 Even Semester (2024)

**Assignment Cover Letter**

**(Individual Work)**

**()**

**Student Information: Surname Given Names Student ID Number**

**1. Kim Morris 2440074174**

**Course Code :COMP6699 Course Name :Object Oriented Programming**

**Class Code :L2BC Name of Lecturer(s) :Jude Joseph Lamug Martinez**

**Major :CS**

**Title of Assignment : Fake Anki  
 (if any)**

**Type of Assignment :Final Project**

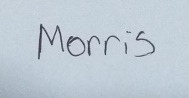
**Submission Pattern**

**Due Date :22-06-2020 Submission Date : 22-06-2020**

The assignment should meet the below requirements.  
 1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer’s instructions  
 2. Soft Copy assignment also requires the signed hardcopy) submission of this form, which automatically validates the softcopy  
 submission  
 3. The above information is complete and legible  
 4. Compiled pages are firmly stapled.  
 5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

**Plagiarism/Cheating**BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offences which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

**Declaration of Originality**  
By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

*Signature of Student: (Name of Student)  
1. Morris Kim*

\*) Delete the inappropriate option

**Table of contents**

Chapter I - Introduction…………………………………………………………………………………………………3

Chapter II – Project Specification…………………………………………………………………………………….4

Chapter III – Solution Design  
 1) UML Diagram & Program Structure………………………………………………………………..5  
 2) The Main Page……………………………………………………………………………………………….6  
 3) StudyBox……………………………………………………………………………………………………13  
 4) AddCard…………………………………………………………………………………………………….19  
 5) AddDeck…………………………………………………………………………………………………….21  
 6) Some Examples of a deck.txt………………………………………………………………………..23

Chapter I – Introduction

At first, I was not quite sure of what to make for this course’s final project – it was, after all, my final semester here and I have even considered not doing it at all. The reason for which is that I have already been accepted into another university, and since they did not allow the transfer of credits, I have to start over again from the first year (which I don’t mind). And so, I was stuck contemplating whether or not to do something that has no overall consequence in the foreseeable future. After giving it a second thought, I realized that although doing the project might be a waste of effort, *not* doing the project would be a definite waste of the tuition fees already put forward for this semester.

As for the idea for my project, it came from the time I have spent casually learning the Japanese language over the past year. Of course, it would be a lie to say my project is original. In fact, it is just ended up being an inferior version of this other existing application called *Anki* (which in Japanese means “memorization”). Hence, I eventually settled on the name *Fake Anki*. In the beginning, I was slightly more ambitious with the idea and planned to make something closer in terms of function and quality, but after receiving my vaccination shot and being bedridden for 3 days straight, I thought it would be best to not push myself too hard and risk missing the deadline.

In short, the idea of the project is “digital flash cards”. The program simply lets users create multiple digital decks and then store their digital flash cards inside the decks. Like flash cards in real life, you have a sort of ‘question’ that prompts your memory and then the ‘answer’ at the back so you can refresh said memory in the case you get it wrong. In the program, the user can navigate through each of their decks’ cards and then the ‘answer’ section can be shown/hidden using the press of a button.

The project’s development started around the 31st of May, but the next couple of weeks were used to complete the other projects, so most of the progress picked up from the 14th of June. *Fake Anki* uses JavaFX to create the GUI and IntelliJ for its development. The source code, the JavaFX software development kit, the video, and this final project report can be found in the github link below:

https://github.com/morris-necc/fake-anki

Chapter II – Project Specification

**Project Purpose:**

This project’s purpose is to let its users create and traverse through flash cards of their own creation. It is entirely possible to share decks between users manually, potentially letting users studying the same subject to help each other by showing each other their decks.

**Project Audience:**

This project’s main audience is students who are struggling to remember topics in class, students who want to make flash cards but are too lazy to do so.

**Project Aim:**

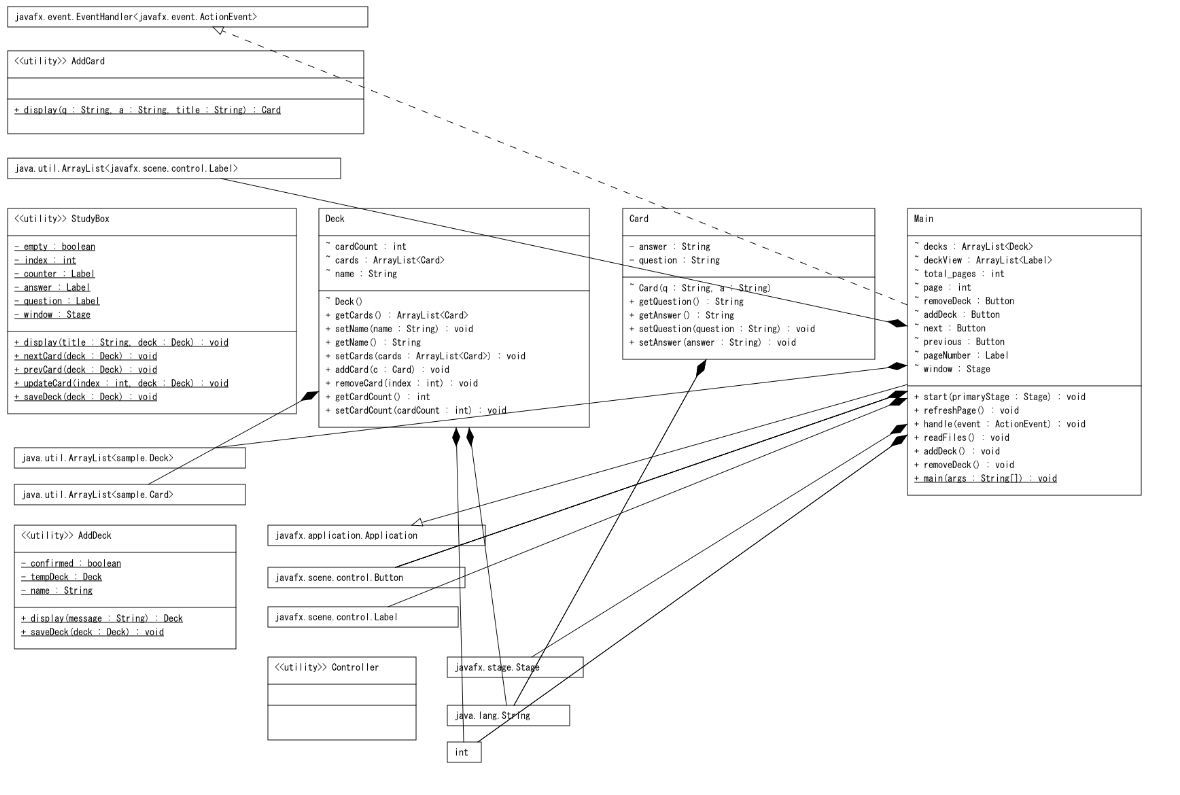
This project aims to aid its users to memorize by making it convenient to make flash cards without wasting as much time as making a physical one. Through this, studying can hopefully be made a less painful process.

