SNAKE GAME

Andrey Skvortsov, Sara Stojanovic, Angelina Ith

Section 1: Setup & Timeline

Purple* = *Completed

- ***Setup***
  - ***Due***: Nov 25
  - ***Days***: 1
- ***Program Specifications***
  - ***Due***: Nov 26 - 28
  - ***Days***: 2
- ***Analysis***
  - ***Due***: Nov 26 - Dec 6
  - ***Days***: 2
- ***Design***
  - ***Due***: Dec 6 - 7 (Safe day: Wednesday)
  - ***Days***: 2
- ***Implementation***
  - ***Due***: Dec 8 - 10 (Safe day: All of week A)
  - ***Days***: 3
- ***Testing + Release***
  - Week of due date/possibly after Christmas

Notes:

- *This program is made for personal/recreational use, meaning program specifications are not taken from a consumer but rather created as a collaborative, informal group task*
- *Completion of the program and its steps are done as a combined effort, with no particular individual roles being assigned*
- *A link to the group Trello Board that helped in the completion of this project: https://trello.com/b/3Z0KU5I2/software-development-project-sara-andrey-and-angelina*

Section 2: Program Specifications

1. Purpose

To allow the user to control a "snake" around a grid, avoiding hitting the grid boundaries or itself while aiming to collect a set amount of "apples", or as many apples as possible before the snake grows too large for the grid. With every apple collected, the snake will become one unit longer. The game will increase in difficulty as it progresses because of the snake's growing size, making it more and more difficult to navigate across the grid as the grid space gets filled up with the body of the snake. The program will provide the user with a simple and enjoyable game to play.

2. List of Features Required to Achieve the Purpose (not finalized)

- Display the game on a GUI for a user-friendly experience
    - Idea: if using a 2D array, the array would have to somehow be reflected on a GUI grid. So far, it is unclear how this would be done. The array values (i.e. 0, 1) would also have to be converted to their corresponding sprite/display image (i.e. a spot with value 1 would have to have a snake part displayed)
- Take input from the directional keypad and convert it into the snake's change of directions
    - Idea: each direction change will involve converting between continuous x-axis and y-axis movement loops. EX. When the snake is moving right, it will be using an x-axis movement loop as it is moving along the x-axis. When the "up" key is pressed, the snake's movement will change to a y-axis loop that allows it to move up on the y-axis (note that x and y-axes will likely be stored on a 2D array)
    - NOTE: Using a regular scanner will not work because scanners pause the program while awaiting input. Our snake game must move continuously, not waiting for user input but rather reacting to it when it is given. We would have to use some sort of continuous scanner that only stops running when the snake dies (runs into an obstacle). This scanner would also have to run throughout the entire program, even during loops (at the end of each loop's increment??)
- APPLE EXTENSION: Spawn apples randomly throughout the grid for the user's snake to try to consume (gives the user a targetable goal during the game so they are not moving around the grid meaninglessly)
    - Idea: an RNG would likely have to be used in order to randomly generate the apple spawn coordinates. This RNG would make two calculations; one for the y-axis and one for the x-axis coordinates of the apple. These numbers would then be inputted into the array variable (i.e. grid[rngX][rngY]) and the program would change the element of the coordinate to a number that signifies an apple is there. In the case that a snake part is already in that coordinate, the RNG would simply redo its calculations to find a new coordinate until it picked an empty grid spot
    - Idea: to avoid picking a spot with a snake part in it, the coordinates of each snake spot could be implemented into the RNG's parameters to avoid it picking those numbers, however, it is unclear how this would be done or if it is possible
- APPLE EXTENSION: Increase the snake's length by one unit each time an apple is consumed

