Module Code: CS2JA16

Assignment Report Title: Major Coursework 1

Student Number: 26011251

Date (when work completed): 21/01/20

Abstract:

This report will show the use of the Java FX libraries to create a Drone Simulation with a Graphical User Interface (GUI). Inheritance is used to hide information during the creation of a Drone class and three sub-classes. The difficulty to create Drones with different behaviours is detailed. Tests were undertaken during the creation of the application to determine the success of particular design decisions. The varying success of these design decisions and their ultimate inclusion/exclusion were dependent on their complexity and the ease at which they could be programmed. Ultimately some features deemed necessary were excluded for these reasons. The conclusions to be drawn from the project and personal reflection are found at the end of the report.

Introduction:

Java FX libraries allow for the creation of graphical applications. The focus of the assignment was to create a graphical version of the previous Drone Arena assignment. Unlike the text-based arena, the graphical arena would contain multiple drone types which inherit from an abstract Drone class.

Design of the Application:

The application is comprised of six classes: Arena, Obstacle, Drone, BasicDrone, HungryDrone, and ZigZagDrone.

The Arena class is run as the Main. It contains:

- ArrayLists for each object
- Random Number Generator
- Canvas, Group, and Scene, Including Arena Rectangle
- Collision Counters and their Text
- Buttons and Text for the Graphical User Interface (GUI)
- Arena Obstacle creator
- Events to create the objects within the Arena
- Methods to detect collision between objects
- Main Method to run the application.

The Arena is populated by five different objects: two types of Obstacles and three types of Drones.

Obstacles:

The first Obstacle (the Lily Pads) are created by a for-loop as the application starts. The for-loop can be changed to alter the number of Lily Pads within the Arena. I chose a default of five Lily Pads as through testing I found more Lily Pads would increase the likelihood of Drones becoming stuck between Lily Pads and the Arena boundary as well as between other Lily Pads.

The for-loop uses the Obstacle class with the parameters of: xPosition, yPosition, radius, and color (colour). xPosition and yPosition are for the position of the Obstacle within the Arena, the radius and color are declared for use for the Circle object which is subsequently created using the parameters of the lily pad Obstacle. The Circle is the

graphical representation of the Obstacle within the Arena, each object is represented by its own Circle, which like the Lily Pad uses the objects' parameters for its own positions, radius, and color.

The Random Number Generator assigns each Lily Pad a random location within the blue Arena Rectangle using the bounds of the Rectangle as bounds for the generator. With more time I would have implemented a method to make sure each Lily Pad is assigned a location a certain distance away from other Lily Pads, this would make sure they would not overlap or be unevenly spread out within the arena. The method would calculate the distance between the newest Lily Pad and the others and generate a new location if the previous one was too close to another Lily Pad (below the shortest distance limit).

Each Lily Pad has the same radius and color. Changing these parameters automatically changes the parameters of the Circle object. The position of each Lily Pad within the Arena is fixed, and cannot be changed by the user. An improved version of the simulation would allow the user to change the number of Lily Pads and their parameters within the application, without having to alter the source code.

Upon creation, each Lily Pad Obstacle is added to the obstacleList ArrayList for use in collision detection between the Lily Pads and Drones. The checkObstacleCollision() method is used to check for collisions between the Lily Pads and the Drones moving about the Arena. It calculates the distance between each Drone and each Obstacle (using advanced for-loops) using the Pythagorean Theorem. If this distance is less than or equal to the sum of the radius of the Drone and the radius of the Lily Pad, the Drone will move in the opposite direction, 'rebounding' off of the Lily Pad. The number of collisions is then logged by the Collision Counter integer, increasing by one with each collision, and then displayed by the GUI. In theory this method would allow for consistent and accurate collision detection however in practice the Drones have a tendency to get stuck on the LilyPads, constantly changing to the opposite direction. As such the Collision Counter and Lily Pad Collision Counters in the bottom-right corner of the GUI are misleading and quickly increase upon the Drone becoming stuck. With a greater technical understanding of Java I would have tried to implement a more advanced collision detection technique however the current method works enough of the time for it to be successful in my estimate.