**Project Requirements:**

* A main page to display all the users’ decks
* Buttons to let the user add and remove decks
* A page that lets the user study cards from a specific deck
* Buttons that let the user traverse through these cards
* Buttons that let the user add, remove, or edit cards
* A page that lets users create a new card by specifying the ‘question’ and ‘answer’ segments
* A saving/loading system so the user does not have to create the same deck over and over again when starting the program

Chapter III – Solution Design

**1. UML Diagram & Program Structure**



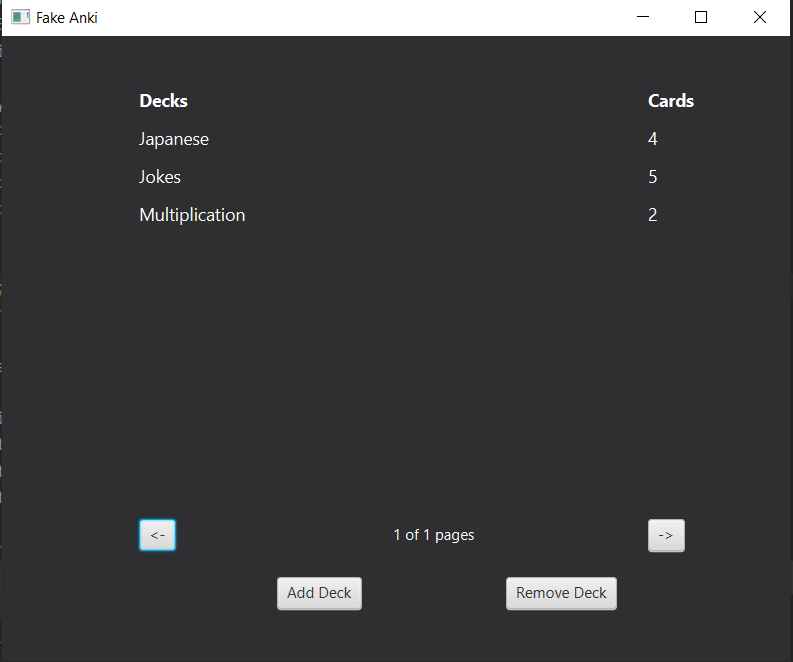
The above image is an automatically generated UML diagram using the IntelliJ plugin called “UML Generator”.

To put it simply, Main inherits from the JavaFX class called Application and implements a JavaFX interface called EventHandler. Like their names suggest, Application has to do with the creation of the application itself, and EventHandler has to do with handling events.

AddDeck, AddCard, and StudyBox are used to create the windows for adding a new deck, adding a new card, and the window for studying the cards respectively.

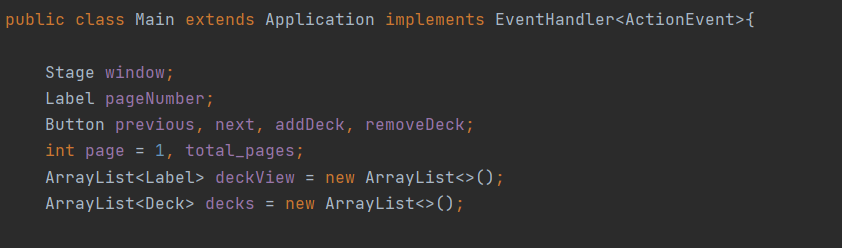
A Deck contains a name and an arraylist to store 0 or more Cards. A Card consists of 2 strings – a question and an answer. They both consist of basic getter and setter methods.

**2. The Main Page**

****

This is what it looks like.

Consists of several public attributes:



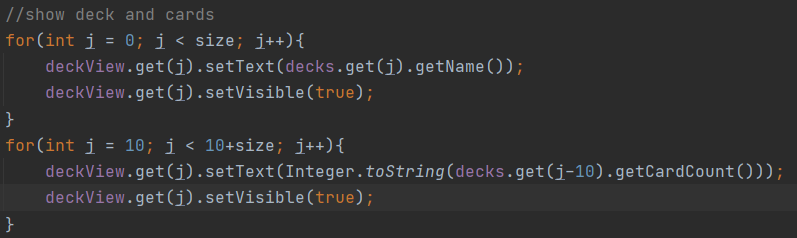
These variables were all declared outside the main scope because the event handler method is written within the same class, and accessing these variables for the event handler is made much easier this way.