    - Idea: have a variable that contains the snake's length in array units (i.e. snakeLength = 5. Each time an apple is consumed: snakeLength++;
- Make the snake move continuously in its inputted direction
    - Idea (one-directional movement): axis one stays the same since we are moving in one direction. With axis two, increment array index and set the element of the coordinate to 1 (this means the snake is there). Then subtract the snakeLength variable from axis two's present index to get the index containing the end of the snake's body. Set the element of this coordinate to 0 (since the snake has left this coordinate, the coordinate is now empty)
    - NOTE: a slight delay (.5s hesitation) would have to be made at the end of each increment in order to avoid the computer incrementing so fast that the snake shoots out of bounds and the user is unable to react
- End the game when the snake bumps into itself or goes out of bounds (extension: or bumps into an obstacle)
    - Idea: if using a 2D array, empty grid spaces will have a value of 0. Grid spaces with the snake on them will have a value of 1. Grid spaces that are out of bounds or obstacles will have a value of 2. A conditional implemented into the continuous movement loops could check if the snake's head has entered a grid space with a value of 1 or 2, in which case the game would end
- APPLE EXTENSION: Score tracking
    - Idea: the number of consumed apples will be kept track of and displayed to the user in a "Score" box. IF the program is made in a way that it gives the user a targeted number of apples to consume, the program will end once "x" amount of apples have been consumed. This could be displayed like so: "Score: 12/15, 13/15… 15/15… You Win!"
- Game rules/instructions
    - Idea: the rules and instructions for the game will be displayed to the user at a convenient click of a button. They will be removed from display once the user clicks "X" or something similar

3. Expected User

- General population looking to play a fun game. People of all ages
- Little-no knowledge of computer science/programming is expected

4. Expected Input

- Only input required is directional keys which will change the direction of the snake's movement (see question ***2. Keys to Achieving the Purpose*** for how this will be done)
- GUI is needed to display grid and snake movement, as well as score and game instructions in order to allow the user to see their inputted snake movement
- Art will not be a big part of the GUI, general icons (sprites) for apples, obstacles and the snake are possible

5. Expected Data Types to be Used

- Likely only ***integers***, possibly ***doubles*** as well
    - If a 2D array is used for the grid, the indexes will be what determine what is on each grid spot (1 as an index means the snake is on that block, 0 as an index means the block is empty, 2 as an index means there is an obstacle)

    - Array coordinates are a big part of the data that the program will be interacting with
- Possibly ***booleans*** in the use of loops

6. Calculations/Processes Required for Program function

- Apple Extension: random numbers will be imported for the RNG to choose where apples spawn
- Apple Extension: Increase the snake's length whenever an apple is consumed (snakeLength++)
- Adding and subtracting from array coordinates to determine the snake head's next position (part of x/y axis movement loops)
- User score tracking done using an array

7. Clarifying Questions

- Will the program just use the arrow keys or WASD keys? Or any other keys?
- Realistically in terms of time, what extensions can we add? Apples? Modifiable difficulty?
- What will the sprites look like?
- What strategies can we use to move the snake?
- Will there be any interactive buttons on the GUI?
- Will the program keep track of previous scores (i.e. best) or reset each play?
- Apple Extension: Will there be a maximum number of apples collected or will the user be able to continuously collect apples until the snake hits itself or an obstacle/border?

Section 3: Analysis & Finalizing of Specifications

1. Answering Clarifying Questions

- Will the program just use the arrow keys or WASD keys? Or any other keys?
    - It should be quite easy to implement the use of WASD keys alongside the arrow keys. If using conditionals to check input, a conditional could be written as such: if (userInput = W || ^) {}; (use "or" operator)
- Realistically in terms of time, what extensions can we add? Apples? Modifiable difficulty?
    - Modifiable difficulty does not seem very hard at all, just time-consuming. It would likely just involve taking input from the user and using it as variables (if user enters 4x5 as grid size, use variables 4 = width and 5 = height where necessary). For obstacle modifications, could take input from user like so: if (userInput = 1) {5 obstacles in ***x*** arrangement on the grid}, else if (userInput = 2) {10 obstacles in ***x*** arrangement on the grid}, etc.
    - Apples would likely involve implementing an RNG (with parameters and having an "error-trap" loop to ensure the RNG doesn't pick a grid space that is already taken by an entity) into the code through a loop in the snake's movement loop that checks if the snake has hit an apple every time the snake moves forward. Considering that apples are half of the game's function, this extension will almost certainly be done
- What will the sprites look like?
    - For time efficiency, sprites can be simple black objects against a white background grid. Apples could be rhombuses, the snake body lengths could be black squares, and the snake head could be a triangle pointing in the direction of travel
    - If time allows, sprites can be given more creative thought (colors, funny & interesting pictures to make the game more attractive to audiences)
- What strategies can we use to move the snake?
    - Create a grid using a 2D array and map it out with element values (snake = 1, obstacle = 2, apple = 3, empty = 0), use movement loops that work with x and y axes to continuously move the snake in one direction (note: need to figure out how to turn the snake. Could be done by saving snake coordinates and having parts move to the next body part's previous coordinate?)
    - Use the GUI coordinate grid (grid stored in a panel on the GUI)
    - Create a grid out of appearing and disappearing buttons that are only visible when they are part of the snake
- Will there be any interactive buttons on the GUI?
    - Instructions will either pop up at the click of a button located just next to the grid, or will be present on the start screen
    - If user score is kept track of, this will not be interactive, simply a tracker
- Will the program keep track of previous scores (i.e. high score) or reset each play?
    - Making the program simply reset each play would likely be more time efficient, however, in the future, adding a "high score" score tracker would make the game more competitive and ultimately more attractive to consumers