The second Obstacle is the Bread ("slice of bread"). As with the Lily Pads, the Bread uses the Obstacle class, with an xPosition and yPosition, along with the radius and color for use by its own Circle. Both the Lily Pad and Bread objects use the Obstacle Class as each are stationary objects and require the same variables, as such they did not need their own classes. The Bread Obstacle is created when the "Add Bread" Button is clicked during simulation, a maximum of five Bread objects can be added to the Arena.
When the maximum number of Bread objects is added the "Add Bread" Button Text will change to "Bread Bin Empty", preventing the user from adding anymore Bread objects. The Bread objects are added to the breadList ArrayList upon creation.

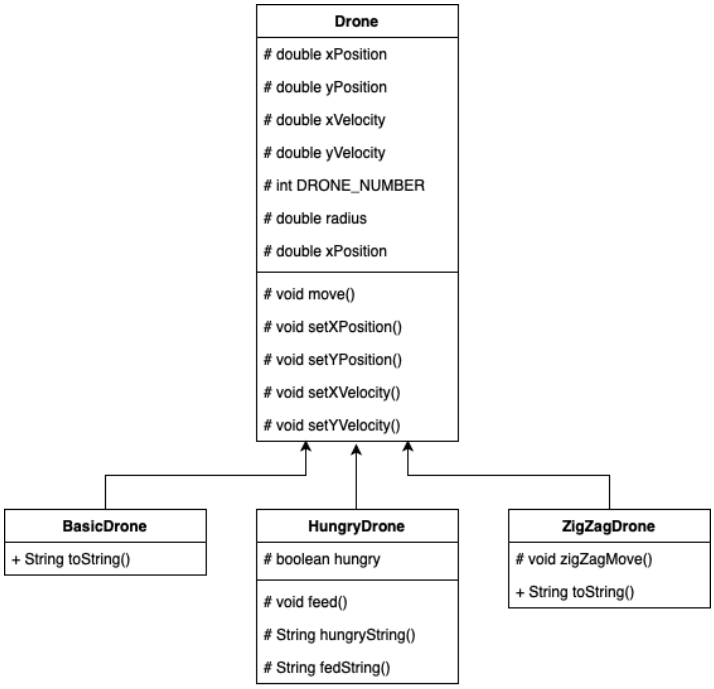
Unlike the Lily Pad, the Bread Obstacle only interacts with one type of Drone, the Hungry Drone. The checkBreadCollision() method works in the same way as the checkObstacleCollision() method, however instead of comparing all Drones to all Obstacles, the method compares all Hungry Drones from the hungryList to all Bread Obstacles in the breadList. If collision is detected, the Hungry Drone will call the feed()

method found within the HungryDrone Class. This increases the size of the Drone, decreases the velocities, and moves the Drone in the opposite direction.

The Lily Pad and Bread Obstacles essentially play the same role within the Arena. They are stationary objects used to change the parameters of the Drones moving within the Arena. Each does so using collision detection based upon distance. With greater understanding of Java I would have had the Bread objects removed from the Arena after collision with the Hungry Drones however I did not know how to remove specific objects from groups. Furthermore I should have added other types of detection as specified in the assignment brief. For example, moving the Drone around the Obstacle if it is detected to the left or right of the Drone, or implementing a cone of detection much like security cameras in Metal Gear Solid or similar stealth-based games. Currently the Drones cannot move around the Obstacles and their detection is solely through calculating the distance between objects.

Drones:

The three Drone types each have their own Class which inherit from the original Drone Class. As shown by the Class Diagram, the common Drone parameters and methods are declared in the Drone Class, with the parameters and methods particular to each Drone type declared in their respective Classes. Each of the Drone parameters and methods are protected as they are only accessed and called by the three SubClass objects within the Arena Class. Each SubClass of Drone is added to its own ArrayList upon creation as well as a general droneList ArrayList. These are for use in the GUI in which the number of Drones and the number of each type of Drone within the Arena is displayed. The number of each Drone is determined by the arrayList.size() method, called for each Drone ArrayList.



The Basic Drone does not have any unique parameters however it has a unique toString() method to display Drone information in the GUI. It inherits the move() method from Drone and uses this to move around the Arena. The Hungry Drone also uses the move() method from the Drone Class to move around the Arena. Using inheritance means the the method only has to be declared once in the Drone Class.