The first method found within Main is the start() method. This method has to do with the structure and appearance of the main screen. It’s a very long section of code so it will be split up into different sections where the actual result will be shown for comparison.



This chunk of code sets the primary stage (conveniently named to ‘window’ as it is easier to understand that way), reads the deck files in order to load them onto the screen afterwards, initializes the gridpane layout that is used to place various items in the right place, and finally sets the headings for the deck names and the number of cards the deck has.



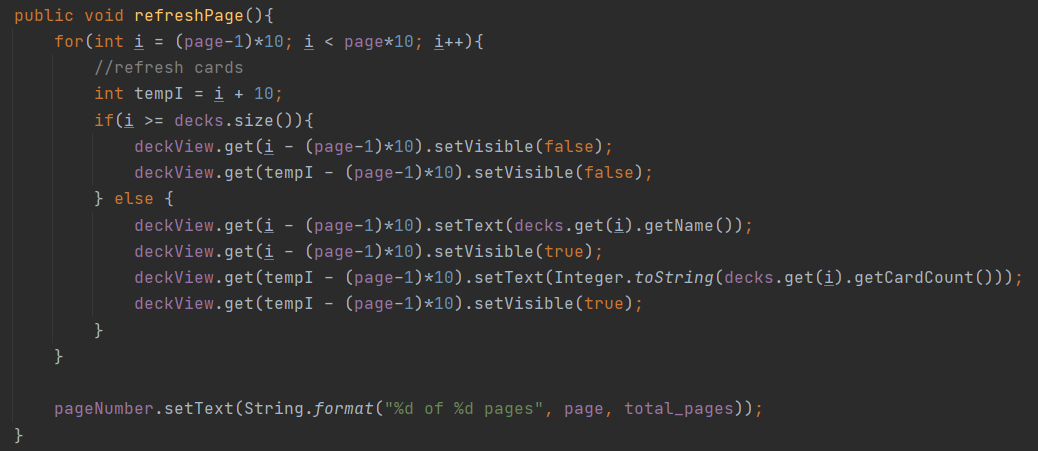


This part initializes the positions of the decks and cards within the page. One page can hold a maximum of 10 decks. The labels are all set to invisible initially, and then set to visible if there is a deck that can fill the label’s place. The part in the middle of the upper picture sets an onclick event for every name label for the deck so that they lead to the StudyBox window. When the StudyBox window is closed, the page is refreshed to show any new changes done (e.g. new cards, removed cards, etc.)

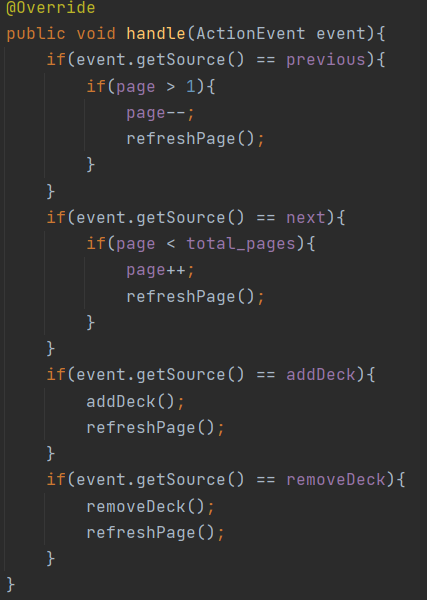


Finally, this chunk of code initializes the buttons and the page counter at the bottom of the screen. The event handling of the buttons will be covered later in this report. The last separated block of code in the picture puts all those elements in from above into the grid and then puts that grid into a scene and then sets that as the window’s scene. Essentially, this just displays all the elements initialized earlier.

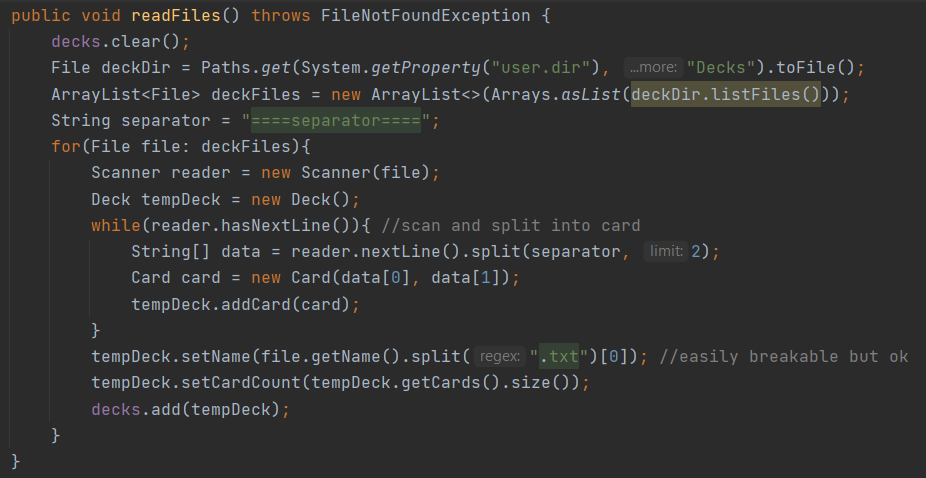
The second method found in the main file is refreshPage():