- Apple Extension: Will there be a maximum number of apples collected or will the user be able to continuously collect apples until the snake hits itself or an obstacle/border?
    - Both of these options are fairly easy: continuous collection would just keep on spawning RNG apples every time one is eaten, while a maximum number could easily be implemented using a conditional (if (userScore = <10){call on endGame method})
    - In the beginning, priority should be given to continuous apple generation as this would pose less problems and be more time efficient. If time allows, a max. value could be added

2. Final Modifications to Feature list

- Display the game on a GUI for a user-friendly experience
    - The main JFrame GUI will contain a panel that is the playing space (grid)
    - Imports such as graphics, draw method, and image methods will be used to *draw* images on the panel
- Game rules/instructions
    - The rules and instructions for the game could be displayed to the user at a convenient click of a button as an interactive part of the GUI. They will be removed from display once the user clicks "X" or something similar.
    - Another, easier option is to simply display movement instructions on the start screen, then remove them for the rest of the game
- Take input from the directional keypad and convert it into the snake's change of directions
    - The main JFrame GUI will use an Action Listener, which will continuously check for user input and set true the direction of user input each time one of the WASD or arrow keys is pressed
    - The setting true (a.k.a. toggling) of the user's inputted direction will determine which direction the snake head moves/where snake will go
- Make the snake move in its inputted direction
    - A timer continuously runs from the beginning of the program's start. Every ***x*** milliseconds, the move method is called, moving the snake forward one unit
    - In the move method, each of the snake's body parts are moved to the coordinates of the body part ahead of them which moves the snake's body forward one unit
    - In the same method, the head is moved one unit in the inputted direction to complete a full movement and determine the direction of movement for the snake
- End the game when the snake bumps into itself, goes out of bounds or bumps into an obstacle
    - An if statement can be used to check if the snake's head (coordinates x[0] and y[0]) has gone out of the grid boundaries. Pseudo code: if (x[0] >= board width) {end game}, if (x[0] <= 0) {end game};
    - In order for the snake to hit itself, it has to have a length of 4 or greater. There will be a for loop that cycles through each of the snake's body part coordinates and checks if those coordinates equal to the coordinates of the snake head. If they are in fact equal, that means the snake hit itself and the game is lost
- Spawn apples randomly throughout the grid for the user's snake to consume

    - An RNG will be used to randomly generate the apple spawn coordinates. This RNG would make two calculations; one for the y-axis and one for the x-axis coordinates of the apple within the board's dimensions.
    - If time allows, apple coordinates could be checked in relation to snake coordinates (if apple is spawned on snake, coordinates are recalculated)
    - Each time an apple is consumed, a new apple will be generated in a random grid spot
- Increase the snake's length by one unit each time an apple is consumed
    - To determine if an apple is consumed, the coordinates of the head of the snake will be compared to the coordinates of the apple in a conditional. If the coordinates of the snake's head and the apple are the same, then the snake consumes the apple and 1 will be added to the length of the snake
- Score tracking
    - The number of consumed apples will be kept track of and displayed to the user in a "Score" box. If a collision is detected between the head of the snake and an apple, 1 will be added to the score counter. This could continue until the user reaches a max. number of apples or endlessly until the game stops
- Displaying images/sprites
    - Required sprites: apple, snake body part, snake head (head should be a separate color/shape from body to make visible the snake's orientation)
    - Technical note: images need to be saved in the same source file as the JFrame form, then saved in variables in a custom method. This requires using the Image class (see pseudo code, Section 4: Design, method *loadImages*)

Section 4: Design

1. IPO Model

a) Defining the Problem/Goal, Assumptions, Extensions

Purpose/Problem: To allow the user to control a snake around a grid while attempting to avoid hitting grid boundaries or the snake's body and aiming to collect as many apples as possible. The game should be played without issues of crashing, glitching or confusion.

