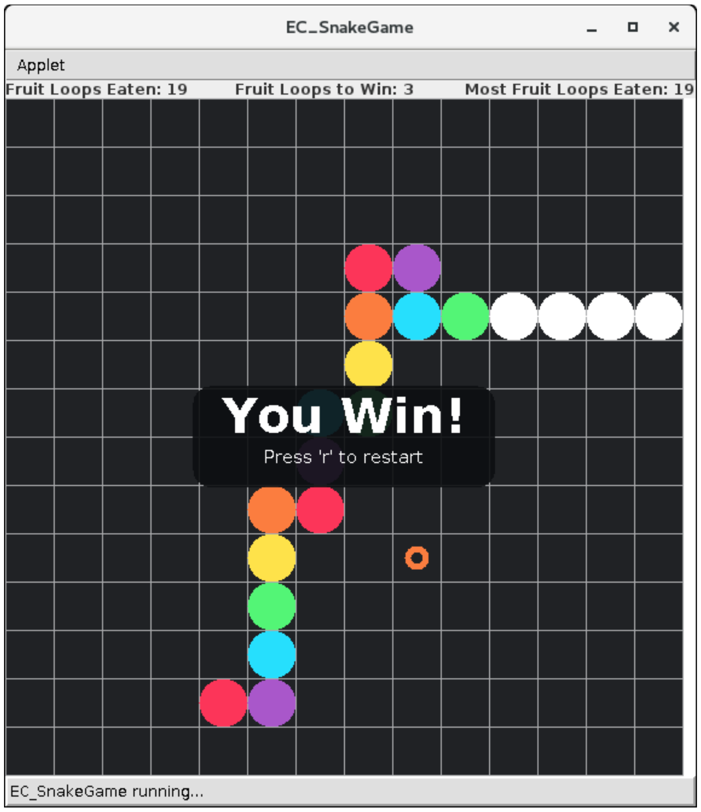
Snake is about to eat a fruit loop.



Snake eats the fruit loop and grows a new segment that is the color of the fruit loop it ate.



After eating a lot more fruit loops.



After “losing”, the win message is displayed and the “Most Fruit Loops Eaten” label is updated.

Back to Top

Turnin Summary

See the turnin instructions here. Your file names must match the below *exactly*.

Due Date: Wednesday night, November 21 @ 11:59 pm

Files Required for Turnin:

`Direction.java`	`PopupMessage.java`	`Acme.jar`
`FruitLoop.java`	`Snake.java`	`objectdraw.jar`
`Grid.java`	`SnakeGame.java`	`README`
`GridCell.java`	`SnakeSegment.java`	
`PA8Constants.java`		

Extra Credit Files:

`EC_FruitLoop.java`	`EC_GridCell.java`	`EC_SnakeGame.java`
`EC_Grid.java`	`EC_Snake.java`	`EC_SnakeSegment.java`

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.

NO EXCUSES!

NO EXTENSIONS!

NO EXCEPTIONS!

NO LATE ASSIGNMENTS ACCEPTED!

DO NOT EMAIL US YOUR ASSIGNMENT!

Start Early, Finish Early, and Have Fun!