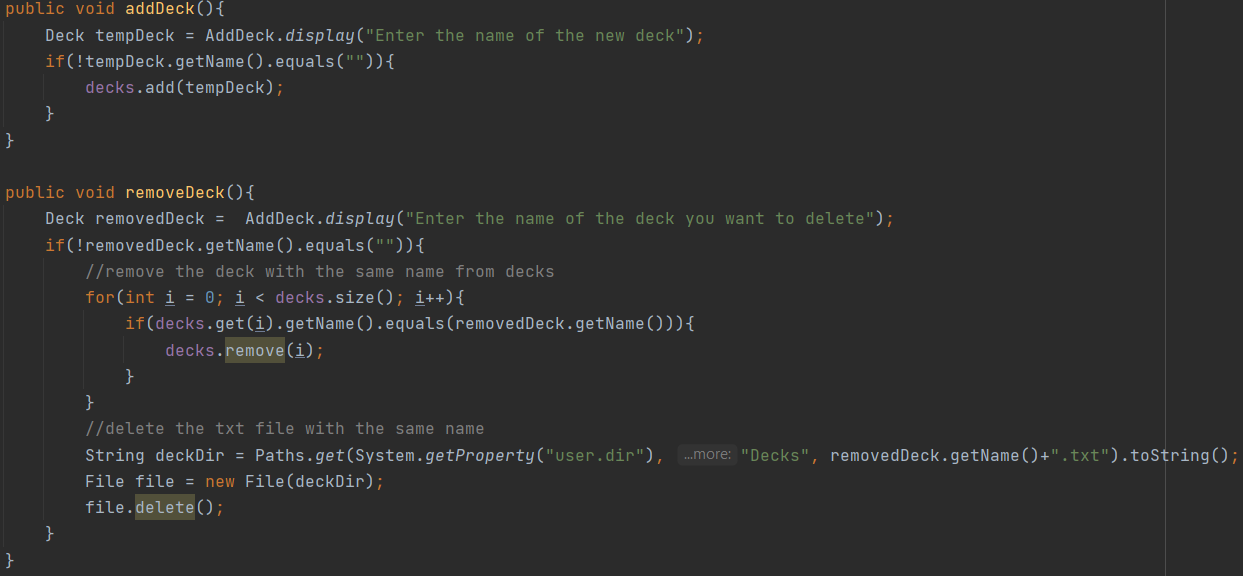
As its name suggests, it refreshes the list of decks on the main page. It does this by looking at what page it currently is and setting parts of the list to be visible or invisible based on whether or not there is a deck that can fill those parts. It also changes the names and the card count of these decks to reflect the new ones. Lastly, the page counter is also updated to show this new change.



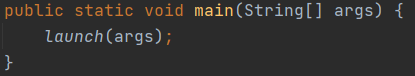
This method is quite self-explanatory – if the previous button is pressed, it goes to the previous page, if the add deck button is pressed then it adds a deck, etc.



This method, like its name implies, reads through the Decks folder in the user’s directory and then reads all the .txt files in them. Every line is converted into a Card object and every Card object is then put inside a Deck object’s “cards” attribute. Every deck is then put inside another arraylist to keep track of all the decks. This arraylist is called ‘decks’.

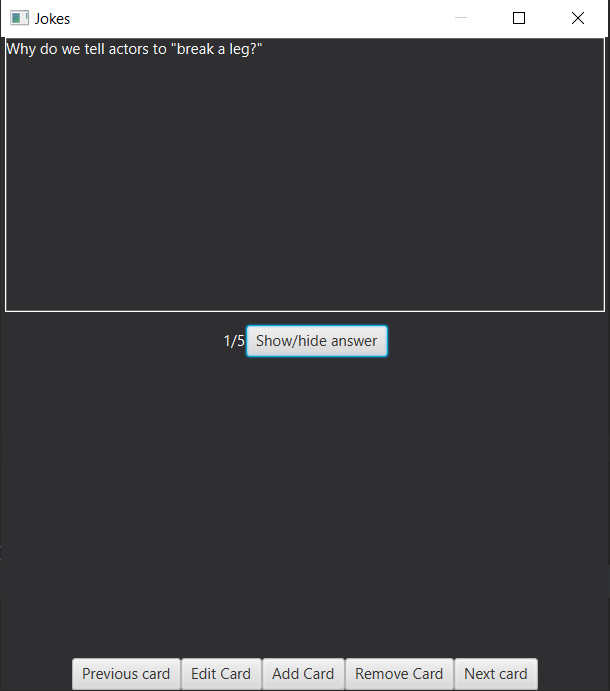


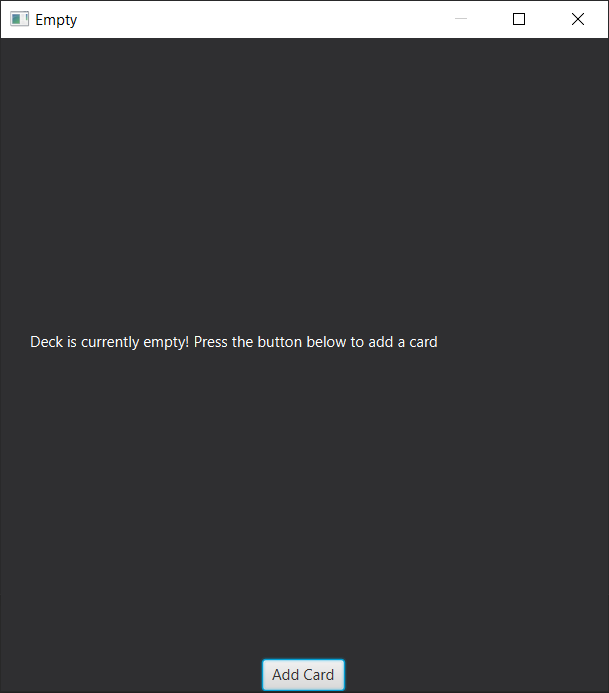
This section deals with the adding and removal of decks. In both cases, the same prompt will show up (AddDeck) bearing a different message depending on the context. A deck object is returned based on the user’s input and then the program will add/remove a deck accordingly. The deletion process is slightly more complicated as it has to look for the deck within decks.