Assumptions:

- The snake's length will start at 3 units
- The snake will increase in length by 1 unit each time an apple is collected
- The user will be able to control the snake's direction with keys
- If the snake hits the grid border or itself, the game will end
- Only one apple will be present in the game at all times (assuming this is for the default difficulty)
- The user will know what part of the snake is the head
- The game can be seen by the user while playing through the GUI

Extensions:

- Obstacles: Spawn set-coordinate OR randomly-generated obstacles around the grid to force the snake the avoid – would increase difficulty and could make the game more fun
- Modifiable Difficulty: Allow the user to choose how many obstacles there are/how big the grid is, modifying how difficult the game can be
- More Attractive Sprites: Immediate sprites will be very simple as the program will focus on functionality. If time allows, more interesting sprites could be used to focus the program more on attractiveness

b) Sample Input/Output

Input	***Processing*** ***(snake movement only)***	***Output***
"Up" or "W" key	If not moving down, set up loop to true, all other loops to false. Perform up loop	Snake turns up
"Down" or "S" key	If not moving up, set down loop to true, all other loops to false. Perform down loop	Snake turns down
"Right" or "D" key	If not moving left, set right loop to true, all other loops to false. Perform right loop	Snake turns right
"Left" or "A" key	If not moving right, set left loop to true, all other loops to false. Perform left loop	Snake turns left
Opposite direction key	EX.: Snake is moving right and user inputs to go left. Nothing will happen as a conditional checks to ensure the direction is not opposite. If it IS opposite, the	Snake continues moving in the same direction

	inputted loop simply will not run, and the snake will continue moving forward	
Random, invalid key	An action listener is what takes input, not a scanner. This means that if an unconsidered key is inputted, nothing will happen	Snake continues moving in the same direction

c) Required Variables

	Variable Name	***Variable Type***	***Info***
Constant variables	BOARD_Y	Final int	Height of the board. Set to 300p
	BOARD_X	Final int	Width of the board. Set to 380p
	DOTSIZE	Final int	Size of each unit in relation to board size. Set to 10p
	TOTAL DOTS	Final int	Total number of dots on the board. Is the area of the board div. by size of each dot
	DELAYSPEED	Final int	Is technically the speed of the snake. Really, this is the amount of time that passes (milliseconds) between each snake movement loop.
	snakeLength	Int	The length of the snake in dots.
	userScore	Int	The number of apples the user has eaten. Will be displayed in a label beside the grid
Coordinates	appleX	Int	The x-coordinate for the apple on the grid
	appleY	Int	The y-coordinate for the apple on the grid
	Array: x[]	Final int array	Array storing the x-coordinates of snake parts
	Array: y[]	Final int array	Array storing the y-coordinates of snake parts
Directions	moveUp	Boolean	If set to true, these run code that makes the snake go in that direction (i.e. if right is true and all others are false, snake will move right
	moveDown	Boolean	
	moveRight	Boolean	
	moveLeft	Boolean	
	apple	Image	

snakeHead	Image	These are the images of the apple, snake head and body. The images themselves must be stored as PNGs in the "src" file of the project
snakeBody	Image	
DRAW	Final Graphics	Used to communicate with board by reading board dimensions and making visual changes to the board (i.e. displaying pictures)
timer	Timer	Starts a timer that, when incremented by 140 milliseconds, runs the program's loops (i.e. moving the snake, checking for the apple, displaying images)
actionDelay	Boolean	Will be set to true after each timer loop, allowing user input to be taken only after the snake moves one unit.

d) Pseudo Code

JFrame Form Section One: Variables

- *Declare all global variables here*

JFrame Form Section Two: Custom Methods

- startGame
  - Imports required: javax.swing.Timer
  - Resets movement direction to default (right)
  - Sets snakeLength to 3 and userScore to 0
  - Sets score label to visible
  - Sets starting coordinates of snake using a for loop
  - Calls on spawnApple to spawn first apple
  - Calls on loadImages & displayImages to display game images
  - Starts timer
- endGame
  - Imports required: javax.swing.Timer
  - Stops timer, thus stopping the entire game and its loops
  - Sets "game over" message and "retry" button to visible
- loadImages
  - Imports required: java.awt.Image, javax.swing.ImageIcon
  - Finds necessary image files in their respective file location, extracts images
  - Saves images into their corresponding variable (apple variable for apple image, snakeHead, snakeBody)
- displayImages