The Hungry Drone has the boolean parameter ‘hungry’. While this is true, if the Hungry Drone collides with a Bread Obstacle using the checkBreadCollision() method, the Hungry Drone will call the feed() method, and the radius, velocities, and direction of the Hungry Drone will change. The boolean parameter ‘hungry’ will then be declared false. Using the boolean parameter allows for two different versions of the same object, one in which the feed() method will be called, and another in which it won’t. As such a separate Drone

Class does not to be created for each version. To improve the simulation, more variations using boolean parameters could be used for the other objects in the Arena, increasing the variation of the Drones without having to create additional Classes. The Hungry Drone also has two String methods to display the Drone information for each boolean state, hungryString() while hungry == true, and fedString() while hungry == false.

The final Drone type is the Zig Zag Drone. As the name implies I originally planned for the Drone to alternate the direction of movement repeatedly so to move in a zig zagged line. The zigZagMove() method originally swapped the xVelocity and the yVelocity on a timer. This did not work as desired however as the Drone would instead change velocities too frequently, bugging out and exiting the bounds of the Arena. As such I had to change it so the Zig Zag Drone would instead be created with random x and y velocities so each Drone would move in a different direction with different speeds. Ultimately this leaves its behaviour very similar to that of the Basic Drone, highlighting what was my struggle to create Drones with unique behaviours.

Graphical User Interface (GUI):

The GUI is comprised of several buttons to add objects to the Arena as well as to stop and play the simulation. It also includes Text objects which display the information for each Drone using their respective toString() methods as well as Collision Counters for collisions with Obstacle and between Drones.

An aspect of the simulation required in the assignment brief was to incorporate the ability to pause and play the simulation via controls. I implemented this by creating a new Timeline when each Drone is created and adding each Timeline to an ArrayList. When either button is selected, the stop() or play() methods will be called in a loop for each Timeline, pausing or continuing the simulation respectively.

The bottom right corner of the GUI contains the counters for the number of collisions and the number of each Drone type within the Arena. I implemented these as to provide a purpose for the simulation other than to watch Drones move around a rectangle. With the counters the user can experiment to test whether the type of Drone within the Arena affects the number of collisions within the Arena. As previously detailed, the issue of Drones getting stuck on each other and the Obstacles leads to inaccurate and misleading counts, as such any results are left unreproducible as there is too little consistency. Furthermore, for the Drones to be compared fairly the Arena would have to be identical for each test, as such randomly located Obstacles would be inappropriate. This could easily be fixed by assigning the same locations for each test in the source code, or by allowing the user to do so in the GUI. Furthermore, the GUI lacks a timer which could have been implemented alongside the Stop and Play functions. I did attempt to incorporate a timer however I could not get it to work with said functions. The lack of a dedicated timer means any user wanting to conduct an experiment would have to use an external one, it would be rather difficult to synchronise the start of an experiment with the start of an external timer and would consequently not be worth pursuing. For any experiments to be conducted the inconsistencies with collision detection would have to be solved. Additionally the Arena would have to be set up in such a way so that tests could be reproduced, the random element would have to be removed for consistent repeats and variations. Lastly a timer function would have to be implemented, enabling the user to plot time vs collisions graphs as well as collision vs Drone type graphs, allowing for comparison of results.

User manual:

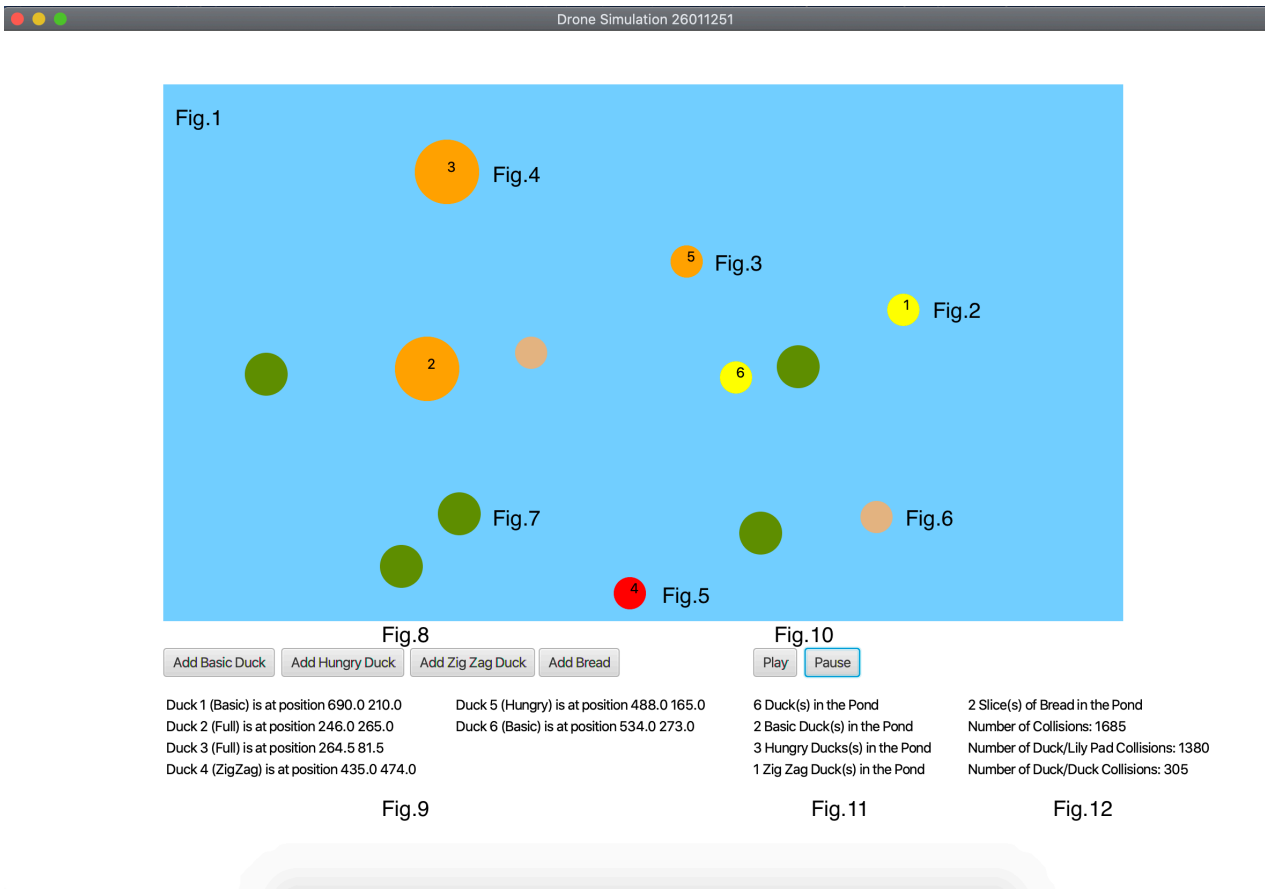


Figure.1 - The Arena Rectangle: The area in which Drones will move and operate. The Drones will rebound off the boundaries of the Arena.

Figure.2 - Basic Drone: The Basic Drone moves around at a set velocity. It will collide with the Lily Pad Obstacle and other Drones within the Arena. Upon collision it will rebound, moving in the opposite direction. Created by the Add Basic Duck button with a random starting position within the Arena.

Figure.3 - Hungry Drone (hungry==true): The Hungry Drone type with the ‘hungry’ boolean true. Will move around the Arena with a set velocity. It will collide with other Drones and both the Lily Pad and Bread Obstacles. Upon collision it will rebound, moving in the opposite direction. Upon collision with the Bread Obstacle the radius will double, the speed will decrease, and the boolean ‘hungry’ will be set false. Created by the Add Hungry Duck button with a random starting position within the Arena.

Figure.4 - Hungry Drone (hungry==false): The Hungry Drone type with the ‘hungry’ boolean false. After collision with the Bread Obstacle the Hungry Drone (hungry==true) will change to the Hungry Drone (boolean==false). It has double the radius of the Hungry Drone (hungry==true) and a decreased velocity. It will no longer collide with Bread Obstacles. Will move around the Arena will a set velocity. It will collide with other Drones and the Lily Pad Obstacle. Upon collision it will rebound, moving in the opposite direction.

Figure.5 - Zig Zag Drone: The Zig Zag Drone type. Moves around the Arena with a randomised velocity, generated upon creation. It will collide will the Lily Pad Obstacle and

other Drones within the Arena. Upon collision it will rebound, moving in the opposite direction. Created by the Add Zig Zag Drone button with a random starting position within the Arena.