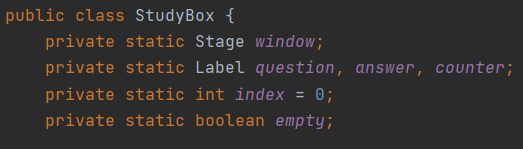
Last, but not least, the main method launches the program by running the launch() method within derived from the Application superclass.

**3. StudyBox**

****

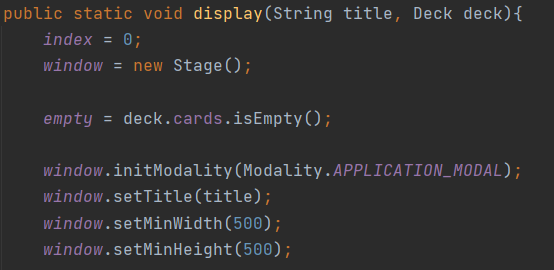
****

Like the images above suggest, there are 2 main states StudyBox can be in. The first state is if there is at least 1 card present within the deck, and the other state happens when the deck is empty.



In StudyBox, there are several variables that are declared outside the local scope of the display() method which is used to call StudyBox. Like the main method, some of these have to do with event handling and some of the other methods in StudyBox.

The first method of this class is the display method, and it takes in 2 arguments for the title of the window and the deck the user is going to be studying respectively.



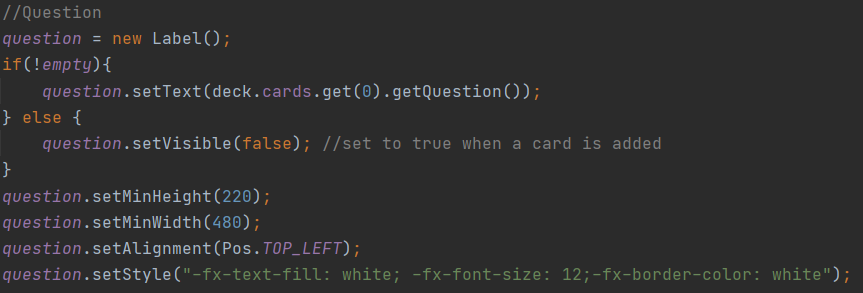
The window is made to be modal. This means that the user cannot interact with the main screen unless this StudyBox screen is closed. This is done to avoid actions that could break the program. For example, the user could potentially delete the deck while this window is active if the window is not modal.

StudyBox’s structure is made up of 4 major sections:  
i. Question  
ii. Counter & Show/Hide button

iii. Answer

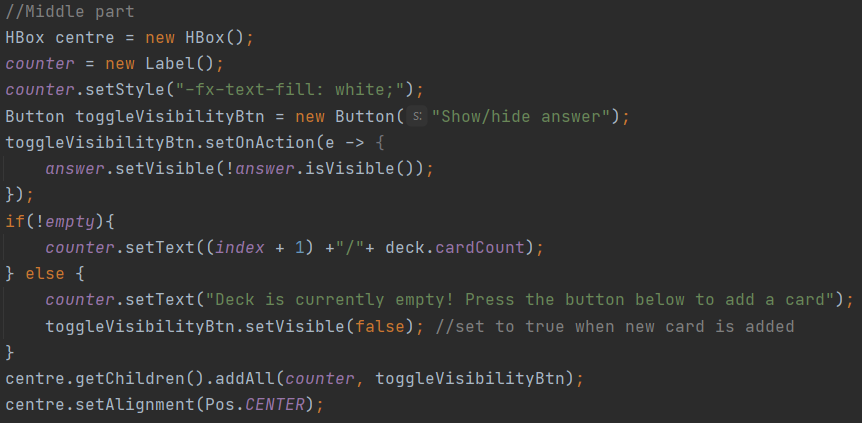
iv. Buttons

I. Question



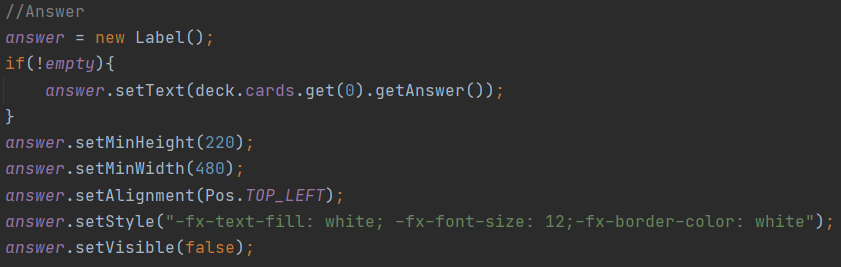
This displays the question of the current card the user is looking at in the deck. If the deck is empty, this section is set to invisible.