- Imports required: java.awt.Graphics
- Uses variable initialized in method “initComponents” (variable DRAW) to draw images on panelBoard (grid)
- For apple image this takes one line of code, for snake this takes a for loop that loops through body part coordinates

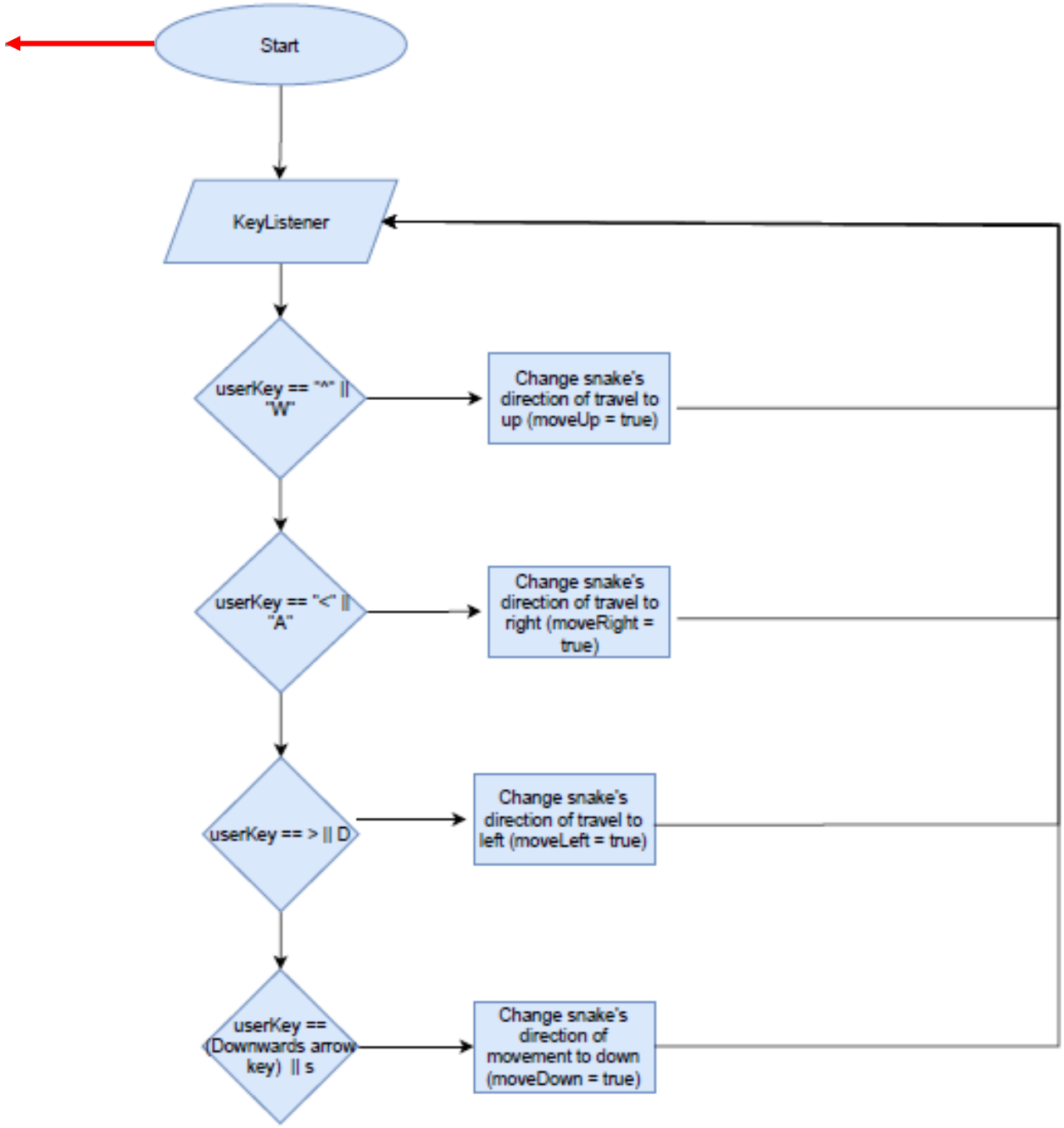
- move
  - Using a for loop, this method will loop through every x/y coordinate of the snake and move each part to the coordinates of the part ahead of it (snake moves forward into itself)
  - Conditionals will make the snake head move in its toggled direction
- checkCollision
  - Snake-on-snake collision: inside a conditional that checks to make sure snakeLength is greater than 4 (less than 4, snake cannot hit itself) a for loop will loop through every x/y coordinate of the snake’s body (as long as snakeLength > 4) and check if the head (x[0], y[0]) has hit them. If it has, endGame method is called
  - Snake-on-border collision: checks if the head is outside of the board boundaries (height = 380, width = 380) like so: if ((y[0]<0)||(y[0]=>height)){call on endGame}, repeat for x axis
- checkApple
  - Coordinates of the snake head (x[0], y[0]) will be checked in relation to apple coordinates
  - If coordinates match: snakeLength increased by 1 (along with userScore) and apple is relocated by calling on spawnApple
- spawnApple
  - RNG will pick for x/y coordinates (separately) of the next apple spawning
  - Will be called on in the checkApple method only if an apple has been eaten and must be replaced