Figure.6 - Bread Obstacle: Stationary Bread Obstacle with a randomly generated location within the Arena. Default maximum of five within the Arena. Interacts with the Hungry Drone (boolean==true). Created by the Add Bread button.

Figure.7 - Lily Pad Obstacle: Stationary Lily Pad Obstacle with a randomly generated location within the Arena. Interacts with all Drone types. Default of five created upon running of application.

Figure.8 - Add Drone / Add Bread buttons: Will create one of the chosen Drone type in a random location within the Arena. Will create one Bread Obstacle in a random location within the Arena. The default maximum number of Drones is eight. Once eights Drones have been created the text will change to “Pond Full”, no more Drones can then be created. The default maximum number of Bread Obstacles is five. Once five Bread Obstacles have been created the text will change to “Bread Bin Empty”, no more Bread Obstacles can then be created.

Figure.9 - Drone Information: Information string listing: the Drone number, Drone type, xPosition, and yPosition.

Figure.10 - Play and Pause buttons: Will play and pause the simulation

Figure.11 - Drone Count and Bread Obstacle Count: Lists the number of Drones currently within the Arena, the number of each individual Drone type currently within the Arena, and the number of Bread Obstacles currently within the Arena.

Figure.12 - Collision Counters: Displays the overall number of collisions within the Arena, the number of Drone/Obstacle collisions, and the number of Drone/Drone collisions.

Tests Conducted:

Test No.	Question	Test	Desired Outcome	Actual Outcome
1	Does the Add Basic Duck button add one moving Basic Drone to the Arena?	Select Add Basic Duck button.	One Basic Drone is added to Arena and moves.	One Basic Drone is added to Arena and moves.
2	Does the Add Hungry Duck button add one moving Hungry Drone to the Arena?	Select Add Hungry Duck button.	One Hungry Drone is added to the Arena and moves.	One Hungry Drone is added to the Arena and moves.
3	Does the Add Zig Zag Duck button add one moving Zig Zag Drone to the Arena?	Select Add Zig Zag Duck button.	One Zig Zag Drone is added to the Arena and moves.	One Zig Zag Drone is added to the Arena and moves.
4	Does the Add Bread button add one stationary Bread Obstacle to the Arena?	Select Add Bread button.	One stationary Bread Obstacle is added the the Arena.	One stationary Bread Obstacle is added the the Arena.

5	Will the Drones collide with other Drones and rebound?	Add two Drones to Arena. Wait to collide. Repeat for each combination: Basic + Basic, Basic + Hungry, Basic + ZigZag, Hungry + Hungry, Hungry + Zig Zag, Zig Zag + Zig Zag	Each Drone will collide and rebound with every Drone type without getting stuck.	Basic + Basic, Basic + Hungry, Hungry + Hungry consistently collide, sometimes get stuck on each other. Zig Zag Drones do not consistently collide with any other Drone.
6	Will the Drones collide with the Lily Pad Obstacle and rebound?	Create each Drone and wait for collision with Lily Pad Obstacle.	All Drone types will collide and rebound without getting stuck on the Lily Pad.	Basic and Hungry Drones consistently collide with the Lily Pad Obstacles, sometimes get stuck on the Lily Pad. Zig Zig Drones do not collide with Lily Pad Obstacle at all.
7	Will Hungry Drone collide with Bread Obstacle and call feed() method?	Create Hungry Drone. Create Bread Obstacle. Wait for Hungry Drone to collide with Bread Obstacle.	Hungry Drone will collide with Bread Obstacle and call feed() method. The radius of the Hungry Drone will increase, the velocity will decrease, and the Hungry Drone will move in the opposite direction.	Hungry Drone will collide with Bread Obstacle and call feed() method. The radius of the Hungry Drone will increase, the velocity will decrease, and the Hungry Drone will move in the opposite direction.
8	Will the “Full” Hungry Drone collide with Bread Obstacle and call feed() method?	Create Hungry Drone. Create Bread Obstacle. Wait for Hungry Drone to collide with Bread Obstacle. Wait for Hungry Drone to collide with Bread Obstacle again.	Hungry Drone will collide with Bread Obstacle and call feed() method. The radius of the Hungry Drone will increase, the velocity will decrease, and the Hungry Drone will move in the opposite direction. Upon the second collision it will not collide or call the feed() method.	Hungry Drone will collide with Bread Obstacle and call feed() method. The radius of the Hungry Drone will increase, the velocity will decrease, and the Hungry Drone will move in the opposite direction. Upon the second collision it will not collide or call the feed() method.
9	Will the Drone Information strings update the location of each Drone?	Create each Drone. Check the positions update and are consistent with the Drone locations.	The positions for each Drone update and are consistent with the Drone locations.	The positions for each Drone update and are consistent with the Drone locations.