II. Counter and Show/Hide button



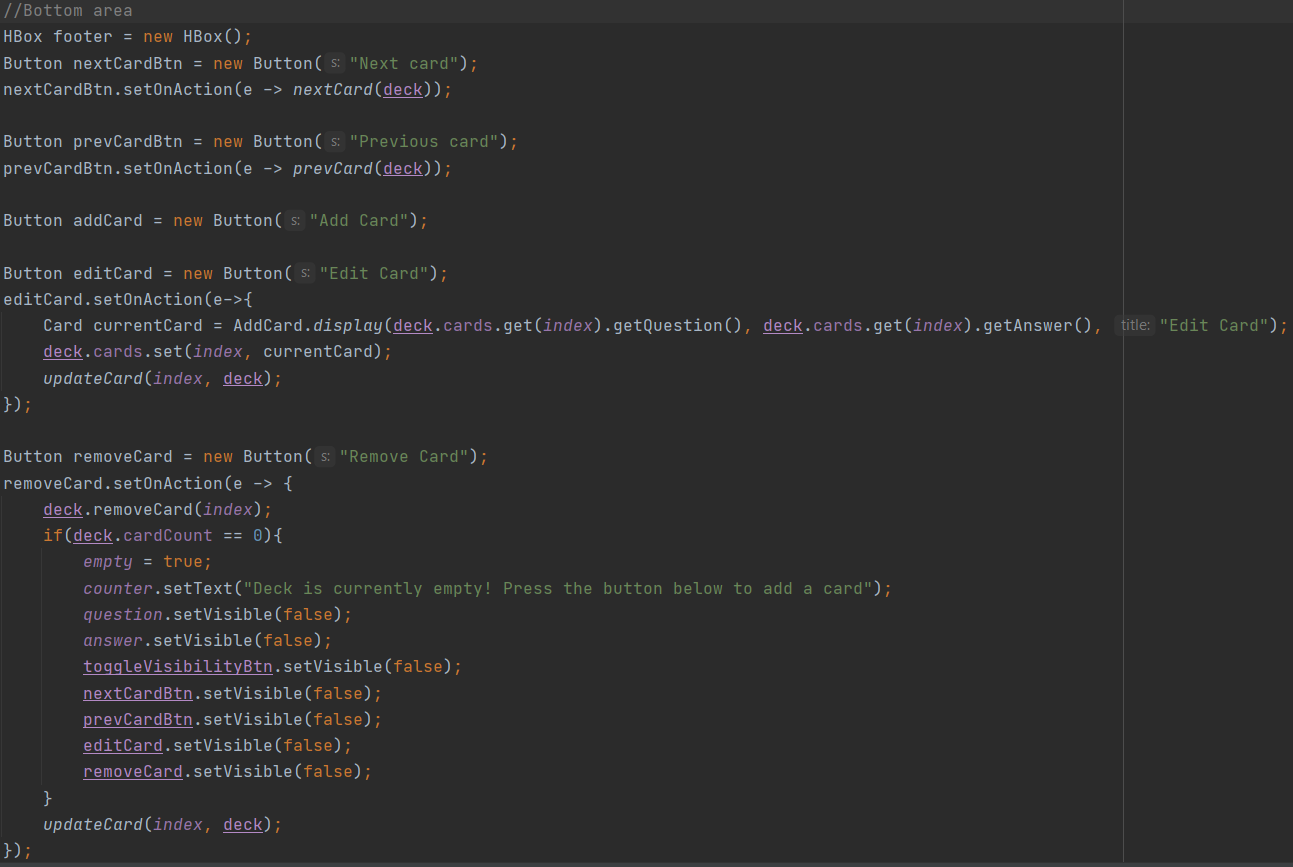
This section adds a counter to keep track of the current card out of the total number of cards as well as add the toggle visibility button for the answer. When this button is clicked, the answer section appears/disappears. If the deck is empty, the button is set to invisible and the counter is changed to show some text that tells the user to add some cards to fill the deck.

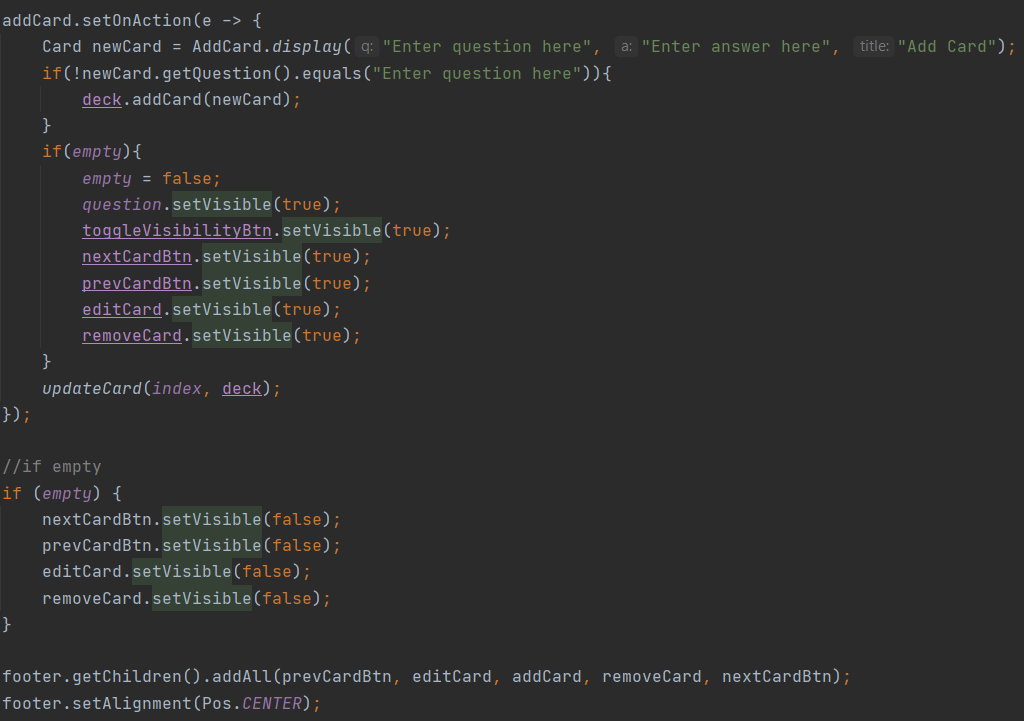
III. Answer



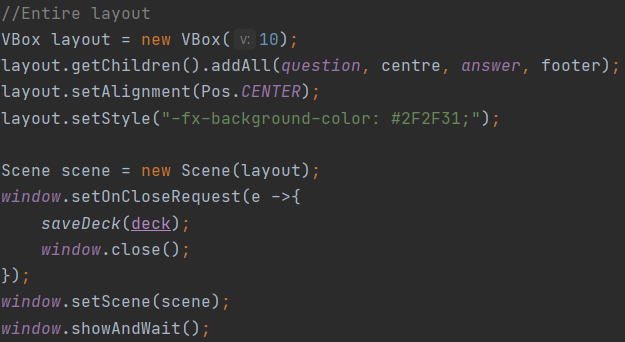
The answer section is similar to the question section with the exception that it is set to invisible at the start as to not let the users accidentally see the answer first.