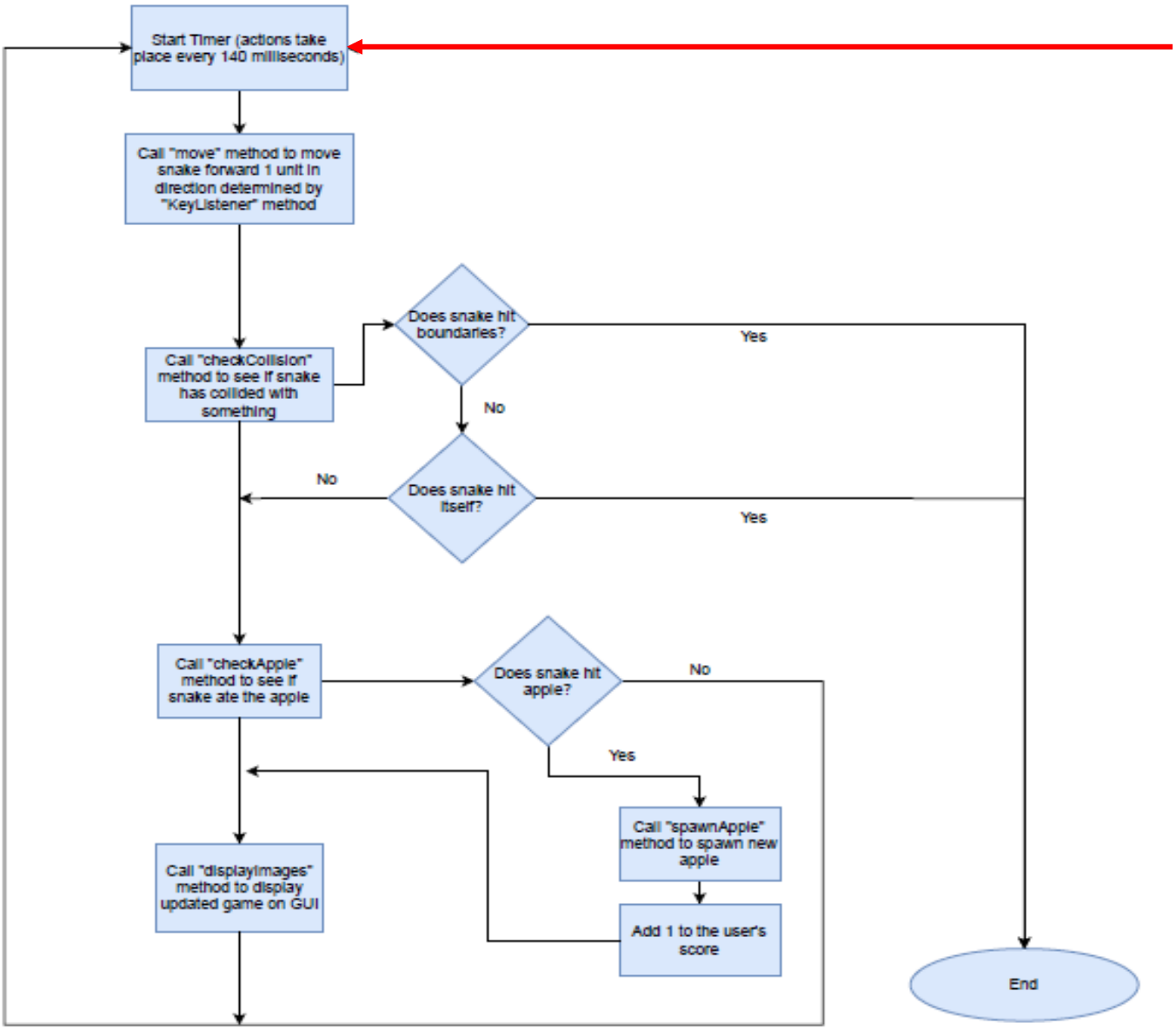
JFrame Form Section Three: Action Listeners