10	Will the Pause button pause the simulation?	Create each Drone. Select the Pause button. Check each Drone no longer moves and the Drone Information strings no longer update.	Each Drone no longer moves. The Drone Information strings no longer update.	Each Drone no longer moves. The Drone Information strings no longer update.
11	Will the Play button continue the simulation?	Create each Drone. Select the Pause button. Check each Drone no longer moves and the Drone Information strings no longer update. Select the Play button. Check each Drone continues to move and the Drone Information strings continue to update.	The Drones continue to move. The Drone Information strings continue to update.	The Drones continue to move. The Drone Information strings continue to update.
12	Will the Drone Counters increment for a maximum of eight Drones in the Arena?	Add eight of one Drone type to the Arena. Check the Add Drone buttons change to “Pond Full”. Check the Drone Counters display eight Drones overall in the Arena, and eight of the selected Drone. Repeat for each other Drone type	Each Drone can be added for a total of eight Drones in the Arena. The Add Drone buttons change to “Pond Full”. Each Drone counter displays the correct number of Drones within the Arena.	Each Drone can be added for a total of eight Drones in the Arena. The Add Drone buttons change to “Pond Full”. Each Drone counter displays the correct number of Drones within the Arena.
13	Will the Bread Counter increment for a maximum of five Bread Obstacles in the Arena?	Select the Add Bread button five times. Check the Add Bread button changes to “Bread Bin Empty”. Check the Bread Counter displays five Bread Obstacles in the Arena.	A maximum of five Bread Obstacles can be added to the Arena. The Add Bread button changes to “Bread Bin Empty”. The Bread Counter displays the correct number of Bread Obstacles within the Arena.	A maximum of five Bread Obstacles can be added to the Arena. The Add Bread button changes to “Bread Bin Empty”. The Bread Counter displays the correct number of Bread Obstacles within the Arena.
14	Will the Collision Counters increment appropriately for every collision within the Arena?	Add two Basic Drones to the Arena. Check for collisions with Lily Pad Obstacles and collisions between the two Drones. Check the Collision Counters increment accordingly. Add another Basic Drone and repeat until a maximum of eight Drones are present within the Arena.	The Collision Counters will increase accordingly with every collision.	The Collision Counters increase accordingly with every collision. Drone stuck on collision with Lily Pad Obstacle or other Drones cause the Collision Counters to increase rapidly.

Conclusion and Reflection:

During the course of the assignment I have improved my understanding of object oriented programming. I have achieved the aim of creating a graphical simulation which utilises inheritance through the use of a Drone class with multiple sub-classes. Inheritance is useful for saving memory as well as hiding information. Repeated code is not necessary when objects of a same type share behaviour. All Drones shared parameters and methods and as such these were declared once in the Drone class.

My understanding of Java has improved however I am still not fully able to use Java and Java FX. I was unsure how to use groups across classes and as a result instead of creating separate classes for the Arena, Canvas, and GUI, they were all incorporated into one class. This does not show good code layout however ultimately the application still worked. A lack of understanding and experience also impacted certain design choices. For example, I had tried to incorporate a timer into the simulation however could not get it to work with the timelines so had to leave it out completely. As such users would be unable to properly conduct experiments using the simulation as they would have no way to efficiently time the experiments.

As a whole I have appreciated the assignment as it has challenged me and made me think deeper about Java and object oriented programming. Personally I do not think my simulation is a good program as it doesn't seem to have any actual use, particularly due to the issues with inconsistent collision detection; however I cannot expect myself to produce any significant piece of work with relatively little experience.

With more experience with Java and coding I can expect the quality of my work to increase and consequently my personal satisfaction with each project.