IV. Buttons

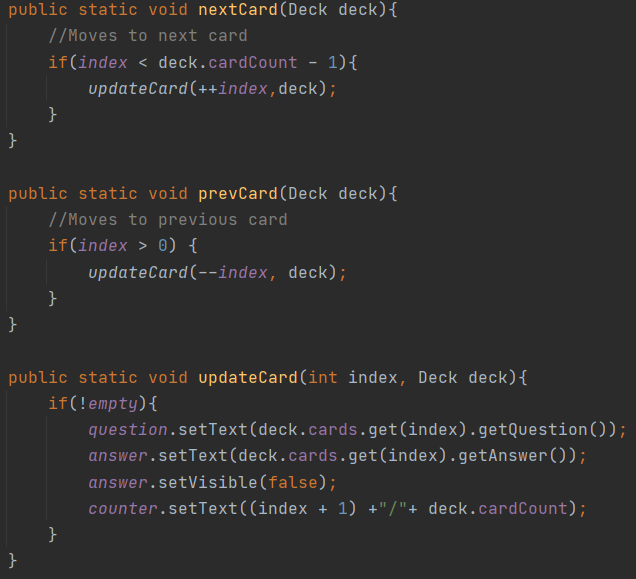




Unlike the Main file, there was no event handler so the code was slightly lengthier than usual, but essentially this adds several buttons to a HBox layout at the bottom of the window. The buttons editCard and addCard both call the display function of AddCard, which returns a Card object. This card object is then used to either alter the current card or add a new card to the end of the deck. It was done this way to make the code more efficient. The removeCard button removes the current card from the deck. A special condition is put if the user deletes the only remaining card in the deck that switches the StudyBox window into its 2nd state. Likewise, addCard also has a special condition that switches the StudyBox window into its 1st state when the user adds a card to an empty deck.



This last section of the display() method displays all the aforementioned elements but also saves the current deck when the window is closed.

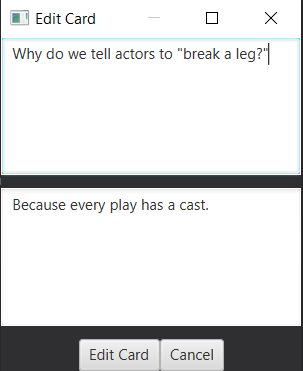
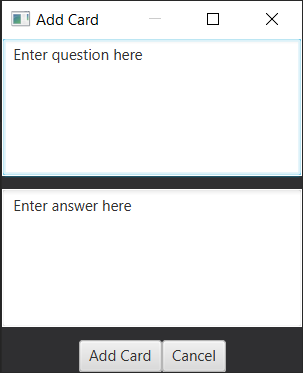


These 3 methods have to do with the nextCardBtn and prevCardBtn buttons. They update the index of the card the user is currently at in the deck and also update the visuals to reflect that change. The answer section of the card is again reset to be invisible so that the user does not accidentally see the answer because that defeats the whole purpose of having flash cards.

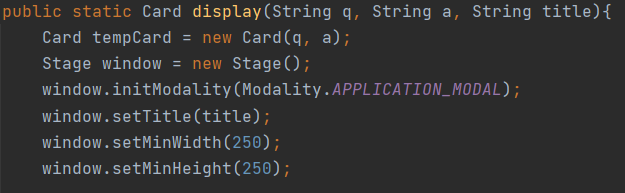


This is a method that honestly belongs within the Deck class, but I put it here for some reason. What it does is that it saves the deck into a .txt file of the same name. This is done by looping through the cards inside the deck and writing the string form of that Card into the .txt file.