- Timer
  - Imports required: java.awt.event.ActionEvent, java.awt.event.ActionListener, javax.swing.Timer
  - Timer is started in startGame method (when user clicks play)
  - Listens for: increments of 140 milliseconds
  - Action: calls on move, displayImages, checkCollision and checkApple methods
- Main JFrame
  - Listens for: WASD or arrow keys
  - Action: sets true the inputted direction and sets false all other directions (converts user input to a snake direction)
- PlayRetry Button
  - Listens for: click of button

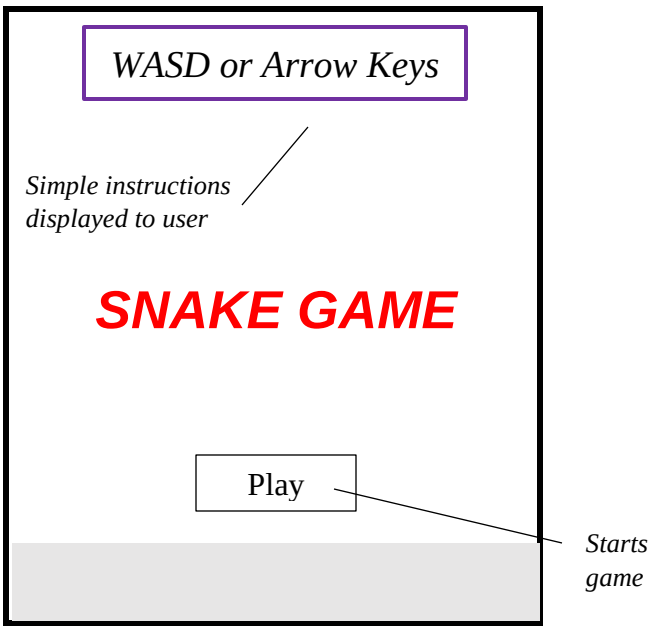
- Action: sets “Game Over” label & itself to invisible to clear the board and calls on startGame method to start the game

2. Flowchart Model

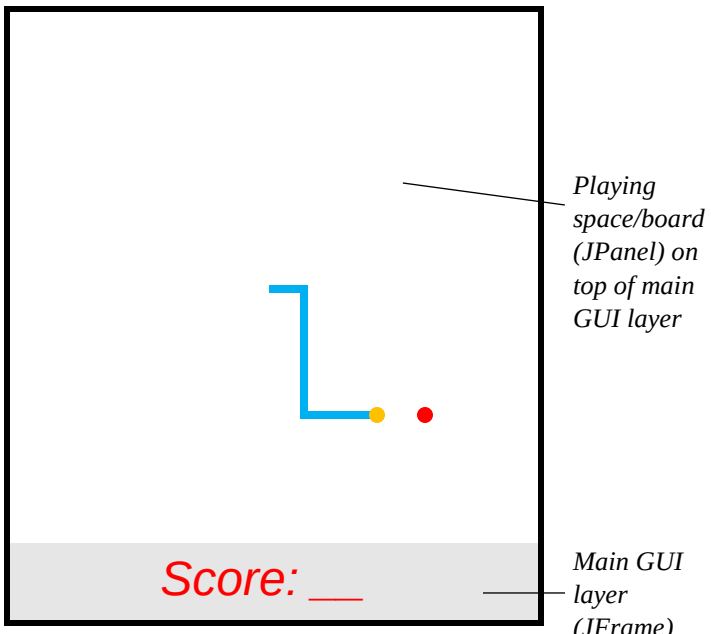



Start Timer (actions take place every 140 milliseconds)
Call "move" method to move snake forward 1 unit in direction determined by "KeyListener" method
Call "checkCollision" method to see if snake has collided with something
Does snake hit boundaries?
Yes
No
Does snake hit itself?
No
Yes
Call "checkApple" method to see if snake ate the apple
Does snake hit apple?
No
Yes
Call "spawnApple" method to spawn new apple
Add 1 to the user's score
Call "displayImages" method to display updated game on GUI
End