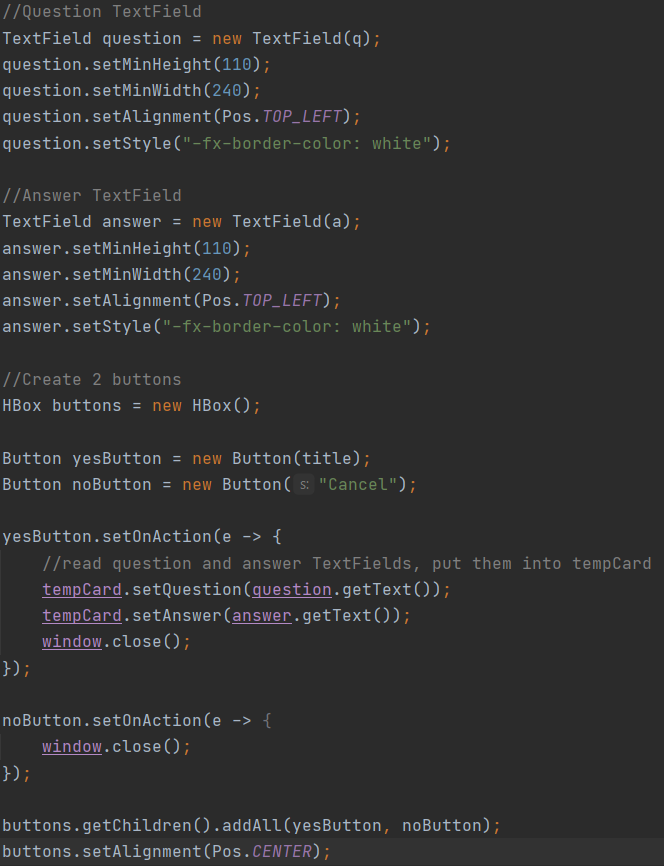
**4. AddCard**



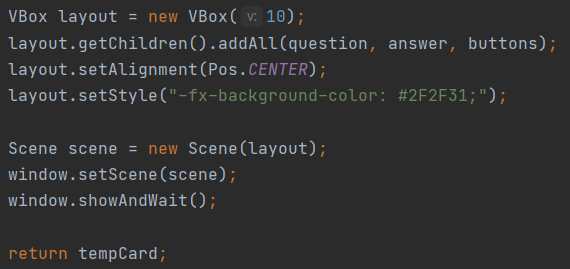
AddCard has 2 states depending on the context. If the user clicked the button to add a card, the first state will show up. If the user clicked the button to edit a card, the 2nd state will show up.



Like StudyBox, AddCard is also modal.

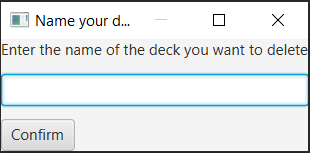
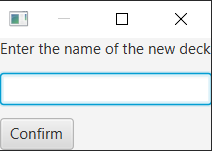


This window has 3 main parts: The question, the answer, and the buttons below. The text inside question and answer changes depending on whether the user picked add card or edit card. If the user picked add card, it will simply contain a text to prompt the user to fill the textfields but if the user picked edit, it will be filled with data from the card the user is editing. The yesButton and noButton refer to the confirm button and cancel button respectively. yesButton’s text changes to “Add Card” and “Edit Card” depending on the context as well. Clicking this button saves the card and adds/edits it while clicking the cancel button just closes the window.

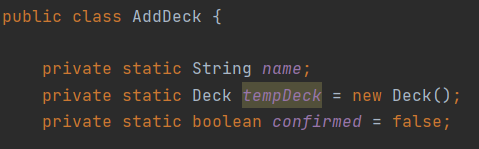


For the sake of completion, this makes all those elements we mentioned previously actually show up on the window.

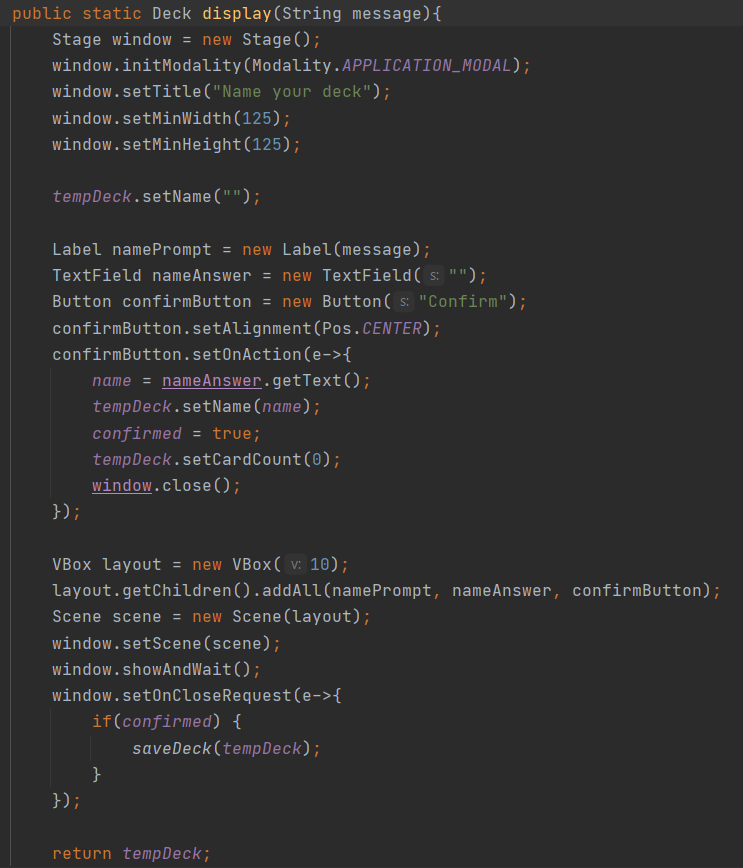
**5. AddDeck**

****

Similar to the above section, AddDeck also has 2 states depending on whether the user picked to add a deck or to delete a deck.



These are the attributes of AddDeck that are used in the 2 methods found inside AddDeck.

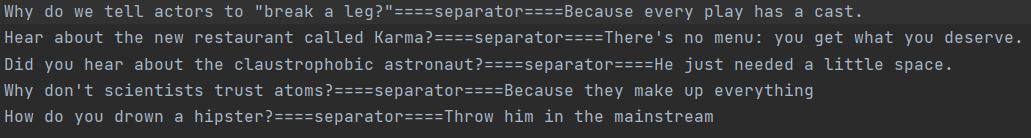
****

Since the window was relatively small with only 3 parts, the code to create it was pretty simple. Like with any other window except the main one, this window is set to modal. The 3 parts inside is a label that prompts the user to write the name of the deck they want to add/delete, a textfield to input said name, and a confirm button to save that name and close the window. The default exit button will not save the name and is synonymous with a cancel button.



Lastly, AddDeck has a method to save an empty deck/overwrite a deck so that it becomes empty. If the user selected to delete a deck, the deck will be deleted using removeDeck() in the main file.

**6. Some examples of a deck.txt**

****