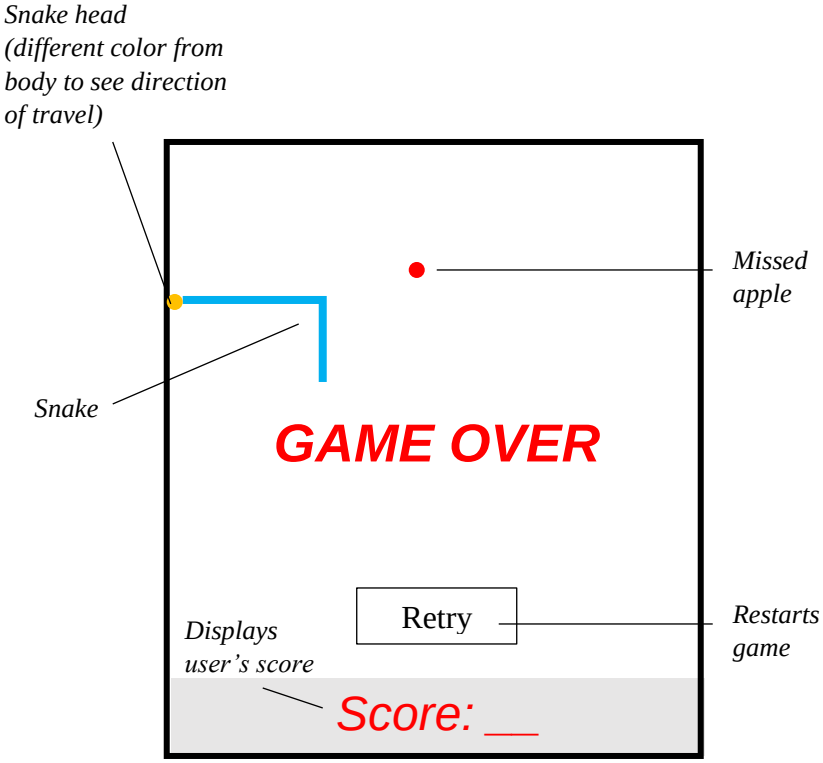
3. User Interface Prototype



Start Screen



In-Game Screen



End Screen

Here, this "Game Over" screen is prompted because the snake illegally hits the board boundary. The snake can also illegally hit itself, which would prompt the same screen

Section 5: Notes on Implementation

Error trapping is not needed – the only input taken is WASD or arrow keys and this is done through an action listener, any other input is simply ignored

Some of the Problems Encountered

a) Apple spawning in unreachable places (i.e. off the grid or not in 10-pixel increments thus not allowing the snake to hit the exact coordinates of the apple

Solution: Force the RNG to only be able to choose coordinates with increments of 10 pixels. This involves dividing coordinates by the size of each "dot" (DOTSIZE – 10), having the RNG choose, then multiplying by dot size once again to convert to coordinates

b) Snake reversing into itself and passing through itself, sometimes dying and sometimes not

Solution: adding a Boolean variable (actionDelay) that allows direction to change only when it is set to true by the timer after completing necessary loops, removing the ability to spam direction keys and have directions change faster than the snake moves

Section 6: Testing

1. Basic Testing Checklist

- ◊ ***Press play***: the GUI should appear without problems, GUI elements should change as required (i.e. "Play" button becomes "Retry" upon death)
- ◊ ***Crash the snake into all sides of the grid***: the game should end when boundaries are hit
- ◊ ***Crash the snake into itself***: the game should end when the snake hits itself
- ◊ ***Spam keys as fast as possible***: the snake should not move through itself/reverse its direction. The game should also not react from random key input, only WASD/Arrow key input
- ◊ ***Eat an apple***: the snake should grow by a length of one, the score should increase by one, and an apple should respawn
- ◊ ***Distribute an unfinished version of the game for pre-release testing***: users should play the game multiple times and try to reach as high of a score as possible without encountering any errors. Users should also understand the game's function without external explanation and without confusion

2. Testing Results

- ➢ All checklist items worked without fail or cause for concern
- ➢ External testers included 2 family members, both encountered no problems and were able to fully understand the purpose and function of the game
- ➢ Sample input/output model also was tested with perfect performance